SAGI RAMA KRISHNAM RAJU ENGINEERING COLLEGE
(AUTONOMOUS)
Approved by AICTE & Affiliated to JNTUK, Kakinada
Bhimavaram, West Godavari Dist. – 534 204, Andhra Pradesh, India.

Student Notebook	
Department	Information Technology
Year / Semester	III B.Tech (IT) – I Semester
Subject	OPERATING SYSTEM LAB
Regulation	R20
Subject Code	B20IT3110

Vision
• To emerge as a world-class technical institution that strives for the socio-ecological well-being of the society.

SAGI RAMA KRISHNAM RAJU ENGINEERING COLLEGE :: BHIMAVARAM
(AUTONOMOUS)
DEPARTMENT OF INFORMATION TECHNOLOGY

Vision:

To evolve as a centre of excellence by adopting innovative methods for teaching learning and research in the diversified fields of Information Technology.

Mission:

- The highest quality technology based education and services in the most effective manner.
- Maintain a vital, state-of-art research to provide its students and faculty with opportunities to create, interpret, apply and disseminate knowledge.
- Empower students towards higher education, Research and becoming Entrepreneur / Employee and meet intellectual, ethical, carrier challenges and community service.

Program Educational Objectives (PEOs):

I. To provide graduates with a good foundation in mathematics, sciences, Information Technology and engineering fundamentals required to solve engineering problems that will facilitate them to find employment in industry and / or to pursue postgraduate studies with an appreciation for lifelong learning.

II. To provide graduates with analytical and problem solving skills to design algorithms, hardware / software systems and inculcate professional ethics, inter-personal skills to work in a multi-cultural team.

III. To facilitate graduates get familiarized with state of the art software / hardware tools, imbibing creativity and Innovation that would enable them to develop cutting-edge technologies of multi-disciplinary nature for societal development.

SAGI RAMA KRISHNAM RAJU ENGINEERING COLLEGE :: BHIMAVARAM
(AUTONOMOUS)
DEPARTMENT OF INFORMATION TECHNOLOGY

Program Outcomes (POs):

PO 1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO 2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO 3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO 4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO 5. **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.

PO 6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO 7. **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.

PO 8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO 9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO 10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO 11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO 12. **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(PSOs):

PSO 1. Apply core Information Technologies of System Architecture, information management, programming, networking for the development of current technical concepts

PSO 2. Integrate IT-based solutions into the user environment.

OPERATING SYSTEM LAB

Subject Code: B20IT3110 L T P C

III Year / I Semester 0 0 3 1.5

Pre-Requisites:

Students must be familiar with Computer Organization and programming Language.

Course Objectives:

To understand the design aspects of operating system
To study the process management concepts & Techniques
To study the storage management concepts
To familiarize students with the Linux environment
To learn the fundamentals of shell scripting/programming

Experiments:

Exercise 1:

a) Study of Unix/Linux general purpose utility command list: man,who,cat, cd, cp, ps,ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown.
b) Study of vi editor
c) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system
d) Study of Unix/Linux file system (tree structure)
e) Study of .bashrc, /etc/bashrc and Environment variables

Exercise 2:

Write a C program that makes a copy of a file using standard I/O, and system calls

Exercise 3:

Write a C program to emulate the UNIX ls –l command.

Exercise 4:

Write a C program that illustrates how to execute two commands concurrently with a command pipe. Ex: - ls –l | sort

Exercise 5:

a) Simulate the following CPU scheduling algorithms:
b) Round Robin (b) SJF (c) FCFS (d) Priority

Exercise 6:

a) Multiprogramming-Memory management-Implementation of fork (), wait (), exec() and exit (), System calls

Exercise 7:

Simulate the following:

a) Multiprogramming with a fixed number of tasks (MFT)

b) Multiprogramming with a variable number of tasks (MVT)

Exercise 8:

Simulate Bankers Algorithm for Dead Lock Avoidance

Exercise 9:

Simulate Bankers Algorithm for Dead Lock Prevention.

Exercise 10:

Simulate the following page replacement algorithms:

a) FIFO b) LRU c) LFU

Exercise 11:

Simulate the following File allocation strategies

(a) Sequenced (b) Indexed (c) Linked

Exercise 12:

Write a C program that illustrates two processes communicating using shared memory.

Exercise 13:

Write a C program to simulate producer and consumer problem using semaphores

Exercise 14:

Write C program to create a thread using pthreads library and let it run its function.

Exercise 15:

Write a C program to illustrate concurrent execution of threads using pthreads library.

Course Outcomes:

CO 1 Use Unix utilities and perform basic shell control of the utilities.

CO 2 Design different system calls for writing application programs.

CO 3 Implement various scheduling, page replacement algorithms and algorithms related to deadlocks.

CO 4 Design programs for shared memory management and semaphores.

TEXT BOOKS:

1. OPERATING SYSTEM, 2/e, Richard F, Gilberg , Forouzan, Cengage
2. OPERATING SYSTEM using C, Aaron M. Tenenbaum, Yedidyah Langsam, Moshe J Augenstein, Pearson, 2nd Edition.
3. OPERATING SYSTEM and Algorithm Analysis in C, Mark Allen Weiss, Pearson Education. Ltd., Second Edition

REFERENCE BOOKS:

1. Silberschatz A, Galvin P B, and Gagne G, Operating System Concepts, 9th edition,Wiley, 2013.
2. Tanenbaum A S, Modern Operating Systems, 3rd edition, Pearson Education, 2008.(for Interprocess Communication and File systems.)

SAGI RAMA KRISHNAM RAJU ENGINEERING COLLEGE
(AUTONOMOUS)
DEPARTMENT OF INFORMATION TECHNOLOGY

Student Notebook	
Department	Information Technology
Year / Semester	III B.Tech (IT) – I Semester
Subject	OPERATING SYSTEM LAB
Regulation	R20
Subject Code	B20IT3110

Student Performance Evaluation

Week / Exercise Number	Date	Marks				Signature of Faculty	Remarks
		Lab (5)	Record (5)	Viva (5)	Total (15)		
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							

Total Marks Awarded: _____ (In Words : _____)

Signature of the Faculty **Signature of HOD**

SAGI RAMA KRISHNAM RAJU ENGINEERING COLLEGE
(AUTONOMOUS)
DEPARTMENT OF INFORMATION TECHNOLOGY

Student Notebook	
Department	Information Technology
Year / Semester	III B.Tech (IT) – I Semester
Subject	OPERATING SYSTEM LAB
Regulation	R20
Subject Code	B20IT3110

Ex.No:1.	BASICS OF UNIX COMMANDS
	INTRODUCTION TO UNIX

AIM:
To study about the basics of UNIX

UNIX:
It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS.
By1980, UNIX had been completely rewritten using C language.

LINUX:
It is similar to UNIX, which is created by Linus Torualds. All UNIX commands works in Linux. Linux is a open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUXSYSTEM:
It consists of three parts.

a. UNIX kernel
b. Shells
c. Tools and Applications

UNIX KERNEL:
Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one programs tops and another starts.

SHELL:
Shell is the command interpreter in the UNIX OS. It accepts command from the userand analyses and interprets them

Ex.No:1.	BASICS OF UNIX COMMANDS
	BASIC UNIX COMMANDS

AIM:

To study of Basic UNIX Commands and various UNIX editors such as vi, ed, exand EMACS.

CONTENT:
Note: Syn->Syntax

a) date–used to check the date and time

Syn:$date

Format	Purpose	Example	Result
+%m	To display only month	$date+%m	06
+%h	To display month name	$date+%h	June
+%d	To display day of month	$date+%d	O1
+%y	To display last two digits of years	$date+%y	09
+%H	To display hours	$date+%H	10
+%M	To display minutes	$date+%M	45
+%S	To display seconds	$date+%S	55

b) cal–used to display the calendar

Syn:$cal 2 2009

c)echo–used to print the message on the screen.

Syn:$echo "text"

d) ls–used to list the files. Your files are kept in a directory.

Syn:$ls–s
All files (include files with prefix)
ls–l Lodetai (provide file statistics)
ls–t Order by creation time
ls– u Sort by access time (or show when last accessed together with –l)
ls–s Order by size
ls–r Reverse order
ls–f Mark directories with /,executable with* , symbolic links with @, local sockets with =, named pipes(FIFOs)with
ls–s Show file size
ls– h" Human Readable", show file size in Kilo Bytes & Mega Bytes (h can be used together with –l or)

ls[a-m]*List all the files whose name begin with alphabets From „a‟ to „m‟
ls[a]*List all the files whose name begins with „a‟ or „A‟
Eg:$ls>my list Output of „ls‟ command is stored to disk file named „my list‟

e)lp–used to take printouts

Syn:$lp filename

f)man–used to provide manual help on every UNIX commands.

Syn:$man unix command
$man cat

g)who & **whoami**–it displays data about all users who have logged into the system currently. The next commanddisplays about current user only.

Syn:whowhoami

h) uptime–tells you how long the computer has been running since its last reboot or power-off.

Syn:$uptime

i)uname–it displays the system information such as hardware platform, system name and processor, OS type.

Syn:$uname–a

j) hostname–displays and set system host name

Syn:$ hostname

k) bc–stands for "best calculator‟

$bc 10/2*3 15	$ bc scale =1 2.25+1 3.35 quit	$ bc ibase=2 obase=16 11010011 89275 1010 Ā Quit	$ bc sqrt(196) 14 quit

$bc for(i=1;i<3;i=i+1)I 1 2 3 quit	$ bc-l scale=2 s(3.14) 0	

FILE MANIPULATION COMMANDS

a) **cat**–this create, view and concatenate files.

Creation:

Syn:$cat>filename

Viewing:

Syn:$cat filename

Add text to an existing file:

Syn:$cat>>filename

Concatenate:

Syn:$catfile1file2>file3

$catfile1file2>>file3 (no over writing of file3)

b) **grep**–used to search a particular word or pattern related to that word from the file.Syn:$grep search word filename

Eg:$grep anu student

c) **rm**–deletes a file from the file systemSyn:$rm filename

d) **touch**–used to create a blank file.

Syn:$touch file names

e) **cp**–copies the files or directoriesSyn:$cpsource file destination fileEg:$cp student stud

f) **mv**–to rename the file or directorysyn:$mv old file new file

Eg:$mv–i student student list(-i prompt when overwrite)

g) **cut**–it cuts or pickup a given number of character or fields of the file.Syn:$cut<option><filename>

Eg: $cut –c filename

$cut–c1-10emp

$cut–f 3,6emp

$ cut –f 3-6 emp

-c cutting columns

-f cutting fields

h) **head**–displays10 lines from the head(top)of a given fileSyn:$head filename

Eg:$head student To display the top two lines:

Syn:$head-2student

i) **tail**–displays last 10 lines of the fileSyn:$tail filename
Eg:$tail student
To display the bottom two lines;
Syn:$ tail -2 student

j) **chmod**–used to change the permissions of a file or directory.Syn:$ch mod category operation permission file Where, Category–is the user type
Operation–is used to assign or remove permission
Permission–is the type of permission
File–are used to assign or remove permission all

Examples:
$chmodu-wx student
Removes write and execute permission for users
$ch modu+rw,g+rwstudent
Assigns read and write permission for users and groups
$chmodg=rwx student
Assigns absolute permission for groups of all read, write and execute permissions

k) **wc**–it counts the number of lines, words, character in a specified file(s)with the options as –l,-w,-c

Category	Operation	Permission
u– users g–group o– others	+assign -remove =assign absolutely	r– read w– write x- execute

Syn: $wc –l filename
$wc –w filename
$wc–c filename

Ex.No:1.	BASICS OF UNIX COMMANDS
	UNIX EDITORS

AIM:
To study of various UNIX editors such as vi, ed, ex and EMACS.

CONCEPT:
Editor is a program that allows user to see a portions a file on the screen and modify characters and lines by simply typing at the current position. UNIX supports variety of Editors. They are:
ed ex vi EMACS
Vi- vi is stands for "visual".vi is the most important and powerful editor.vi is a full screen editor that allows user to view and edit entire document at the same time.vi editor was written in the University of California, at Berkley by Bill Joy, who is one of the co-founder of Sun Microsystems.

Features of vi:
It is easy to learn and has more powerful features.
Itworksgreatspeedandiscasesensitive.vihaspowerfulundofunctionsandhas3modes:
1. Command mode
2. Insert mode
3. Escape or ex mode

In command mode, no text is displayed on the screen.
In Insert mode, it permits user to edit insert or replace text.In escape mode, it displays commands at command line.
Moving the cursor with the help of h, l, k, j, I, etc

EMACS Editor
Motion Commands:
M-> Move to end of file
M-< Move to beginning of file
C-v Move forward a screen M –v
Move backward a screen C –n Move to next lineC-p Move to previous line
C-a Move to the beginning of the lineC-e Move to the end of the line
C-f Move forward a character C-b Move backward a characterM-f
Move forward a word
M-b Move backward a word

Deletion Commands:

DEL	delete the previous
character C -d	delete the current
character M -DEL	delete the previous
word	
M-d	delete the next word
C-x DEL	deletes the previous sentence
M-k	delete the rest of the current sentence
C-k	deletes the rest of the current line
C-xu	undo the lasted it change

Search and Replace in EMACS:

y	Change the occurrence of the pattern
n	Don‟t change the occurrence, but look for the other q Don‟t change. Leave queryreplace completely
!	Change this occurrence and all others in the file

Ex.No:2.	Write a C program that makes a copy of a file using standard I/O, and system calls

By using System calls:

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
main(int agrc,char *argv[])
{
char buf[20];
int fd1,fd2,n;
fd1=open(argv[1],O_RDONLY);
fd2=open(argv[2],O_WRONLY);
n=read(fd1,buf,sizeof(buf));
write(fd2,buf,n);
close(fd1);
close(fd2);
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out student class
[dmgv@LinuxServer ~]$ cat student

hai

[dmgv@LinuxServer ~]$ cat class

hai
```

By using Standard I/O functions:

```
#include&lt;stdio.h&gt;
int main(int argc,char *argv[])
{
FILE *fp1,*fp2;
int ch;
fp1=fopen(argv[1],"r");
fp2=fopen(argv[2],"w");
while((ch=fgetc(fp1))!=-1)
fputc(ch,fp2);

}
```

Ex.No:3.	Write a C program to emulate the UNIX ls –l command.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
int main()
{
int pid;            //process id
pid = fork();     //create another process
if ( pid < 0 )
{                              //fail
printf("\nFork failed\n");
exit (-1);
}
else if ( pid == 0 )
{                   //child
execlp ( "/bin/ls", "ls", "-l", NULL );  //execute ls
}
else
{                           //parent
wait (NULL);              //wait for child
printf("\nchild complete\n");
exit (0);
}
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

total 100
-rwxrwx—x 1 guest-glcbls guest-glcbls 140 2012-07-06 14:55 f1
drwxrwxr-x 4 guest-glcbls guest-glcbls 140 2012-07-06 14:40 dir1
child complete
```

Ex.No:4.	**Write a C program that illustrates how to execute two commands concurrently with a command pipe.** **Ex: - ls –l | sort**

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

void main(int argc, char *argv[])
{
  int fd[2],pid,k;
  k=pipe(fd);

if(k==-1)
  {
    perror("pipe");     exit(1);
  }
  pid=fork();
  if(pid==0)
  {
    close(fd[0]);
    dup2(fd[1],1);
    close(fd[1]);
    execlp(argv[1],argv[1],NULL);
    perror("exec1");
  }
  else
  {
   wait(2);
   close(fd[1]);
   dup2(fd[0],0);
   close(fd[0]);
   execlp(argv[2],argv[2],NULL);
   perror("exec1");
 }
}
```

(or)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
int main()
{
```

```
int pfds[2];
char buf[30];
if(pipe(pfds)==-1)
{
perror("pipe failed");
exit(1);
}
if(!fork())
{
close(1);
dup(pfds[1];
system (“ls –l”);
}
else
{
printf("parent reading from pipe \n");
while(read(pfds[0],buf,80))
printf("%s \n" ,buf);
}
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char** argv) {

    int fd[2];
    int pid;
    char read_buf[READ_SIZE + 1];
    int chars_read;

    /*
     * Check for adequate args.
     */
    if (argc < 1) {
        fprintf(stderr, "usage: grab-stdout prog arg ...");
        exit(-1);
    }

    /*
     * Create a pipe that will have its output written to by the executed
     * program.
     */
    pipe(fd);

    /*
     * Fork off a process to exec the program.
     */
    if ((pid = fork()) < 0) {
        perror("fork");
  exit(-1);
    }

    /*
     * Forked child has its stdout be the write end of the pipe.
     */
    else if (pid == 0) {

        /*
         * Don't need read end of pipe in child.
         */
        close(fd[0]);

        /*
         * This use of dup2 makes the output end of the pipe be stdout.
         */
        dup2(fd[1], STDOUT_FILENO);

        /*
```

```
         * Don't need fd[1] after the dup2.
         */
        close(fd[1]);

        /*
         * Exec the program given in the command line, including any args.
         */

        execlp("ls", "ls", 0);

        perror("exec1");
        exit(-1);
    }

    /*
     * Parent takes its input from the read end of the pipe.
     */
    else {
        /*
         * Don't need write end of pipe in parent.
         */
        close(fd[1]);

        dup2(fd[0], STDIN_FILENO);

        close(fd[0]);

        //I'm unsure of what to do here but this is all I could think of
        execlp("sort", "sort", "-r" );

        perror("exec2");
        exit(-1);

    }

    exit(0);
}
```

OUTPUT:

[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

Parent reading from pipe
Total 24
-rwxrwxr-x l student student 5563Aug 3 10:39 a.out
-rw-rw-r—l
Student student 340 jul 27 10:45 pipe2.c
-rw-rw-r—l student student
Pipes2.c
-rw-rw-r—l student student 401 34127 10:27 pipe2.c
student

Ex.No:5.	**Simulate the following CPU scheduling algorithms:** **(a) FCFS (b) SJF (c) Priority (d) Round Robin**

FCFS

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

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 name and the burst time
Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time
Step 5: 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 6: Calculate
a)Average waiting time = Total waiting Time / Number of process

b)Average Turnaround time = Total Turnaround Time / Number of process Step 7: Stop the process

SOURCE CODE:

```
#include<stdio.h> #include<conio.h> main()
{
int bt[20], wt[20], tat[20], i, n; float wtavg, tatavg;
clrscr();
printf("\nEnterthe number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\nEnter Burst Time for Process %d -- ", i); scanf("%d", &bt[i]);
}
```

```
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];

for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out
```

INPUT

Enter the number of processes --	3
Enter Burst Time for Process 0 --	24
Enter Burst Time for Process 1 --	3
Enter Burst Time for Process 2 --	3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30
Average Waiting Time--	17.000000		
Average Turnaround Time --		27.000000	

SJF

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

SOURCE CODE :

```
#include<stdio.h> #include<conio.h> main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- "); scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i); scanf("%d", &bt[i]);

}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;

temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
```

```
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out
```

INPUT

Enter the number of processes --	4
Enter Burst Time for Process 0 --	6
Enter Burst Time for Process 1 --	8
Enter Burst Time for Process 2 --	7
Enter Burst Time for Process 3 --	3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24
Average Waiting Time --		7.000000	
Average Turnaround Time --		13.000000	

Priority:

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

SOURCE CODE:

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

```
if(pri[i] > pri[k])
{
temp=p[i]; //process swapping
p[i]=p[k];
p[k]=temp;

temp=bt[i];  // burst time swapping
bt[i]=bt[k];
bt[k]=temp;

temp=pri[i]; // priority swapping
pri[i]=pri[k];
pri[k]=temp;
}
wtavg = wt[0] = 0;
tatavg = tat[0] = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] + bt[i-1];
tat[i] = tat[i-1] + bt[i];

wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\nPROCESS\t\tPRIORITY\tBURST    TIME\tWAITING    TIME\tTURNAROUND
TIME");
for(i=0;i<n;i++)
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]);
printf("\nAverage Waiting Time is --- %f",wtavg/n);
printf("\nAverage Turnaround Time is --- %f",tatavg/n);
getch();
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out
```

INPUT

```
Enter the number of processes -- 5
Enter the Burst Time & Priority of Process 0 --- 10        3
Enter the Burst Time & Priority of Process 1 --- 1         1
Enter the Burst Time & Priority of Process 2 --- 2         4
Enter the Burst Time & Priority of Process 3 --- 1         5
Enter the Burst Time & Priority of Process 4 --- 5         2
```

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITIN G TIME	TURNARO UND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

Average Waiting Time is --- 8.200000

Average Turnaround Time is ---------- 12.000000

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 fromprocess(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 ofprocess Step 8: Stop the process

SOURCE CODE

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

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

```
getch();
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

Enter the no of processes – 3

Enter Burst Time for process 1 – 24
Enter Burst Time for process 2 -- 3
Enter Burst Time for process 3 – 3

Enter the size of time slice – 3
```

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUNDTIME
1	24	6	30
2	3	4	7
3	3	7	10

```
The Average Turnaround time is – 15.666667
The Average Waiting time is  5.666667
```

Ex.No:6.	Implementation of fork (), vfork, wait (), exec() and exit (), System calls

fork() System call

```
include<stdio.h>
#include<unistd.h>
main(void)
{
int pid;
pid=fork();
if(pid>=0)
{
if(pid==0)
{
printf("\n Child process id :%d",pid);
}
else
{
printf("\n Parent process id :%d",pid);
}
}
else
printf("\n Process is not created");
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out
Parent process id :6848
Child process :0
```

vfork() System call

```
#include<unistd.h>
#include<stdio.h>
int global=5;
main()
{
int pid,local=6;
pid=vfork();

if(pid==0)
{
```

```
global++;
local--;
exit(0);
}
printf("\n Global value is:%d \nLocal value is: %d",global,local);
exit(0);
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

Global value is:6
Local value is:5
```

wait() System call

```
#include<stdio.h>
#include<unistd.h>
main()
{
int pid,status,childpid;
printf(" parent process with PID is %d \n ",getpid());
pid=fork();
if(pid!=0)
{
 sleep(1);
 printf(" parent process with PID %d and PPID %d \n ",getpid(),getppid());
 childpid=wait(&status);
 printf("child process PID %d terminated with exit code %d\n",childpid,status>>8
);
}
else
{
 printf(" child process with PID %d and PPId %d \n ",getpid(),getppid());
 sleep(5);
 exit(42);
 }
 printf("PID %d terminates \n ",getpid());
}
```

Output:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

parent process with PID is 7242
child process with PID 7243 and PPId 7242
```

parent process with PID 7242 and PPID 6984
child process PID 7243 terminated with exit code 42
PID 7242 terminates

exec() System call

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

main( int argc,char *argv[])
{
if(fork()==0)
{
execvp(argv[1],&argv[1]);
fprintf(stderr,"couldnot execute %s \n",argv[1]);
}
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out  samplefilename.txt

Hello welcome to OS lab we are executing execvp() system call
```

execl() System call

```
#include<stdio.h>
main()
{
printf("i am process %d and iam about exec an ls -1\n",getpid());
execl("/bin/ls","ls","-1",NULL);
printf("this line should never be executed\n");
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out  ls


i am process 7189 and iam about exec an ls -1
a.out
armstrong.sh
banker.c
class
d1
d2
```

To display process ids and group ids using fork system call

```
#include<stdio.h>
#include<unistd.h>
main()
{
int pid;
pid=fork();
pid=getpid();
printf("\n child id=%d",pid);
pid=getppid();
printf("\n parent process id=%d",pid);
pid=getuid();
printf("\n user id=%d",pid);
pid=getgid();
printf("\n group id=%d",pid);
}
```

OUTPUT:

```
[dmgv@LinuxServer ~]$ cc filename.c
[dmgv@LinuxServer ~]$ ./a.out

child id=6810
 parent process id=6809
 user id=842
 group id=842
 child id=6809
 parent process id=6104
 user id=842
 group id=842
```

Ex.No:7.	**Simulate the following:** **1) Multiprogramming with a fixed number of tasks (MFT)** **2) Multiprogramming with a variable number of tasks (MVT)**

MFT

DESCRIPTION:

In this the memory is divided in two parts and process is fit into it. The process which is best suited will be placed in the particular memory where it suits. In MFT, the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. In MVT, each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

ALGORITHM:

Step1: Start the process.
Step2: Declarevariables.
Step3: Enter total memory size ms.
Step4: Allocate memory for os.
Ms=ms-os
Step5: Read the no partition to be divided n Partition size=ms/n. Step6: Read the process no and process size.
Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation.

```
if(pn[i]==pn[j])f=1;
if(f==0){ if(ps[i]<=siz)
{
extft=extft+size- ps[i];avail[i]=1; count++;
}
}
```

Step 8: Print the results

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
 main()
{
int ms, bs, nob, ef,n, mp[10],tif=0; int i,p=0;
clrscr();
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
printf("Enter the block size (in Bytes) -- ");
scanf("%d", &bs);
nob=ms/bs; ef=ms - nob*bs;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter memory required for process %d (in Bytes)-- ",i+1);
scanf("%d",&mp[i]);
}
printf("\nNo. of Blocks available in memory--%d",nob);
printf("\n\nPROCESS\tMEMORYREQUIRED\tALLOCATED\tINTERNAL
FRAGMENTATION");
for(i=0;i<n && p<nob;i++)
{
printf("\n %d\t\t%d",i+1,mp[i]); if(mp[i] > bs)
printf("\t\tNO\t\t---");
else
{
printf("\t\tYES\t%d",bs-mp[i]);
tif = tif + bs-mp[i];
p++;
}
}
if(i<n)
printf("\nMemory is Full, Remaining Processes cannot be accomodated");
printf("\n\nTotal Internal Fragmentation is %d",tif);
```

```
printf("\nTotal External Fragmentation is %d",ef);
getch();
}
```

INPUT

Enter the total memory available (in Bytes) -- 1000
Enter the block size (in Bytes)-- 300
Enter the number of processes – 5
Enter memory required for process 1 (in Bytes) -- 275
Enter memory required for process 2 (in Bytes) -- 400
Enter memory required for process 3 (in Bytes) -- 290
Enter memory required for process 4 (in Bytes) -- 293
Enter memory required for process 5 (in Bytes) -- 100
No. of Blocks available in memory -- 3

OUTPUT

PROCESS	MEMORY REQUIRED	ALLOCATED	INTERNAL FRAGMENTATION
1	275	YES	25
2	400	NO	-----
3	290	YES	10
4	293	YES	7

Memory is Full, Remaining Processes cannot be accommodated

TotalInternal Fragmentation is 42
Total External Fragmentation is 100

MVT

ALGORITHM:

Step1: start the process.
Step2: Declare variables.
Step3: Enter total memory size ms.
Step4: Allocate memory for os.
Ms=ms-os
Step5: Read the no partition to be divided n Partition size=ms/n.
Step6: Read the process no and process size.
Step 7: If process size is less than partition size allot alse blocke the process. While allocating update memory wastage-external fragmentation.
if(pn[i]==pn[j])
f=1;
if(f==0){ if(ps[i]<=size)
{
extft=extft+size- ps[i];
avail[i]=1;
count++;
}
}
Step 8: Print the results
Step 9: Stop the process.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
main()
{
int       ms,mp[10],i, temp,n=0;
char ch = 'y';
clrscr();
printf("\nEnter the total memory available (in Bytes)-- ");
scanf("%d",&ms);
temp=ms;
for(i=0;ch=='y';i++,n++)
{
```

```
printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);
scanf("%d",&mp[i]);
if(mp[i]<=temp)
{
printf("\nMemory is allocated for Process %d ",i+1);
temp = temp - mp[i];
}
else
{
printf("\nMemory is Full");
break;
}
printf("\nDo you want to continue(y/n) -- ");
scanf(" %c", &ch);
}
printf("\n\nTotal Memory Available -- %d", ms);
printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED ");
for(i=0;i<n;i++)
printf("\n \t%d\t\t%d",i+1,mp[i]);
printf("\n\nTotal Memory Allocated is %d",ms-temp);
 printf("\nTotal External Fragmentation is %d",temp);
getch();
}
```

OUTPUT:

Enter the total memory available (in Bytes) – 1000
Enter memory required for process 1 (in Bytes) – 400 Memory is allocated for Process 1
Do you want to continue(y/n) -- y
Enter memory required for process 2 (in Bytes) -- 275 Memory is allocated for Process 2
Do you want to continue(y/n) – y
Enter memory required for process 3 (in Bytes) – 550

Memory is Full

Total Memory Available – 1000

PROCESS	MEMORY ALLOCATED
1	400
2	275

Total Memory Allocated is 675
Total External Fragmentation is 325

Ex.No:8.	**Simulate Bankers Algorithm for Dead Lock Avoidance**

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.

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

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
int alloc[10][10],max[10][10];
 int avail[10],work[10],total[10];
int i,j,k,n,need[10][10];
int m;
int       count=0,c=0;
char     finish[10];
```

```
clrscr();
printf("Enter the no. of processes and resources:");
scanf("%d%d",&n,&m);
for(i=0;i<=n;i++)
finish[i]='n';
printf("Enter the claim matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<m;j++)
scanf("%d",&max[i][j]);
printf("Enter the allocation matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<m;j++) scanf("%d",&alloc[i][j]);
printf("Resource vector:");
for(i=0;i<m;i++)
scanf("%d",&total[i]);
for(i=0;i<m;i++)
avail[i]=0;
for(i=0;i<n;i++)
 for(j=0;j<m;j++)
avail[j]+=alloc[i][j];
for(i=0;i<m;i++) work[i]=avail[i];
for(j=0;j<m;j++) work[j]=total[j]-work[j];
for(i=0;i<n;i++)
for(j=0;j<m;j++)
 need[i][j]=max[i][j]-alloc[i][j];
A:
for(i=0;i<n;i++)
{
c=0;
for(j=0;j<m;j++)
if((need[i][j]<=work[j])&&(finish[i]=='n'))
c++;
if(c==m)
{
printf("All the resources can be allocated to Process %d", i+1);
printf("\n\nAvailable resources are:");
for(k=0;k<m;k++)
{
work[k]+=alloc[i][k];
printf("%4d",work[k]);
}
printf("\n");
finish[i]='y';
printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
```

```
count++;
}
}
if(count!=n)
goto A;
else
printf("\n System is in safe mode");
printf("\n The given state is safe state");
getch();
}
```

OUTPUT:

```
Enter the no. of processes and resources: 4 3 Enter the claim matrix:
3 2 2
6 1 3
3 1 4
4 2 2
Enter the allocation matrix:
1 0 0
6 1 2
2 1 1
0 0 2
Resource vector:9 3 6
All the resources can be allocated to Process 2 Available resources are: 6 2 3
Process 2 executed?:y
All the resources can be allocated to Process 3 Available resources are: 8 3 4
Process 3 executed?:y
All the resources can be allocated to Process 4 Available resources are: 8 3 6
Process 4 executed?:y
All the resources can be allocated to Process 1 Available resources are: 9 3 6
Process 1 executed?:y
System is in safe mode
The given state is safe state
```

Ex.No:9.	Simulate Bankers Algorithm for Dead Lock Detection

ALGORITHM:

Step-1: Start the program.
Step-2: Declare the memory for the process.
Step-3: Read the number of process, resources, allocation matrix and available matrix. Step-4: Compare each and every process using the banker‟s algorithm.
Step-5: If the process is in safe state then it is a not a deadlock process otherwise it is a deadlock process
Step-6: produce the result of state of process Step-7: Stop the program

PROGRAM:

```
#include<stdio.h>
 #include<conio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
int i,j;
printf("********** Deadlock Detection Algo **********\n"); input();
show();
cal();
getch();
 return 0;
}
void input()
{
int i,j;
```

```
printf("Enter the no of Processes\t");
 scanf("%d",&n);


printf("Enter the no of resource instances\t");
 scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{for(j=0;j<r;j++)
{
scanf("%d",&max[i][j]);
}}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{for(j=0;j<r;j++)
{
scanf("%d",&alloc[i][j]);
}}
printf("Enter the available Resources\n"); for(j=0;j<r;j++)
{
scanf("%d",&avail[j]);
}}
void show()
{
int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++)
{
printf("\nP%d\t ",i+1); for(j=0;j<r;j++)
{
printf("%d ",alloc[i][j]);
}
printf("\t");
for(j=0;j<r;j++)
{printf("%d ",max[i][j]);
}
printf("\t");
 if(i==0)
{
for(j=0;j<r;j++)
 printf("%d ",avail[j]);
}}}
void cal()
{ int finish[100],temp,need[100][100],flag=1,k,c1=0; int dead[100];
```

```
int safe[100]; int i,j;
```

```
for(i=0;i<n;i++)
{
finish[i]=0;
}
//find need matrix
for(i=0;i<n;i++)
{for(j=0;j<r;j++)
{
need[i][j]=max[i][j]-alloc[i][j];
}}
while(flag)
{
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0)&&(need[i][j]<=avail[j]))
{
c++;
if(c==r)
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}/
printf("\nP%d",i);
if(finish[i]==1)
{
i=n;
}}}}}}
j=0;
flag=0;
for(i=0;i<n;i++)
{
if(finish[i]==0)
{
```

```
dead[j]=i; j++;
flag=1;
}}
if(flag==1)
{
printf("\n\nSystem is in Deadlock and the Deadlock process are\n");
for(i=0;i<n;i++)
{
printf("P%d\t",dead[i]);
}}
else
{
printf("\nNo Deadlock Occur");
}}
```

Ex.No:10.	Simulate the following page replacement algorithms: a) FIFO b) LRU c) LFU

FIFO

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.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
int fr[3];
void main()
{
void display();
int i,j,page[12]={2,3,2,1,5,2,4,5,3,2,5,2};
int flag1=0,flag2=0,pf=0,frsize=3,top=0;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0; flag2=0;
for(i=0;i<3;i++)
{
if(fr[i]==page[j])
{
flag1=1;
flag2=1;
break;
}
```

```
}
if(flag1==0)
{
for(i=0;i<frsize;i++)
{
if(fr[i]==-1)
{
fr[i]=page[j];
 flag2=1;
break;
}
}
}
if(flag2==0)
{
fr[top]=page[j];
top++;
pf++;
if(top>=frsize)
 top=0;
}
display();
}

printf("Number of page faults : %d ",pf+frsize);
 getch();
}
void display()
{
int       i;
printf("\n"); for(i=0;i<3;i++)
 printf("%d\t",fr[i]);
}
```

OUTPUT:

2 -1 -1
2 3 -1
2 3 -1
2 3 1
5 3 1
5 2 1
5 2 4
5 2 4
3 2 4

3 2 4
3 5 4
3 5 2
Number of page faults: 9

LRU

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
 int fr[3];
void main()
{
void display();
int p[12]={2,3,2,1,5,2,4,5,3,2,5,2},i,j,fs[3];
int index,k,l,flag1=0,flag2=0,pf=0,frsize=3;
clrscr();
for(i=0;i<3;i++)
{
fr[i]=-1;
}
for(j=0;j<12;j++)
{
flag1=0,flag2=0; for(i=0;i<3;i++)
{
if(fr[i]==p[j])
{
flag1=1; flag2=1; break;
}
}
if(flag1==0)

{
for(i=0;i<3;i++)
{
if(fr[i]==-1)
{
fr[i]=p[j];        flag2=1; break;
}
}
}
if(flag2==0)
{
```

```
for(i=0;i<3;i++) fs[i]=0;
for(k=j-1,l=1;l<=frsize-1;l++,k--)
{
for(i=0;i<3;i++)
{
if(fr[i]==p[k]) fs[i]=1;
}}
for(i=0;i<3;i++)
{
if(fs[i]==0) index=i;
}
fr[index]=p[j]; pf++;
}
display();
}
printf("\n no of page faults :%d",pf+frsize); getch();
}
void display()
{
int i; printf("\n"); for(i=0;i<3;i++) printf("\t%d",fr[i]);
}
```

OUTPUT:

```
2 -1 -1
2 3 -1
2 3 -1
2 3 1
2 5 1
2 5 1
2 5 4
2 5 4
3 5 4
3 5 2
3 5 2
3 5 2
```

No of page faults: 7

LFU

SOURCE CODE:

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

```
int f,p;
int pages[50],frame[10],hit=0,count[50],time[50];
 int i,j,page,flag,least,minTime,temp;
printf("Enter no of frames : ");
scanf("%d",&f);
printf("Enter no of pages : ");
scanf("%d",&p);
for(i=0;i<f;i++)
{
frame[i]=-1;
}
for(i=0;i<50;i++)
{
count[i]=0;
}
printf("Enter page no : \n");
for(i=0;i<p;i++)
{
scanf("%d",&pages[i]);
}
printf("\n"); for(i=0;i<p;i++)
{
count[pages[i]]++;
time[pages[i]]=i;
 flag=1;
 least=frame[0];
for(j=0;j<f;j++)
{
if(frame[j]==-1 || frame[j]==pages[i])
{
if(frame[j]!=-1)
{
hit++;
}
flag=0;
frame[j]=pages[i];
break;
}
if(count[least]>count[frame[j]])
{
least=frame[j];
}
}
if(flag)
```

```
{
minTime=50; for(j=0;j<f;j++)
{
if(count[frame[j]]==count[least] && time[frame[j]]<minTime)
{
temp=j;
minTime=time[frame[j]];
}
}
count[frame[temp]]=0;
 frame[temp]=pages[i];
}
for(j=0;j<f;j++)
{
printf("%d ",frame[j]);
}
printf("\n");
}
printf("Page hit = %d",hit);
return 0;
}
```

OUTPUT:

Ex.No:11.	Simulate the following File allocation strategies (a) Sequenced (b) Indexed (c) Linked

SEQUENTIAL:

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.

SOURCE CODE:

```
#include<stdio.h>
 main()
{
int f[50],i,st,j,len,c,k;
clrscr();
for(i=0;i<50;i++) f[i]=0;
X:
printf("\n Enter the starting block & length of file");
scanf("%d%d",&st,&len);
for(j=st;j<(st+len);j++) if(f[j]==0)
{
f[j]=1
;
printf("\n%d->%d",j,f[j]);
}
else
{
printf("Block already allocated"); break;
}
if(j==(st+len))
printf("\n the file is allocated to disk");
printf("\n if u want to enter more files?(y-1/n-0)"); scanf("%d",&c);
if(c==1)
goto X;
else
```

```
 exit();
getch();
}
```

OUTPUT:

```
Enter the starting block & length of file 4 10 4->1
5->1
6->1
7->1
8->1
9->1
10->1
11->1
12->1
13->1
The file is allocated to disk.
```

INDEXED:

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.

SOURCE CODE:

```
#include<stdio.h>
int       f[50],i,k,j,inde[50],n,c,count=0,p;
main()
{
clrscr();
for(i=0;i<50;i++) f[i]=0;
x: printf("enter index block\t");
scanf("%d",&p);
if(f[p]==0)
{ f[p]=1;
printf("enter no of files on index\t");
scanf("%d",&n);
}
else
{
printf("Block already allocated\n");
```

```
goto x;
}
for(i=0;i<n;i++)
scanf("%d",&inde[i]);
for(i=0;i<n;i++)
if(f[inde[i]]==1)
{
printf("Block already allocated");
goto x;
}
for(j=0;j<n;j++)
f[inde[j]]=1;
printf("\n allocated");
printf("\n file indexed");
for(k=0;k<n;k++)
printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);
printf(" Enter 1 to enter more files and 0 to exit\t");
scanf("%d",&c);
if(c==1)
 goto    x;
else exit();
getch();
}
```

OUTPUT:

```
enter index block 9
Enter no of files on index 3 1 2 3
Allocated File indexed 9->1:1
9->2;1
9->3:1
enter 1 to enter more files and 0 to exit
```

LINKED

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

SOURCE CODE:

```
#include<stdio.h>
 main()
```

```
{
int f[50],p,i,j,k,a,st,len,n,c;
clrscr();
for(i=0;i<50;i++)
 f[i]=0;
printf("Enter how many blocks that are already allocated");
scanf("%d",&p);
printf("\nEnter the blocks no.s that are already allocated");
 for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
X:
printf("Enter the startingindex block & length");
scanf("%d%d",&st,&len);
k=len;
for(j=st;j<(k+st);j++)
{
if(f[j]==0)
{ f[j]=1;
printf("\n%d->%d",j,f[j]);
}
else
{
printf("\n %d->file is already allocated",j);
k++;
}
}
printf("\n If u want to enter one more file? (yes-1/no-0)");
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
getch( );
}
```

OUTPUT:

Enter how many blocks that are already allocated 3 Enter the blocks no.s that are already allocated 4 7 Enter the starting index block & length 3 7 9
3->1
4->1 file is already allocated 5->1
6->1

7->1 file is already allocated 8->1
9->1file is already allocated 10->1
11->1
12->1

Ex.No:12.	Write a C program that illustrates two processes communicating using shared memory.

DESCRIPTION

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and 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. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

SOURCE CODE WRITER PROCESS:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
//creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT is
used to create the shared segment if it does not exist. 0666 are the permissions on the shar
ed segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
//process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
//this prints the address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
```

```
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

OUTPUT:

```
Key of shared memory is 0
Process attached at 0x7ffe040fb000
Enter some data to write to shared memory
Hello World
You wrote: Hello World
```

SOURCE CODE READER PROCESS:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

OUTPUT:

```
Key of shared memory is 0
Process attached at 0x7f76b4292000
Data read from shared memory is: Hello World
```

Ex.No:13.	Write a C program to simulate producer and consumer problem using semaphores

DESCRIPTION

Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and 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. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

SOURCE CODE:

```
#include<stdio.h>
int mutex=1,full=0,empty=3,x=0;
 main()
{
int n;
void producer();
void consumer();
 int wait(int);
int signal(int);
printf("\n1.PRODUCER\t2.CONSUMER\t3.EXIT\n");
while(1) {
printf("\nENTER YOUR CHOICE\n"); scanf("%d",&n);
switch(n)
{
case 1:
if((mutex==1)&&(empty!=0)) producer();
else
printf("BUFFER IS FULL");
break;
case 2: if((mutex==1)&&(full!=0)) consumer();
else
printf("BUFFER IS EMPTY");
break;
```

```
case 3:exit(0);
break;
}
}
}
int wait(int s)
{
return(--s);
}
int signal(int s)
{
 return(++s);
}
void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\nProducer produces the item%d",x);
mutex=signal(mutex); }
void consumer()
{
 mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\n Consumer consumes item%d",x);
 x--;
mutex=signal(mutex);
}
```

OUTPUT:

```
1. PRODUCER        2. CONSUMER        3.EXIT
Enter your choice: 2
Buffer is Empty
1. PRODUCER        2. CONSUMER        3.EXIT
 Enter your choice: 1
Producer produces the item 1
1. PRODUCER        2. CONSUMER        3.EXIT
Enter your choice: 2
Consumer consumes item 1
1. PRODUCER        2. CONSUMER        3.EXIT
 Enter your choice: 3
```

Ex.No:14.	Write C program to create a thread using pthreads library and let it run its function.

DESCRIPTION

Here two threads of execution are created in the code. The order of the lines of output of the two threads may be interchanged depending upon the thread processed earlier. The main thread waits on the newly created thread for exiting. Therefore, the final line of the output is printed only after the new thread exits. The threads can terminate independently of each other by not using the pthread_join function. If we want to terminate the new thread manually, we may use pthread_cancel to do it.

Note: If we use exit() instead of **pthread_exit()** to end a thread, the whole process with all associated threads will be terminated even if some of the threads may still be running.

SOURCE CODE:

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

void* func(void* arg)
{
   // detach the current thread
   // from the calling thread
   pthread_detach(pthread_self());

   printf("Inside the thread\n");

   // exit the current thread
   pthread_exit(NULL);
}

void fun()
{
   pthread_t ptid;

   // Creating a new thread
   pthread_create(&ptid, NULL, &func, NULL);
```

```
    printf("This line may be printed"
           " before thread terminates\n");

    // The following line terminates
    // the thread manually
    // pthread_cancel(ptid);

    // Compare the two threads created
    if(pthread_equal(ptid, pthread_self())
       printf("Threads are equal\n");
    else
       printf("Threads are not equal\n");

    // Waiting for the created thread to terminate
    pthread_join(ptid, NULL);

    printf("This line will be printed"
           " after thread ends\n");

    pthread_exit(NULL);
}

// Driver code
int main()
{
    fun();
    return 0;
}
```

OUTPUT:

```
This line may be printed before thread terminates
Threads are not equal
Inside the thread
This line will be printed after thread ends
```

Ex.No:15.	**Write a C program to illustrate concurrent execution of threads using pthreads library.**

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

void *mythread1(void *vargp)
{
   int i;
   printf("thread1\n");
      for(i=1;i<=10;i++)
     printf("i=%d\n",i);
   printf("exit from thread1\n");
  return NULL;
}

void *mythread2(void *vargp)
{
    int j;
   printf("thread2 \n");
   for(j=1;j<=10;j++)
    printf("j=%d\n",j);

 printf("Exit from thread2\n");
  return NULL;
}


int main()
{
  pthread_t tid;
  printf("before thread\n");
  pthread_create(&tid,NULL,mythread1,NULL);
  pthread_create(&tid,NULL,mythread2,NULL);
```

```
  pthread_join(tid,NULL);
  pthread_join(tid,NULL);
  exit(0);
}
```

OUT PUT:

```
$ cc filename.c – l pthread

$./a.out
thread1
i=1
i=2;
i=3
thread2
j=1
j=2
j=3
j=4
j=5
j=6
j=7
j=8
i=4
i=5
i=6
i=7
i=8
i=9
i=10
exit from thread1
j=9
j=10
exit from thread2
```