List of Experiments

1. Basics of UNIX commands.
2. Shell Programming.
3. Implement the following CPU scheduling algorithms
   a) Round Robin b) SJF c) FCFS d) Priority
4. Implement all file allocation strategies
   a) Sequential b) Indexed c) Linked
5. Implement Semaphores
6. Implement all File Organization Techniques
   a) Single level directory b) Two level c) Hierarchical d) DAG
7. Implement Bankers Algorithm for Dead Lock Avoidance
8. Implement an Algorithm for Dead Lock Detection
9. Implement e all page replacement algorithms
   a) FIFO b) LRU c) LFU
10. Implement Shared memory and IPC
11. Implement Paging Technique of memory management.
12. Implement Threading & Synchronization Applications

List of Innovative Experiments

1. Creating directory structures using Linux Commands.
2. Performing Database oriented applications in Linux Programming.
3. Generating new Operating system with the help of Existing Operating System concepts.

EX.NO:1 **BASICS OF UNIX COMMANDS**

AIM:

To Study the basic commands in UNIX.

COMMANDS:

- cat --- for creating and displaying short files
- chmod --- change permissions
- cd --- change directory
- cp --- for copying files
- date --- display date
- echo --- echo argument
- ftp --- connect to a remote machine to download or upload files
- grep --- search file
- head --- display first part of file
- ls --- see what files you have
- lpr --- standard print command
- more --- use to read files
- mkdir --- create directory
- mv --- for moving and renaming files
- ncftp --- especially good for downloading files via anonymous ftp.
- print --- custom print command (see also lpr)
- pwd --- find out what directory you are in
- rm --- remove a file
- rmdir --- remove directory
- rsh --- remote shell
- setenv --- set an environment variable
- sort --- sort file
- tail --- display last part of file
- tar --- create an archive, add or extract files
- telnet --- log in to another machine
- wc --- count characters, words, lines

cat

This is one of the most flexible Unix commands. We can use to create, view and concatenate files. For our first example we create a three-item English-Spanish dictionary in a file called "dict."

cat >dict
red rojo
green verde
blue azul

If we wish to add text to an existing file we do this:

cat >>dict
white blanco
black negro

Now suppose that we have another file tmp that looks like this:

```
% cat tmp
 cat gato
 dog perro
%
```

Then we can join dict and tmp like this:

```
% cat dict tmp >dict2
```

We could check the number of lines in the new file like this:

```
% wc -l dict2
8
```

The command wc counts things --- the number of characters, words, and line in a file.

---

chmod

This command is used to change the permissions of a file or directory. For example to make a file essay.001 readable by everyone, we do this:

```
% chmod a+r essay.001
```

To make a file, e.g., a shell script mycommand executable, we do this

```
% chmod +x mycommand
```

Now we can run mycommand as a command.

To check the permissions of a file, use **ls -l** . For more information on chmod, use **man chmod**.

---

cd

Use **cd** to change directory. Use pwd to see what directory you are in.

```
% cd english
% pwd
% /u/ma/jeremy/english
% ls
novel poems
% cd novel
% pwd
% /u/ma/jeremy/english/novel
% ls
ch1 ch2 ch3 journal scrapbook
% cd ..
% pwd
% /u/ma/jeremy/english
% cd poems
% cd
% /u/ma/jeremy
```

---

cp

Use **cp** to copy files or directories.

```
% cp foo foo.2
```

This makes a copy of the file foo.

```
% cp ~/poems/jabber .
```

This copies the file jabber in the directory poems to the current directory. The symbol "." stands for the current directory. The symbol "~" stands for the home directory.

---

date

Use this command to check the date and time.

```
% date
Fri Jan  6 08:52:42 MST 1995
```

---

echo

The echo command echoes its arguments. Here are some examples:

```
% echo this
  this
% echo $EDITOR
  /usr/local/bin/emacs
% echo $PRINTER
  b129lab1
```

Things like PRINTER are so-called *environment variables*. This one stores the name of the default printer --- the one that print jobs will go to unless you take some action to change things. The dollar sign before an environment variable is needed to get the value in the variable. Try the following to verify this:

```
% echo PRINTER
  PRINTER
```

---

ftp

Use ftp to connect to a remote machine, then upload or download files. See also: ncftp

We'll connect to the machine fubar.net, then change director to mystuff, then download the file homework11:

```
% ftp solitude
  Connected to fubar.net.
  220 fubar.net FTP server (Version wu-2.4(11) Mon Apr 18 17:26:33 MDT 1994) ready.
Name (solitude:carlson): jeremy
  331 Password required for jeremy.
Password:
  230 User jeremy logged in.
ftp> cd mystuff
  250 CWD command successful.
ftp> get homework11
ftp> quit
```

---

grep

Use this command to search for information in a file or files. For example, suppose that we have a file dict whose contents are

```
red rojo
green verde
blue azul
white blanco
```

black negro

Then we can look up items in our file like this;

```
% grep red dict
 red rojo
% grep blanco dict
 white blanco
% grep brown dict
%
```

Notice that no output was returned by grep brown. This is because "brown" is not in our dictionary file.

Grep can also be combined with other commands. For example, if one had a file of phone numbers named "ph", one entry per line, then the following command would give an alphabetical list of all persons whose name contains the string "Fred".

```
% grep Fred ph | sort
  Alpha, Fred: 333-6565
  Beta, Freddie: 656-0099
  Frederickson, Molly: 444-0981
  Gamma, Fred-George: 111-7676
  Zeta, Frederick: 431-0987
```

The symbol "|" is called "pipe." It pipes the output of the grep command into the input of the sort command.

For more information on grep, consult

```
% man grep
```

---

head

Use this command to look at the head of a file. For example,

```
% head essay.001
```

displays the first 10 lines of the file essay.001 To see a specific number of lines, do this:

```
% head -n 20 essay.001
```

This displays the first 20 lines of the file.

---

ls

Use **ls** to see what files you have. Your files are kept in something called a directory.

```
% ls
 foo       letter2
 foobar    letter3
 letter1   maple-assignment1
%
% ls l*
 letter1 letter2 letter3
%
```

---

lpr

This is the standard Unix command for printing a file. It stands for the ancient "line printer." See

```
% man lpr
```

for information on how it works. See print for information on our local intelligent print command.

---

mkdir
Use this command to create a directory.
% **mkdir essays**
To get "into" this directory, do
% **cd essays**
To see what files are in essays, do this:
% **ls**
There shouldn't be any files there yet, since you just made it. To create files, see cat or emacs.

---

more
More is a command used to read text files. For example, we could do this:
% more poems
The effect of this to let you read the file "poems ". It probably will not fit in one screen, so you need to know how to "turn pages". Here are the basic commands:

- **q** --- quit more
- **spacebar** --- read next page
- **return key** --- read next line
- **b** --- go back one page

For still more information, use the command **man more**.

---

mv
Use this command to change the name of file and directories.
% **mv foo foobar**
The file that was named foo is now named foobar

---

print
This is a moderately intelligent print command.
% **print foo**
% **print notes.ps**
% **print manuscript.dvi**
In each case print does the right thing, regardless of whether the file is a text file (like foo), a postcript file (like notes.ps, or a dvi file (like manuscript.dvi. In these examples the file is printed on the default printer. To see what this is, do
% **print**
and read the message displayed. To print on a specific printer, do this:
% **print foo jwb321**
% **print notes.ps jwb321**
% **print manuscript.dvi jwb321**
To change the default printer, do this:
% **setenv PRINTER jwb321**

---

rm
Use **rm** to remove files from your directory.

```
% rm foo
  remove foo? y
% rm letter*
  remove letter1? y
  remove letter2? y
  remove letter3? n
%
```

The first command removed a single file. The second command was intended to remove all files beginning with the string "letter." However, our user (Jeremy?) decided not to remove letter3.

---

rmdir
Use this command to remove a directory. For example, to remove a directory called "essays", do this:

```
% rmdir essays
```

A directory must be empty before it can be removed. To empty a directory, use rm.

---

rsh
Use this command if you want to work on a computer different from the one you are currently working on. One reason to do this is that the remote machine might be faster. For example, the command

```
% rsh solitude
```

connects you to the machine solitude. This is one of our public workstations and is fairly fast.
See also: telnet

---

sort
Use this commmand to sort a file. For example, suppose we have a file dict with contents

```
red rojo
green verde
blue azul
white blanco
black negro
```

Then we can do this:

```
% sort dict
  black negro
  blue azul
  green verde
  red rojo
  white blanco
```

Here the output of sort went to the screen. To store the output in file we do this:

```
% sort dict >dict.sorted
```

You can check the contents of the file dict.sorted using cat , more , or emacs .

---

tail
Use this command to look at the tail of a file. For example,

% **tail essay.001**

displays the last 10 lines of the file essay.001 To see a specific number of lines, do this:

% **tail -n 20 essay.001**

This displays the last 20 lines of the file.

---

tar

Use create compressed archives of directories and files, and also to extract directories and files from an archive. Example:

% **tar -tvzf foo.tar.gz**

displays the file names in the compressed archive foo.tar.gz while

% **tar -xvzf foo.tar.gz**

extracts the files.

---

telnet

Use this command to log in to another machine from the machine you are currently working on. For example, to log in to the machine "solitude", do this:

% **telnet solitude**

See also: rsh.

---

wc

Use this command to count the number of characters, words, and lines in a file. Suppose, for example, that we have a file dict with contents

red rojo
green verde
blue azul
white blanco
black negro

Then we can do this

% **wc dict**

5 10 56
tmp

This shows that dict has 5 lines, 10 words, and 56 characters.

The word count command has several options, as illustrated below:

% **wc -l dict**

5 tmp

% **wc -w dict**

10 tmp

% **wc -c dict**

56 tmp

RESULT:

Thus the basic UNIX Commands are studied and verified using various samples.

SHELL PROGRAMMING

Ex.No.2a **EVEN OR ODD**

AIM:

To write a program to find whether a number is even or odd .

ALGORITHM:

STEP 1: Read the input number.
STEP 2: Perform modular division on input number by 2.
STEP 3: If remainder is 0 print the number is even.
STEP 4: Else print number is odd.
STEP 5: Stop the program.

PROGRAM

```
//evenodd.sh
echo "enter the number"
read num
if [ ` expr  $num  %  2` -eq 0 ]
then
echo "number is even"
else
echo "number is odd"
fi
```

OUTPUT:

```
[students@localhost ~]$  sh evenodd.sh
enter the number: 5 the number is odd.
```

RESULT:

Thus the Shell program for finding the given number ODD or EVEN was implemented and output verified using various samples.

Ex.No.2b **BIGGEST OF THREE NUMBERS**

AIM:

To write a Shell Program to Find Biggest In Three Numbers.

ALGORITHM:

STEP 1: Read The Three Numbers.
STEP 2: If A Is Greater Than B And A Is Greater Than C Then Print A Is Big.
STEP 3: Else If B is greater Than C Then C Is Big.
STEP 4: Else Print C Is Big.
STEP 5: Stop The Program.

PROGRAM:

```
// big3.sh
echo "enter three numbers"
read a b c
if [ $a -gt $b ] && [ $a -gt $c ]
then
echo "A is big"
else if [ $b -gt $c ]
then
echo "B is big"
else
echo "C is big"
fi
```

OUTPUT:

```
[students@localhost ~]$  sh big3.sh
        ENTER THREE NUMBERS:
        23 54 78
        C IS BIG.
```

RESULT:

Thus the Shell program for finding biggest of three numbers was implemented and output verified using various samples.

Ex.No.2c **FACTORIAL OF NUMBER**

AIM:

To find a factorial of a number using Shell programming.

ALGORITHM:

Step 1: read a number.
Step 2: Initialize fact as 1.
Step 3: Initialize I as 1.
Step 4: While I is lesser than or equal to no.
Step 5: Multiply the value of I and fact and assign to fact increment the value of I by 1.
Step 6: print the result.
Step 7: Stop the program.

PROGRAM:

```
// fact.sh
echo "enter the number"
read n
fact=1
i=1
while [ $i -le $n ]
do
fact=`expr $i \* $fact`
i=`expr $i + 1`
done
echo "the fcatorial number of $ni is $fact
```

OUTPUT:

```
[students@localhost ~]$  sh fact.sh
Enter the number :4
The factorial of 4 is 24.
```

RESULT:

Thus the Shell program for factorial operation was implemented and output verified using various samples

Ex.No.2d **FIBONACCI SERIES**

AIM :

To write a Shell program to display the Fibonacci series.

ALGORITHM:

Step 1: Initialize n1& n2 as 0 & 1.
Step 2: enter the limit for Fibonacci.
Step 3: initialize variable as 0
Step 4: Print the Fibonacci series n1 and n2.
Step 5: While the var number is lesser than lim-2
Step 6: Calculate n3=n1+n2.
Step 7: Set n1=n2 and n2=n3
Step 8: Increment var by 1 and print n2
Step 9: stop the program.

PROGRAM:

```
// fib.sh
echo " ENTER THE LIMIT FOR FIBONNACI SERIES"
read lim
n1=0
n2=1
var=0
echo "FIBONACCI SERIES IS "
echo "$n1"
echo "$n2"
while [ $var -lt `expr $lim - 2` ]
do
n3=`expr $n1 + $n2 `
n1=`expr $n2 `
n2=`expr $n3 `
var=`expr $var + 1 `
echo "$n2"
done
```

OUTPUT :

```
[students@localhost ~]$  sh fib.sh
enter the limit for Fibonacci: 5
The Fibonacci series is:0,1,1,2,3
```

RESULT:

Thus the Shell program for generating Fibonacci series was implemented and output Verified using various samples

BEYOND THE SYLLABUS – Other SHELL Programs

AIM:

1. MULTIPLICATION TABLE

Write a shell script to print the for the given number.

Algorithm:

1.Start.
2.Read n.
3.i=1.
4.f = n * i.
5.i = i + 1.
6.Display f.
7.Repeat 4,5,6 steps until i = 10.
8.End.

Source Code:

```
#! /bin/sh
echo “enter the number”
read n
for i in 1 2 3 4 5 6 7 8 9 10
do
x= `expr $n \* $i`
done
```

2. COPIES MULTIPLE FILES INTO A DIRECTORY

AIM:

Write a shell script that copies multiple files into a directory.

Algorithm:

1.Start.
2.Read *.c files into i.
3.Copy $i to the root directory.
4.Repeat 2,3 steps until all values of $i..
5.End.

Source Code:

```
#! /bin/sh
for i in `ls *.c`
do
cp $i /root
done
echo “file are copied into root”
```

3. NO. OF WORDS AND CHARACTERS IN A GIVEN FILE

AIM:

Write a shell script to find no. of words and characters in a given file.

Algorithm:

1. Start.
2. cc = 0,w=0.
3. give file name as commandline argument (i.e $1).
4. for i in `cat $1`.
5. w = w + 1.
6. c = expr "$i" : "*".
7. cc = cc + c.
8. Repeat 4,5,6,7 steps until eof.
9. End.

Source Code:

```
# /bin/sh
w=0
cc=0
for i in 'cat $1'
do
j= $i
echo $j
w=`expr $w + 1`
c=`expr "$j":".*"`
cc = `expr $cc + $c`
done
echo "no.of characters " $cc
echo "no.of words" $w
```

4. TO DISPLAY ALL FILES IN A GIVEN DIRECTORY

AIM:

Write a shell script to display all files in a given directory.

Algorithm:

1.Start.
2.Read i value from ls sss.
3.Display i value .
4.Repeat above 2 steps until end of directory.
5.End.

Source Code:

```
#! /bin/sh
for i in `ls sss`
do
echo $i
done
```

5. CALCULATOR

AIM:

Write a shell script to perform simple calculator.

Algorithm:

1.Start
2.Read a and b
3.Read op
4.Depending on value of operator perform case
Operation.
5.End

Source Code:

```
#! /bin/sh
echo ‘enter the value for a’
read a
echo ‘enter the value for b’
read b
echo ‘enter operator’
read op
case $op in
+) c=`expr $a + $b`;;
-) c = `expr $a -$b`;;
\*) c = `expr $a \* $b`;;
/) c = `expr $a / $b ;;
esac
echo $c
```

Viva Questions

1. What is an operating system?

An operating system is a program that controls the execution of application programs and acts as an interface between the user of the computer and the computer hardware. It creates a user friendly environment. We can view an operating system as a resource allocator.

2. What are the two primary goals for operating system?

i.Convenience for the user ii.Efficient operation of the computer system.

3. Define resource allocator?

OS as a resource allocator keeps track of the status of each resources and decides how to allocate them to specific programs and users. so that it can operate the computer system efficiently and fairly.

4. Define graceful degradation and fault tolerance?

The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems designed for graceful degradation are also called as fault tolerance.

5. Differentiate symmetric and asymmetric multi processing?

SYMMETRIC ASYMMETRIC

In symmetric multiprocessing (SMP), each processor runs an identical copy of operating system. In asymmetric, each processor is assigned to a specified task. In SMP, no master-slave

relationship exists between processors. Master-slave exists. Master processor schedules and allocates work to the slave processor.

6. Define loosely coupled system (or) distributed system?

The computer networks used in the applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines such as high speed bus or telephone lines, these systems are usually referred as loosely coupled system or distributed system.

Implement the following CPU scheduling algorithms

Ex.No:3a FCFS (FIRST COME FIRST SERVE) CPU SCHEDULING

AIM:

To write a C - Program to implement the FCFS (First Come First Serve) CPU scheduling Algorithm.

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Calculation of Turn Around Time and Waiting Time

a) tot_TAT = tot_TAT + pre_TAT
b) avg_TAT = tot_TAT/num_of_proc
c) tot_WT = tot_WT + pre_WT + PRE_BT
d) avg_WT = tot_WT/num_of_proc

5. Display the result
6. STOP the program

PROGRAM: (FCFS Scheduling)

```
//fcfs.c
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,wt[30],n,tot_wt=0,tat[30],FCFS_wt=0,FCFS_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
int i;
clrscr();
printf("\nEnter the no.of processes \n");
scanf("%d",&n);
printf("Enter burst time for each process\n");
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
p[i] = i;
}
printf("\n FCFS Algorithm \n");
WT_TAT(&FCFS_tat,&FCFS_wt);
printf("\n\nTotal Turn around Time:%d",FCFS_tat);
```

```
printf("\nAverage Turn around Time :%d ", FCFS_tat/n);
printf("\nTotal Waiting Time:%d",FCFS_wt);
printf("\nTotal avg. Waiting Time:%d",FCFS_wt/n);
getch();
}
int WT_TAT(int *a, int *b)
{
int i;
for(i=0;i<n;i++)
{
if(i==0)
tat[i] = bt[i];
else
tat[i] = tat[i-1] + bt[i];
tot_tat=tot_tat+tat[i];
}
*a = tot_tat;
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=wt[i-1]+bt[i-1];
tot_wt = tot_wt+wt[i];
}
*b = tot_wt;
printf("\nPROCESS\t\tBURST TIME\tTURN AROUND TIME\tWAITING TIME");
for(i=0; i<n; i++)
printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d",p[i],bt[i],tat[i],wt[i]);
return 0;
}
```

OUTPUT: (FCFS Scheduling Algorithm)

```
[students@localhost ~]$  cc fcfs.c
[students@localhost ~]$ ./a.out
```

RESULT:

Thus the C - program for First Come First Serve (FCFS) CPU Scheduling algorithm was implemented and output verified using various samples.

Ex.No:3b SJF (SHORTEST JOB FIRST) CPU SCHEDULING ALGORITHM

AIM:

To write a C - Program to implement the SJF (Shortest Job First) CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Sort the processors based on the burst time

5. Calculation of Turn Around Time and Waiting Time
e) tot_TAT = tot_TAT + pre_TAT
f) avg_TAT = tot_TAT/num_of_proc
g) tot_WT = tot_WT + pre_WT + PRE_BT
h) avg_WT = tot_WT/num_of_proc
6. Display the result
7. STOP the program

PROGRAM: (SJF Scheduling)

```
//sjf.c
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,wt[30],n,tot_wt=0,tat[30],SJF_wt=0,SJF_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
int i;
clrscr();
printf("\nEnter the no.of processes \n");
scanf("%d",&n);
printf("Enter burst time for each process\n");
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
p[i] = i;
} sort();
WT_TAT(&SJF_tat,&SJF_wt);
printf("\n\nTotal Turn around Time:%d",SJF_tat);
printf("\nAverage Turn around Time :%d ", SJF_tat/n);
printf("\nTotal Waiting Time:%d",SJF_wt);
printf("\nTotal avg. Waiting Time:%d",SJF_wt/n);
getch();
}
int sort()
{
int t,i,j;
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(bt[i]>bt[j])
{
swap(&bt[j],&bt[i]);
swap(&p[j],&p[i]);
}
}
```

```
}
return 0;
}
int swap(int *a, int *b)
{
int t;
t = *a;
*a = *b;
*b = t;
return 0;
}
int WT_TAT(int *a, int *b)
{
int i;
for(i=0;i<n;i++)
{
if(i==0)
tat[i] = bt[i];
else
tat[i] = tat[i-1] + bt[i];
tot_tat=tot_tat+tat[i];
}
*a = tot_tat;
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=wt[i-1]+bt[i-1];
tot_wt = tot_wt+wt[i];
}
*b = tot_wt;
printf("\nPROCESS\t\tBURST TIME\tTURN AROUND TIME\tWAITING TIME");
for(i=0; i<n; i++)
printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d",p[i]+1,bt[i],tat[i],wt[i]);
return 0;
}
```

OUTPUT: (SJF Scheduling Algorithm)

```
[students@localhost ~]$  cc sjf.c
[students@localhost ~]$ ./a.out
```

RESULT:

Thus the C - program for Shortest Job First (SJF) CPU Scheduling algorithm was implemented and output verified using various samples.

Ex.No:3c **PRIORITY CPU SCHEDULING ALGORITHM**

AIM:

To write a C - Program to implement the Priority CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time of each processors
4. Get the priority of all the processors
5. Sort the processors based on the priority
6. Calculation of Turn Around Time and Waiting Time
    i) tot_TAT = tot_TAT + pre_TAT
    j) avg_TAT = tot_TAT/num_of_proc
    k) tot_WT = tot_WT + pre_WT + PRE_BT
    l) avg_WT = tot_WT/num_of_proc
7. Display the result
8. STOP the program

PROGRAM: (Priority Scheduling)

```
//priority.c
#include<stdio.h>
#include<conio.h>
int p[30],bt[30],tot_tat=0,pr[30],wt[30],n,tot_wt=0,tat[30],PR_wt=0,PR_tat=0;
float awt,avg_tat,avg_wt;
void main()
{
int i;
clrscr();
printf("\nEnter the no.of processes \n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter burst time and priority of process[%d]:",i+1);
scanf("%d%d",&bt[i],&pr[i]);
p[i] = i;
} sort();
WT_TAT(&PR_tat,&PR_wt);
printf("\n\nTotal Turn around Time:%d",PR_tat);
printf("\nAverage Turn around Time :%d ", PR_tat/n);
printf("\nTotal Waiting Time:%d",PR_wt);
printf("\nTotal avg. Waiting Time:%d",PR_wt/n);
getch();
}
int sort()
{
int t,i,j,t2,t1;
for(i=0;i<n;i++)
```

```
{
for(j=i+1;j<n;j++)
{
if(pr[i]>pr[j])
{
swap(&bt[j],&bt[i]);
swap(&p[j],&p[i]);
swap(&pr[j],&pr[i]);
}
}
}
return 0;
}
int swap(int *a, int *b)
{
int t;
t = *a;
*a = *b;
*b = t;
return 0;
}
int WT_TAT(int *a, int *b)
{
int i;
for(i=0;i<n;i++)
{
if(i==0)
tat[i] = bt[i];
else
tat[i] = tat[i-1] + bt[i];
tot_tat=tot_tat+tat[i];
}
*a = tot_tat;
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=wt[i-1]+bt[i-1];
tot_wt = tot_wt+wt[i];
}
*b = tot_wt;
printf("\nPROCESS\t\tBURST TIME\tPRIORITY\tTURN AROUND TIME\tWAITING
TIME");
for(i=0; i<n; i++)
printf("\nprocess[%d]\t\t%d\t\t%d\t\t%d\t\t%d"
,p[i]+1,bt[i],pr[i],tat[i],wt[i]);
return 0; }
```

OUTPUT: (Priority Scheduling Algorithm)

```
[students@localhost ~]$  cc priority.c
[students@localhost ~]$ ./a.out
```

RESULT:

Thus the C - program for Priority CPU Scheduling algorithm was implemented and output verified using various samples.

Ex.No:3d ROUND ROBIN CPU SCHEDULING ALGORITHM

AIM:

To write a C - Program to implement the Round Robin CPU scheduling Algorithm

ALGORITHM:

1. START the program
2. Get the number of processors
3. Get the Burst time(BT) of each processors
4. Get the Quantum time(QT)
5. Execute each processor until reach the QT or BT
6. Time of reaching processor's BT is it's Turn Around Time(TAT)
7. Time waits to start the execution, is the waiting time(WT) of each processor
8. Calculation of Turn Around Time and Waiting Time
    m) tot_TAT = tot_TAT + cur_TAT
    n) avg_TAT = tot_TAT/num_of_proc
    o) tot_WT = tot_WT + cur_WT
    p) avg_WT = tot_WT/num_of_proc
9. Display the result
10. STOP the program

PROGRAM: (Round Robin Algorithm)

```
//rr.c
#include<stdio.h>
#include<conio.h>
int TRUE = 0;
int FALSE = -1;
int tbt[30],bt[30],tat[30],n=0,wt[30],qt=0,tqt=0,time=0,lmore,t_tat=0,t_wt=0;
void main()
{
int i,j;
clrscr();
printf("\nEnter no. of processors:");
scanf("%d",&n);
printf("\nEnter Quantum Time:");
scanf("%d",&qt);
for(i=0;i<n;i++)
{
```

```
printf("\nEnter Burst Time of Processor[%d]:",i+1);
scanf("%d",&bt[i]);
tbt[i] = bt[i];
wt[i] = tat[i] = 0;
}
lmore = TRUE;
while(lmore == TRUE)
{
lmore = FALSE;
for(i=0;i<n;i++)
{
if(bt[i] != 0)
wt[i] = wt[i] + (time - tat[i]);
tqt = 1;
while(tqt <= qt && bt[i] !=0)
{
lmore = TRUE;
bt[i] = bt[i] -1;
tqt++;
time++;
tat[i] = time;
}
}
}
printf("\nProcessor ID\tBurstTime\tTurnAroundTime\tWaitingTime\n");
for(i=0;i<n;i++)
{
printf("Processor%d\t\t%d\t\t%d\t\t%d\n",i+1,tbt[i],tat[i],wt[i]);
t_tat = t_tat + tat[i];
t_wt = t_wt + wt[i];
}
printf("\nTotal Turn Around Time:%d",t_tat);
printf("\nAverage Turn Around Time:%d",t_tat/n);
printf("\nTotal Waiting Time:%d",t_wt);
printf("\nAverage Waiting Time:%d",t_wt/n);
getch();
}
```

OUTPUT: (Round Robin Scheduling Algorithm)

```
[students@localhost ~]$  cc rr.c
[students@localhost ~]$ ./a.out
```

RESULT:

Thus the C - program for Round Robin CPU Scheduling algorithm was implemented and output verified using various samples.

Viva Questions

1. Define long term and short term scheduler.

Long term scheduler/Job scheduler.

It determines which programs are admitted to the system for processing. It selects from the queue and loads the processes into memory for the CPU scheduler. When process changes the state from new to ready then there be long term scheduler.

Short term scheduler/CPU scheduler.

It is the change of ready state to running state of the process also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next.

2. What is context switch?

When the scheduler switches the CPU from executing one process to executing another, the context switcher saves the content of all processor registers for the process being removed from the CPU in its process descriptor.

3) Differentiate between preemptive and non-preemptive?

Preemptive Scheduling:

1. When process switches from running State to ready state.
2. When a process switches from waiting state to ready state.

Non preemptive scheduling:

Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by a process switches from running state to waiting state or the process terminates.

4) Define dispatcher and dispatch latency.

The dispatcher is the module that gives control of the CPU to the process selected by short term scheduler. The function involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program.

Dispatch latency: Time taken by the dispatcher to stop one process and start another process for running is known as the dispatch latency.

5) Define throughput and response time?

Throughput refers to the no of process completed per time unit. Higher the throughput higher amount of work can be done by the CPU. Response time refers to the measure the amount of time taken from submission of the request till the first response is found.

6) Define turnaround time and waiting time.

Turnaround time is the interval from the time of submission of a process to the time of completion. Turnaround time is the sum of periods spent waiting to get into the ready queue , executing on the CPU and doing I/O. Waiting time is the average period of time a process spends waiting in the ready queue.

7) What is starvation or indefinite blocking?

The process that is ready to run but lacking the CPU can be considered as starvation or indefinite blocking.

8) Define aging.

A solution to the problem of the indefinite blocking or starvation is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Ex.No:04 **IMPLEMENT ALL FILE ALLOCATION STRATEGIES**

AIM:

To Write a C Program to implement file allocation technique.

ALGORITHM:

1. Start the process
2. Declare the necessary variables
3. Get the number of files
4.Get the total no. of blocks that fit in to the file
5. Display the file name, start address and size of the file.
6.Stop the program.

PROGRAM:

```
//contiguos.c

#include<stdio.h>
void main()
{
int i,j,n,block[20],start:
printf("Enter the no. of file:\n");
scanf("%d",&n);
printf("Enter the number of blocks needed for each file:\n");
for(i=0,i<n;i++)
scanf("%d",&block[i]);
start=0;
printf("\t\tFile name\tStart\tSize of file\t\t\n");
printf("\n\t\tFile1\t\t%d\t\t\t%d\n",start,block[0]);
for(i=2;i<=n;i++)
{
Start=start+block[i-2];
Printf("\t\tFile%d\t\t%d\t\tD\n",i,start,block[i-1]);
}
}
```

OUTPUT

```
[students@localhost ~]$  cc contiguous.c
[students@localhost ~]$ ./a.out
        Enter the number of file:4
        Enter the number of blocks needed for each file:
        3
        5
        6
        1
        Filename start size of file
        File 1          0 3
        File 2          3 5
        File 3          8 6
```

1. **Sequential File allocation**

The most common from 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 an d 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 o rder

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

b). 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].fl
ag)==0)
count++;
}
if(count==p[i])
break;
}
```

c). 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 fileno, lenth ,Blocks allocated.

Step 6: Stop the program

Program

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

int f[50],i,st,j,len,c,k,count=0;
for(i=0;i<50;i++)
f[i]=0;
X:
printf("\n enter starting block & length of files");
scanf("%d%d",&st,&len);
printf("\n file not allocated(yes-1/no-0)");
```

```
for(k=st;k<(st+len);k++)
if(f[k]==0)
count++;
if(len==count)
{
for(j=st;j<(st+len);j++)
if(f[i]==0)
{
f[j]=1;
printf("\n%d\t%d",,j,f[j]);
if(j==(st+len-1))
printf("\n the file is allocated to disk");
}
}
else
printf("file is not allocated");
count=0;
printf("\n if u want to enter more files(y-1/n-0)");
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
getch();
}
```

2. Indexed File allocation

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 fileno, lenth ,Blocks allocated.
Step 6: Stop the program

Program

```
#include<stdio.h>
#include<conio.h>
main()
{
int f[50],i,k,j,index[50],n,c,count=0;

for(i=0;i<50;i++)
f[i]=0;
X:
printf("enter index block");
scanf("%d",&i);
if(f[i]!=1)
{
f[i]=1;
printf("enter no of files on index");
scanf("%d",&n);
}
y:
for(i=0;i<n;i++)
scanf("%d",&index[i]);
if(f[index[i]==0)
count++;
if(count==n)
{ for(j=0;j<nj++)
f[index[j]=1;
printf("\nallocated");
printf("\n file indexed");
for(k=0;k<n;k++)
printf("\n%d->%d:%d",i,index[k],f[index[k]);
}
else
{
printf("\n file in the index already allocation");
printf("\nenter another file indexed");
goto y;
}
printf("\n index is already allocated");
count=0;
printf("\n if u enter one more block(1/0)");
scanf("%d",&c);
if(c==1)
goto x;
getch( );
}
```

3. Lined File allocation

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

ALGORTHIM:

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. While allocating next location address to attach it to previous location

```
for(i=0;i<n;i++)
{
for(j=0;j<s[i];j++)
{
q=random(100);
if(b[q].flag==0)
b[q].flag=1;
b[q]
.fno=j;
r[i][j]=q;
if(j>0)
{
p=r[i][j-1];
b[p].next=q;
}
}
}
```

Step 5: Print the results fileno, lenth ,Blocks allocated.
Step 6: Stop the program

Program:

```
#include<stdio.h>
#include<conio.h>
main()
{
int f[50],p,i,j,k,a,st,len,n;
for(i=0;i<50;i++)
f[i]=0;
printf("enter how many blocks already allocated");
scanf("%d",&p);
printf("\nenter the blocks nos");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
X:
printf("enter index sarting block & length");
scanf("%d%d",&st,&len);
k=len;
if(f[st]==0)
```

```
{
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++;
}
}
}
else
printf("\nif u enter one more (yes-1/no-0)");
scanf("%d",&c);
if(c==1)
goto X;
else
exit();
getch( );
}
```

Result:

Thus the C - program for Contiguous file allocation technique was implemented and output verified using various samples.

Viva Questions

1.Mention different file attributes.

i) Name
ii)Identifier
iii)Type
iv)Location
v)Size
vi)Protection
vii)Time,date,and user identification

2. Mention different file operations.

i) Creating a file
ii) Writing a file
iii) Reading a file
iv) Repositioning within a file
v) Deleting a file
vi) Truncating a file

3. Define immutable shared files.

• A unique approach is that of immutable shared files. Once a file is declared as shared by its creator it cannot modified.
• An immutable file has two key properties: Its name may not be reused and its contents may not be altered.
• Thus the name of an immutable file signifies that the contents of the file are fixed rather than the file being a container for variable information.
• The implementation of these semantics in a distributed system is simple because the sharing is disciplined (read-only).

4. How do you give the protection for files?

Protection can be provided in many ways. For a small single user system we might provide protection by physically removing the floppy disks and locking them in a disk drawer of file cabinet. In a multi-user system, however other mechanisms are needed.

1. Types of Access.
2. Access Control List (ACL).

5.What are the advantages of Linked allocation and Indexed allocation?

Advantages of Linked allocation: It solves the external-fragmentation and size declaration problems of contiguous allocation. Advantages of Indexed allocation: It supports the efficient direct access methods over the disk. It also solves the external-fragmentation and size-declaration problems of contiguous allocation.

6. What is FAT?

An important variation on the linked allocation method is the use of a File Allocation Table(FAT).

A benefit is that random access time is improved. This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems.

A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number.

The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file.

The table entry indexed by that block number then contains the block number of the next block in the file.

Ex.No:5 **IMPLEMENT SEMAPHORES**

PRODUCER CONSUMENT PROBLEM USING SEMAPHORES

AIM:

To write a C - Program to solve the Producer Consumer Problem using Semaphores.

ALGORITHM:

1. START the program
2. If the request is for Producer
    a. Get the count of number of items to be produced
    b. Add the count with current Buffer size
    c. If the new Buffer size is full
    Don't produce anything; display an error message
    Producer has to wait till any consumer consume the existing item
    d. Else
    Produce the item

Increment the Buffer by the number of item produced

3. If the request is for Consumer

a. Get the count of number of items to be consumed

b. Subtract the count with current Buffer size

c. If the new Buffer size is lesser than 0

Don't allow the consumer to consume anything; display an error message

Consumer has to wait till the producer produce new items

d. Else

Consume the item

Decrement the Buffer by the number of item consumed

4. STOP the program

PROGRAM:

```
// procon.c
#include<stdio.h>
int n=0,buffersize=0,currentsize=0;
void producer()
{
printf("\nEnter number of elements to be produced: ");
scanf("%d",&n);
if(0<=(buffersize-(currentsize+n)))
{
currentsize+=n;
printf("%d Elements produced by producer where buffersize is %d\n", currentsize,
buffersize);
}
else
printf("\nBuffer is not sufficient\n");
}
void consumer()
{
int x;
printf("\nEnter no. of elements to be consumed: ");
scanf("%d",&x);
if(currentsize>=x)
{
currentsize-=x;
printf("\nNumber of elements consumed: %d, Number of Elements left: %d", x,
currentsize);
}
else
{
printf("\nNumber of Elements consumed should not be greater than Number of Elements
produced\n");
}
}
void main()
```

```
{
int c;
printf("\nEnter maximum size of buffer:");
scanf("%d",&buffersize);
do
{
printf("\n1.Producer 2.Consumer 3.Exit");
printf("\nEnter Choice:");
scanf("%d",&c);
switch(c)
{
case 1:
if(currentsize >= buffersize)
printf("\nBuffer is full. Cannot produce");
else
producer(); break;
case 2:
if(currentsize <= 0)
printf("\nBuffer is Empty. Cannot consume");
else
consumer();
break;
default:
exit();
break;
}
}
while(c!=3);
}
```

OUTPUT: (Producer Consumer problem)

```
[students@localhost ~]$  cc procon.c
[students@localhost ~]$ ./a.out
```

/* producer & consumer*/

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
int buffer[45];
```

```
int p=0,c=0,con=0;
int max=45;
int ch;
void prod()
{
int val;
if(p>max)
{
printf("\nbuffer is full");
getch();

}
else
{
printf("\nenter the value\n");
scanf("%d",&buffer[p]);
p++;
}
}
void cons()
{
if(p==0)
{
printf("\n no data is buffer\n");
getch();

}
else if ((c>max)||(p==c))
{
printf("\n all data are consumed");
con=1;
getch();
}
else
{
printf("\n the values is %d",buffer[p-1]);
getch();
p--;
}
}
void main()
{
int yn,yn1;
while(ch!=3)
{
clrscr();
```

```
printf("\n\n menu");
printf("\n1.procedure 2.consumer\n3.exit");
scanf("%d",&ch);
switch(ch)
{
case 1:
do
{
prod();
printf("\n do u want to continue(1/0)\n");
scanf("%d",&yn);
}
while(yn!=0);
break;
case 2:
do
{ cons();
if(cons==0)
{
printf("do u want to continue(1/0)\n");
scanf("%d",&yn);
}
else
{
con=0;
break;
}
}
while(yn!=0);
case 3:
if(p!=0)
{
printf("\nthere r data 2 be consume\n");
getch();
}
else
{
printf("thank u\n");
getch();
exit(0);
} break;
default:
printf("\n invalid choice\n");
break;
```

```
}
}
}
```

RESULT:

Thus the C - program for producer consumer problem was solved using semaphores and output verified using various samples.

Viva Questions

1. Define semaphore?

Semaphore is used to solve critical section problem. A semaphore is a synchronization tool. Semaphore is a variable that has an integer value upon which the following operations are defined:

1) Wait
2) Signal

There are two types of semaphore 1. Counting semaphore 2. Binary semaphore.

2. What is busy waiting and spin lock?

While a process is in its critical section, any process that tries to enter its critical section must loop continuously in the entry code. This continual looping is called busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock.

3. Define binary semaphore

Binary semaphore is a semaphore with an integer value that can range only between 0 and 1

4 Define sleeping barber problem?

A barbershop consisting of a waiting room with n chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chair are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customers sits in one of the free chairs. If the barber is a sleep, the customer wakes up the barber.

5. Define cigarette-smokers problem.

There are three smoker process and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper and matches. One of the smoker processes has paper, another has tobacco and the third has matches. The agent places two of the ingredients on the table. The smoker who has the remaining ingredients then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out of the three ingredients and the cycle repeats. This is called cigarette-smokers problem.

Ex.No: 06 IMPLEMENT ALL FILE ORGANIZATION TECHNIQUES

AIM:

To write a C - Program to implement File Organization Techniques.

ALGORITHM:

Viva Questions

1. Define Virtual memory.

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Running a program that is not entirely in memory would benefit both the system and the user. Virtual memory is commonly implemented by demand paging. Virtual memory is available in the secondary storage.

2. Explain equal allocation & proportional allocation.

The easiest way to spilt m frames among n processes is to give everyone an equal share, m/n frames. This scheme is called equal allocation. We allocate available memory to each process according to its size. This scheme is called proportional allocation.

3. What is the difference between global and local replacement algorithm?

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

5. Define packing.

All disk I/O is performed in units of one block(physical record), and all blocks are same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

6. What are the operations performed in the directory?

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

Ex.No: 7 IMPLEMENT BANKERS ALGORITHM FOR DEAD LOCK AVOIDANCE

AIM:

To write a C - Program to implement Bankers Algorithm for Dead Lock Avoidance

Source Code:

```
#include<stdio.h>
#include<conio.h>
main()
{ int
i,j,a=0,b=0,c=0,f[10],t[10][10],al[10][10],ta[10][10];
int a1[10][10], max[10][10], n[10][10], n1,p,k=0;
printf("\n enter no.of resources");
scanf("%d",n1);
printf("\nenter the max no .of resources for each type");
for(i=0;i<n1;i++)
scanf("%d",&t[b][i]);
```

```
printf(“\nenter no .of process”);
scanf(“%d”,&p);
printf(“\nenter allocation resources”);
for(i=0;i<p;i++)
{ f[i]=0;
for(j=0;j<n1;j++)
scanf(“%d”,&a1[i][j]);
}
for(i=0;i<p;i++)
for(j=0;j<n1;j++)
{
if(a1[i][j]<=t[b][j])
{
t[b][j]+=a1[i][j];
continue;
}
else
printf(“\n wrong resourcesallocation”);
printf(“\n chance of deadlock occurrence after
allocation”);
for(j=0;j<n1;j++)
printf(“%d”,a1[b][j]);
printf(“\n enter the max resources for every process”);
for(i=0;i<p;i++)
for(j=0;j<n1;j++);
{
scanf(“%d”,&max[i][j]);
n[i][j]=max[i][j]-a1[i][j];
}
j=0;
printf(“\n needed resources for every process to start
execution”);
for(i=0;i<p;i++)
printf(“\n%d %d%d”,n[i][j],n[i][j+1],n[i][j+2]);
printf(“\n safe sequence the sequence of process to
compute
their  execution”);
for(a=0;a<(p-c);)
for(i=0;i<p;i++)
{
j=0;
b=0;
if(f[i]==0)
{
if(n[i][j]<=a1[b][j]&&n[i][j+1]<=a1[b][j+1]&&
```

```
n[i][j+2]<=a1[b][j+2])
{
printf("\n process %d execution started and
completed",i+1);
for(k=0;k<n-1;k++)
a1[b][k]+=a1[i][k];
f[i]=1;
c++;
} else
f[i]=0;
}
}
getch();
}
```

Ouput:

Input:

```
enter no.of resources
3
enter the max no .of resources for each type
10 5 7
enter the no .of process
5
enter allocation of resources
0 1 0 2 0 0 3 0 2 2 1 1 0 0 2
```

Output:

```
total available resources after allocation
3 3 2
enter the max resources for every process
7 5 3 3 2 2 9 0 2 2 2 2 4 3 3
needed resources for every process to start execution
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
Safe sequence ,the sequence of process to complete their
execution
Procee 2 execution started & completed
Procee 4 execution started & completed
Procee 5 execution started & completed
Procee 1 execution started & completed
Procee 3 execution started & completed
```

RESULT:

Thus the C - program for Bankers Algorithm for Dead Lock Avoidance was implemented and output verified using various samples.

Viva Questions:

1. Define bakery algorithm.

Bakery algorithm is used to solve the critical section problem for n process. The bakery algorithm was developed for a distributed environment, which permits processes to enter into the critical section in the order of the token numbers.

2. What is busy waiting and spin lock?

While a process is in its critical section, any process that tries to enter its critical section must loop continuously in the entry code. This continual looping is called busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock.

3. What is resource allocation graph?

Dead locks can be described more precisely in terms of directed graph called a system resource-allocation graph. This graph consists of set of vertices V and set of edges E.

4. What are the data structures required for the bankers algorithms?

The data structures used for bankers algorithm are

- Available
- Max
- Allocation
- Need

5. How do you recover the system from deadlock?

There are generally two methods to recover from deadlock

- Process termination
- Resource preemption

6. How do you avoid deadlock?

- System must always in safe state. Resource allocation graph should not consist of cycle. System must satisfy the safety and resource-request algorithm.
- Dont to start the process if demands might lead to deadlock
- Do not grant an increment resources request to a process if this allocation might lead to deadlock.

Ex.no.08 IMPLEMENT AN ALGORITHM FOR DEAD LOCK DETECTION

AIM:

To write a C - Program to implement Dead Lock Detection

Source Code:

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

```
main()
{ int
i,j,a=0,b=0,c=0,f[10],t[10][10],al[10][10],ta[10][10];
int a1[10][10], max[10][10], n[10][10], n1,p,k=0;
printf("\n enter no.of resources");
scanf("%d",n1);
printf("\nenter the max no .of resources for each type");
for(i=0;i<n1;i++)
scanf("%d",&t[b][i]);
printf("\nenter no .of process");
scanf("%d",&p);
printf("\nenter allocation resources");
for(i=0;i<p;i++)
{ f[i]=0;
for(j=0;j<n1;j++)
scanf("%d",&a1[i][j]);
}
for(i=0;i<p;i++)
for(j=0;j<n1;j++)
{
if(a1[i][j]<=t[b][j])
{
t[b][j]+=a1[i][j];
continue;
}
else
printf("\n wrong resourcesallocation");
printf("\n chance of deadlock occurrence after
allocation");
for(j=0;j<n1;j++)
printf("%d",a1[b][j]);
printf("\n enter the max resources for every process");
for(i=0;i<p;i++)
for(j=0;j<n1;j++);
{
scanf("%d",&max[i][j]);
n[i][j]=max[i][j]-a1[i][j];
}
j=0;
printf("\n needed resources for every process to start
execution");
for(i=0;i<p;i++)
printf("\n%d %d%d",n[i][j],n[i][j+1],n[i][j+2]);
printf("\n safe sequence the sequence of process to
```

```
compute
their execution");
for(a=0;a<(p-c);)
for(i=0;i<p;i++)
{
j=0;
b=0;
if(f[i]==0)
{
if(n[i][j]<=a1[b][j]&&n[i][j+1]<=a1[b][j+1]&&
n[i][j+2]<=a1[b][j+2])
{
printf("\n process %d execution started and
completed",i+1);
for(k=0;k<n-1;k++)
a1[b][k]+=a1[i][k];
f[i]=1;
c++;
} else
f[i]=0;
}
}
getch();
}
```

Viva Questions:

1. What is the necessary condition for deadlock?

A deadlock situation can arise if the following four conditions hold simultaneously in system: Mutual exclusion, Hold and wait, No preemption, Circular wait

2. How do you select a victim for resource preemption?

We must determine the order of preemption to minimize the cost

We must consider the cost parameters such as

- No of resources a dead lock process is holding.
- Amount of time consumed by the deadlock process during its execution

3. Define monitors.

Monitor is a programming language construct that provides equivalent functionality to that of the semaphores but it is easier to control. A monitor is characterized by a set of programmer-defined operators. Monitors are a high level data abstraction tool combining three features: Shared data, Operation on data, Synchronization, scheduling

4. What is real time scheduling? What are its types?

A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data. There are two types of real time scheduling

Hard real time systems are required to complete a critical task within a guaranteed amount of time.

Soft real time system, where a critical real-time task gets priority over other tasks, and retains that priority until it completes.

5. Define preemption points.

Preemption points are one of the ways to keep the dispatch latency low. Preemption points are usually used to see whether a high priority process needs to be run.

Ex.No 9 IMPLEMENTS ALL PAGE REPLACEMENT ALGORITHMS

(a)FIFO:

AIM:

To write C - Program to implement FIFO page replacement algorithm

ALGORITHM:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Forma queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

PROGRAM:

```
//fifo.c

#include<stdio.h>
int main()
{
int i,j,n,a[50],frame[10],no,k,avail,count=0;
printf("\n enter the number of pages:\n");
scanf("%d",&n);
printf("\n enter the page number:\n");
for(i=1;i<=n;i++)
scanf("%d",&a[i]);
printf("\n enter the number of frames:\n");
scanf("%d",&no);
for(i=0;i<no;i++)
frame[i]=-1;
j=0;
printf("\tref string\t page frmaes\n");
for(i=1;i<=n;i++)
{
printf("%d\t\t",a[i]);
avail=0;
for(k=0;k<no;k++)
```

```
if(frame[k]==a[i])
avail=1;
if(avail==0)
{ frame[j]=a[i];
j=(j+1)%no;
count++;
for(k=0;k<no;k++)
printf("%d\t",frame[k]);
}
printf("\n");
}
printf("page fault is %d",count);
getch();
return 0;
}
```

OUTPUT:

```
[students@localhost ~]$  cc fifo.c
[students@localhost ~]$ ./a.out
enter the number of pages:
4
enter the reference string:
7
2
1
0
enter the number of frames:
3
ref string page frmaes
7 7 -1 -1
2 7 2 -1
1 7 2 1
0 0 2 1
page fault is 4
```

RESULT:

Thus the C - Program for FIFO page replacement was implemented and output verified using various samples.

(b)LRU:

AIM:

To write a C - program to implement LRU page replacement algorithm

ALGORITHM :

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack

6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

PROGRAM:

```
//lru.c

#include<stdlib.h>
#include<stdio.h>
#define max 100
#define min 10
int ref[max],count,frame[min],n;
void input()
{
int i,temp;
count=0;
printf("\n\n\tEnter the number of page frames : ");
scanf("%d",&n);
printf("\n\n\tEnter the reference string (-1 for end) : ");
scanf("%d",&temp);
while(temp != -1)
{
ref[count++]=temp;
scanf("%d",&temp);
}
}
void LRU()
{
int i,j,k,stack[min],top=0,fault=0;
system("CLS");
for(i=0;i<count;i++)
{ if(top<n)
stack[top++]=ref[i],fault++;
else
{ for(j=0;j<n;j++)
if(stack[j]==ref[i])
break;
if(j<n)
{
for(k=j;k<n-1;k++)
stack[k]=stack[k+1];
stack[k]=ref[i];
}
else
```

```
{
for(k=0;k<n-1;k++)
stack[k]=stack[k+1];
stack[k]=ref[i];
fault++;
}
}
printf("\n\nAfter inserting %d the stack status is : ",ref[i]);
for(j=0;j<top;j++)
printf("%d ",stack[j]);
}
printf("\n\n\tEnd to inserting the reference string.");
printf("\n\n\tTotal page fault is %d.",fault);
printf("\n\n\tPress any key to continue.");
}
void main()
{
int x;
//freopen("in.cpp","r",stdin);
while(1)
{
printf("\n\n\t-----MENU-----");
printf("\n\t1. Input ");
printf("\n\t2. LRU (Least Recently Used) Algorithm");
printf("\n\t0. Exit.");
printf("\n\n\tEnter your choice.");
scanf("%d",&x);
switch(x)
{
case 1:
input();
break;
case 2:
LRU();
break;
case 0:
exit(0);
}
}
}
```

OUTPUT:

```
[students@localhost ~]$  cc lru.c
[students@localhost ~]$ ./a.out
-----MENU-----
1. Input
2. LRU (Least Recently Used) Algorithm
```

0. Exit.
Enter your choice.1
Enter the number of page frames : 3
Enter the reference string (-1 for end) : 2
0
1
1
-1
-----MENU-----
1. Input
2. LRU (Least Recently Used) Algorithm
0. Exit.
Enter your choice.2
After inserting 2 the stack status is : 2
After inserting 0 the stack status is : 2 0
After inserting 1 the stack status is : 2 0 1
After inserting 1 the stack status is : 2 0 1
End to inserting the reference string.
Total page fault is 3.
Press any key to continue.

RESULT:

-----MENU-----
1. Input
2. LRU (Least Recently Used) Algorithm
0. Exit.
Enter your choice.0
The program for LRU page replacement was implanted and hence verified.

Result:

Thus the C - program for LRU Page replacement was implemented and output verified using various samples.

Viva Questions:

1. What is paging?

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Paging reduces the external fragmentation.

2. What are the advantages of Paging?

• Support higher degree of multiprogramming.
• Paging reduces fragmentation.
• Paging increases memory and processor utilization.
• Compaction overhead required for the relocatable partition scheme is also eliminated.

3. What is TLB?

The TLB is associative, high speed memory. Each entry in the Translation Look aside Buffer (TLB) consists of two parts: a key or tag and a value. The search is fast, hardware is expensive. The number of entries in a TLB is small, often numbering
between 64 and 1,024.

4. Define segmentation.

Segmentation is a memory management scheme. Segmentation divides a program into a number of smaller blocks called segments. A segment can be defined as a logical grouping of information, such as sub routine, array or data area. **Segmentation** is variable size.

5. What are the different types of page table structure?

- Hierarchical Paging
- Hashed page table
- Inverted page table

Ex.No:10 IMPLEMENT SHARED MEMORY AND IPC
(USING SHARED MEMORY, PIPES OR MESSAGE QUEUES)

(a)Shared memory:

AIM:

To write a C – Progarm to implement the interprocess communication using shared memory.

ALGORITHM:

1. Start the program
2. Declare the necessary variables
3. shmat() and shmdt() are used to attach and detach shared memory segments. They are prototypes as follows:
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
4. shmat() returns a pointer, shmaddr, to the head of the shared segment associated with a valid shmid. shmdt() detaches the shared memory segment located at the address indicated by shmaddr
5. Shared1.c simply creates the string and shared memory portion.
6. Shared2.c attaches itself to the created shared memory portion and uses the string (printf)
7. Stop the program.

PROGRAM:

```
//shared1.c:

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27
main()
{
char c;
int shmid;
key_t key;
char *shm, *s;
key = 5678;
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
```

```
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
perror("shmat");
exit(1);
}

//shared2.c

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

OUTPUT:

```
[students@localhost ~]$  cc shared2.c
[students@localhost ~]$ ./a.out
        Abcdefghijklmnopqrstuvwxyz
```

RESULT:

Thus the C - program for interprocess communication using shared memory was implemented and output verified using various samples.

(b)PIPES:

AIM:

To write a C – Program to implement the interprocess communication using pipes.

ALGORITHM:

1. Start the program
2. Create the child process using fork()
3. Create the pipe structure using pipe()

4. Now close the read end of the parent process using close()
5. Write the data in the pip[e using write()
6. Now close the write end of child process using close()
7. Read the data in the pipe using read()
8. Display the string
9. Stop the program.

PROGRAM:

```
// pipes.c
#include<stdio.h>
int main()
{
int fd[2],child;
char a[10];
printf("\n enter the string to enter into the pipe:");
scanf("%s",a);
pipe(fd);
child=fork();
if(!child)
{   close(fd[10]);
write(fd[1],a,5);
wait(0);
}
else
{ close(fd[1]);
read(fd[0],a,5);
printf("\n the string retrived from pipe is %s\n",a);
}
return 0;
}
```

OUTPUT:

```
[students@localhost ~]$  cc pipes.c
[students@localhost ~]$ ./a.out
        enter the string to enter into the pipe:computer
        the string retrived from pipe is computer
```

RESULT:

Thus the C - program for interprocess communication using pipes was implemented and output verified using various samples.

(c)message queue:

AIM:

To write a C – Program to implement the Interprocess communication using message passing.

ALGORITHM:

1. Start the program
2. Create two files msgsend and msgrecv.c
3. Msgsend creates a message queue with a basic key and message flag msgflg =

IPC_CREAT | 0666 -- create queue and make it read and appendable by all, and sends one message to the queue.
4. Msgrecv reads the message from the queue
5. A message of type (sbuf.mtype) 1 is sent to the queue with the message ``Did you get this?"
6. The Message queue is opened with msgget (message flag 0666) and the ***same*** key as message_send.c
7. A message of the ***same*** type 1 is received from the queue with the message ``Did you get this?" stored in rbuf.mtext.
8. Stop the program.

PROGRAM:

```
// msgsend.c

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#define MSGSZ 128
typedef struct msgbuf {
long mtype;
char mtext[MSGSZ];
} message_buf;
main()
{
int msqid;
int msgflg = IPC_CREAT | 0666;
key_t key;
message_buf sbuf;
size_t buf_length;
key = 1234;
(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);
if ((msqid = msgget(key, msgflg )) < 0) {
perror("msgget");
exit(1);
}
else
(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
sbuf.mtype = 1;
(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
(void) strcpy(sbuf.mtext, "Did you get this?");
(void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);
buf_length = strlen(sbuf.mtext) + 1 ;
```

```
if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
perror("msgsnd");
exit(1);
}
else
printf("Message: \"%s\" Sent\n", sbuf.mtext);
}
// msgrecv.c

exit(0);
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
typedef struct msgbuf {
long mtype;
char mtext[MSGSZ];
} message_buf;
main()
{
int msqid; key_t
key; message_buf
rbuf;
key = 1234;
if ((msqid = msgget(key, 0666)) < 0) {
perror("msgget");
exit(1);
}
if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
perror("msgrcv");
exit(1);
}
printf("%s\n", rbuf.mtext);
eixit(0);
}
```

OUTPUT:

```
[students@localhost ~]$  cc msgrecv.c
[students@localhost ~]$ ./a.out
        "Did you get this?"
```

RESULT:

Thus the C - program for interprocess communication using message passing was implemented and output verified using various samples.

Viva Questions

1. Differentiate Logical Address Space & Physical Address Space.

Logical Address Space and Physical Address Space

Logical address is generated by CPU. Physical Address is an address of main memory. Set of all logical addresses generated by program is a Logical Address Space. Set of all physical addresses corresponding to logical address is a Physical Address Space.

2. Define Compaction.

When enough total memory space exists to satisfy a request, but it is not contiguous, storage is fragmented into a large number of small holes. This situation leads to the external fragmentation. One of the solutions to this problem is compaction. Compaction is to move all the allocated holes to one side and all free holes are moved to another side.

3. How do you calculate the effective memory access time?

To find effective memory access time, we must weigh each by its probability:

For example, consider a page number that is found 80 percent of times in a TLB. If it takes 20 nanoseconds to search and 100 nanoseconds to access the memory, then the mapped memory access takes 120 nanoseconds. If we fails to find page number in TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds), and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

Effective memory access time = 0.80*120+0.20*220= 140 nanoseconds.

4. Define Belady's Anomaly.

For some page replacement algorithms, the page fault rate may increase as the number of allocated frames increases. This problem is known as Belady's Anomaly. FIFO page replacement algorithm may affect this unexpected problem.

5. Define Overlays.

The idea to keep in memory only those instructions and data that are needed at any given time are known as overlays. We can use overlays to enable a process to be larger than the amount of memory allocated to it.

6. Define Swapping.

It is a technique of temporarily removing inactive program from the memory of a system. It removes the process from the primary memory when it is blocked and deallocating the memory. Then this free memory is allocated to other processes.

Ex.No.11 Implement Paging Technique of memory management.

AIM:

A program to simulate Paging technique of memory management.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
main()
{int np,ps,i;
int *sa;
clrscr();
printf("enter how many pages\n");
scanf("%d",&np);
printf("enter the page size \n"); scanf("%d",&ps);
sa=(int*)malloc(2*np);
for(i=0;i<np;i++)
```

```
{
sa[i]=(int)malloc(ps);
printf("page%d\t address %u\n",i+1,sa[i]);
}
getch();
}
```

RESULT:

Thus the C - program for Paging Technique of memory management was implemented and output verified using various samples.

Viva Questions:

1.What is paging?

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Paging reduces the external fragmentation.

2. What are the advantages of Paging?

- Support higher degree of multiprogramming.
- Paging reduces fragmentation.
- Paging increases memory and processor utilization.
- Compaction overhead required for the relocatable partition scheme is also eliminated.

3. Define reentrant code or pure code.

Re-entrant code is non-self-modifying code. If the code is re-entrant, then it never changes during execution.

4. Define segmentation.

Segmentation is a memory management scheme. Segmentation divides a program into a number of smaller blocks called segments. A segment can be defined as a logical grouping of information, such as sub routine, array or data area. **Segmentation** is variable size.

5. Differentiate LDT and GDT.

Local Descriptor Table Global Descriptor Table Information about the first partition among the two partitions of the logical address space is kept in the LDT. Information about the second partition among the two partitions of the logical address space is kept in the GDT. This first process is private to the processes. This second process is shared among to the processes.

6.What is address binding?

Binding the memory address of instructions and data is called as address binding. Address binding can be done in three ways

1. Compile time
2. Load time
3. Execution time.

Ex.No.12 Implement Threading & Synchronization Applications

AIM:

To implement Threading and Synchronization applications using C-Programming.

Program:

Thread Cretation:

```
#include <pthread.h>
pthread_attr_t tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Thread Termination:

```
#include <pthread.h>
pthread_t tid;
int ret;
int status;
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);
/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

Process 1:

```
// for WinXp
#define _WIN32_WINNT 0x0501
#include <windows.h>
#include <stdio.h>

int main()
{
// one process creates the mutex object.
HANDLE hMutex;
char * MName = "MyMutex";
hMutex = CreateMutex(
  NULL,        // no security descriptor
FALSE,     // mutex not owned
MName);   // object name
if (hMutex == NULL)
  printf("CreateMutex(): error: %d.\n", GetLastError());
else
```

```
{
if (GetLastError() == ERROR_ALREADY_EXISTS)
printf("CreateMutex(): opened existing %s mutex.\n", MName);
else
 printf("CreateMutex(): new %s mutex successfully created.\n", MName);
}
 return 0;
}
```

Process 2:

```
#define _WIN32_WINNT 0x0501
#include <windows.h>
#include <stdio.h>
int main()
{
HANDLE hMutex1;
hMutex1 = OpenMutex( MUTEX_ALL_ACCESS,
   // request full access FALSE,       // handle not
   inheritable MName);          // object name
if (hMutex1 == NULL)
printf("OpenMutex(): error: %d.\n", GetLastError());
else
    printf("OpenMutex(): %s mutex opened successfully.\n", MName);
return 0;
}
```

Prcess 3:

```
#define _WIN32_WINNT 0x0501
#include <windows.h>
#include <stdio.h>
int main()
{
// one process creates the mutex object.
HANDLE hMutex;
char * MName = "MyMutex";
hMutex = CreateMutex(
   NULL,        // no security descriptor
  FALSE,     // mutex not owned
  MName);   // object name
if (hMutex == NULL)
  printf("CreateMutex(): error: %d.\n", GetLastError());
else
{
   if (GetLastError() == ERROR_ALREADY_EXISTS)
```

```
        printf("CreateMutex(): opened existing %s mutex.\n", MName);
      else
        printf("CreateMutex(): new %s mutex successfully created.\n", MName);
    }

//=================================================================
    HANDLE hMutex1;
    hMutex1 = OpenMutex( MUTEX_ALL_ACCESS, //
      request full access FALSE,        // handle not
      inheritable MName);         // object name

    if (hMutex1 == NULL)
        printf("OpenMutex(): error: %d.\n", GetLastError());
    else
        printf("OpenMutex(): %s mutex opened successfully.\n", MName);

     return 0;
    }
```

Semaphore (Using Semaphore for Synchronization)

The following example uses the CreateSemaphore() function to illustrate a named-object creation operation that fails if the object already exists.

```
    // for WinXp
    #define _WIN32_WINNT 0x0501
    #include <windows.h>
    #include <stdio.h>

    HANDLE CreateNewSemaphore(LPSECURITY_ATTRIBUTES lpsa, LONG cInitial,
    LONG cMax, LPTSTR lpszName)
    {
      HANDLE hSem;
      // create or open a named semaphore.
      hSem = CreateSemaphore(
        lpsa,             // security attributes, NULL = handle cannot be inherited
        cInitial,          // initial count
        cMax,           // maximum count
        lpszName);    // semaphore name

      // close handle, and return NULL if existing semaphore opened.
      if (hSem == NULL && GetLastError() == ERROR_ALREADY_EXISTS)
      {
        CloseHandle(hSem);
        return NULL;
      }
```

```
        else
        {
            printf("Checking the last error: %d.\n", GetLastError());
            printf("CreateNewSemaphore(): New semaphore was created successfully.\n");
        }
    // if new semaphore was created, return the handle.
    return hSem;
}
int main()
{
    LPSECURITY_ATTRIBUTES lpsa = NULL;
    LONG cInitial = 0;
    LONG cMax = 10;
    LPTSTR lpszName = "MySemaphore";

    HANDLE hSemaphore = CreateNewSemaphore(lpsa, cInitial, cMax, lpszName);
    return 0;
}
```

RESULT:

Thus the C - program for implementing Threading and Synchronization application was implemented and output verified using various samples.

Viva Questions:

1. Define Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler (typically as part of an operating system). The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is a component of a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).

2. What are the multithreading models in OS?

- Many-to-One Model
- One-to-One Model
- Many-to-Many Model

3. What is cascading termination?

If a process terminates, then all its children must also be terminated. This phenomenon, referred to as cascading termination

4. Define microkernel.

The microkernel method structures the OS by removing all nonessential components from the kernel, and implementing them as system and user-level programs. The result is a smaller kernel. Microkernel provide minimal process and memory management in addition to a communication facility.

5. What is Thread? Mention the benefits of Multithreaded Programming.

A thread is a flow of execution through the process code, with its own program counter, system register and stack. It is a light weight process.

BENEFITS:

- Responsiveness.
- Resource Sharing.
- Economy.
- Utilization of multiprocessor architecture.

6. Define user and kernel thread.

User threads are supported above the kernel and are implemented library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel.

Kernel threads are supported directly by the operating system: the kernel performs thread creation, scheduling, and management in kernel space