EX.NO:1	ANALYSIS AND SYNTHESIS OF BASIC COMMANDS AND PROGRAMS IN SHELL PROGRAMMING

A. BASIC COMMANDS

Command	Example	Description
Cat	cat .bashrc	Sends file contents to standard output. This is a way to list the contents of short files to the screen. It works well with piping Sends the contents of the ".bashrc" file to the screen.
Cd	cd /home	Change directory Change the current working directory to /home. The '/' indicates relative to root, and no matter what directory you are in when you execute this command, the directory will be changed to "/home".
	cd httpd	Change the current working directory to httpd, relative to the current location which is "/home". The full path of the new working directory is "/home/httpd".
	cd ..	Move to the parent directory of the current directory. This command will make the current working directory "/home.
	cd ~	Move to the user's home directory which is "/home/username". The '~' indicates the users home directory.
Cp		Copy files
	cp myfile yourfile	Copy the files "myfile" to the file "yourfile" in the current working directory. This command will create the file "yourfile" if it doesn't exist. It will normally overwrite it without warning if it exists.
	cp -i myfile yourfile	With the "-i" option, if the file "yourfile" exists, you will be prompted before it is overwritten.
	cp -i /data/myfile .	Copy the file "/data/myfile" to the current working directory and name it "myfile". Prompt before overwriting the file.
	cp -dpr srcdir destdir	Copy all files from the directory "srcdir" to the directory "destdir" preserving links (-p option), file attributes (-p option), and copy recursively (-r option). With these options, a directory and all it contents can be copied to another directory.
Dd	dd if=/dev/hdb1 of=/backup/	Disk duplicate. The man page says this command is to "Convert and copy a file", but although used by more advanced users, it can be a very handy command. The "if" means input file, "of" means output file.
Df		Show the amount of disk space used on each mounted filesystem.
less	less textfile	Similar to the more command, but the user can page up and down through the file. The example displays the contents of textfile.
Ln		Creates a symbolic link to a file.
	ln -s test	Creates a symbolic link named symlink that points to the file test

	symlink	Typing "ls -i test symlink" will show the two files are different with different inodes. Typing "ls -l test symlink" will show that symlink points to the file test.
locate		A fast database driven file locator.
	slocate -u	This command builds the slocate database. It will take several minutes to complete this command. This command must be used before searching for files, however cron runs this command periodically on most systems.
	locate whereis	Lists all files whose names contain the string "whereis".
logout		Logs the current user off the system.
Ls		List files
	ls	List files in the current working directory except those starting with . and only show the file name.
	ls -al	List all files in the current working directory in long listing format showing permissions, ownership, size, and time and date stamp
more		Allows file contents or piped output to be sent to the screen one page at a time
	more /etc/profile	Lists the contents of the "/etc/profile" file to the screen one page at a time.
	ls -al |more	Performs a directory listing of all files and pipes the output of the listing through more. If the directory listing is longer than a page, it will be listed one page at a time.
mv		Move or rename files
	mv -i myfile yourfile	Move the file from "myfile" to "yourfile". This effectively changes the name of "myfile" to "yourfile".
	mv -i /data/myfile .	Move the file from "myfile" from the directory "/data" to the current working directory.
pwd		Show the name of the current working directory
shutdown		Shuts the system down.
	shutdown -h now	Shuts the system down to halt immediately.
	shutdown –r now	Shuts the system down immediately and the system reboots.
whereis		Show where the binary, source and manual page files are for a Command
	whereis ls	Locates binaries and manual pages for the ls command.

B.SHELL PROGRAMMING

Objectives:

To implement Shell programming using Command syntax, Substitutions, Expansion, Simple functions, Patterns and Loops.

Learning Outcomes:

After the completion of the experiment, Student will be able to

- Understand the basic Commands in UNIX.
- Write simple functions, loops, Patterns, Expansions, Substitutions.

Problem Statement:

The program in shell scripting language to find,

- Greatest of three numbers
- Factorial of N Numbers.
- Sum of N numbers.
- Odd or Even Number.
- Fibonacci Series
- Multiplication Table.
- Swapping of Two Numbers.
- File manipulation.
- Palindrome or not
- Positive or negative number.
- Prime number or not.
- Area of different shapes.

System and Software tools required:

Software Required **:** TURBO C version 3 (or) GCC version 3.3.4

Operating System **:** WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required **:** Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB Hard disk

Algorithm

Greatest of three numbers:

- Start the program.
- Enter any three numbers.
- It will check the condition $a -gt $b -a $a -gt $c.
- If a is greater than y print a is greater.
- If b is not greater then compare b and c.
- If b is greater than c print b is greater.
- Else print c is greater.

- Stop the program.
- Execute the program.

Execution of the Program:

$ sh filename.sh// Executing the program

Sample Coding:

// Greatest among 3 numbers

```
echo Enter 3 numbers with spaces in between

read a b c
l=$a
if [ $b -gt $l ]
then
l=$b
fi
if [ $c -gt $l ]
then
l=$c
fi
echo Largest of $a $b $c is $l
```

Sample Output:

```
Enter 3 numbers with spaces in between
3 8 5
Largest of 3 8 5 is 8
```

Test cases:

Enter Numbers in different ranges.
Include choices for executing the many concepts in one program

Sample Coding

// Factorial

```
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 factorial number of $ni is $fact
```

Sample Output:

Enter the number :
4
The factorial of 4 is 24.

Pre lab Questions:

1. How do you find out what's your shell? - echo $SHELL
2. How do you refer to the arguments passed to a shell script? - $1, $2 and so on. $0 is your script name.
3. What's the conditional statement in shell scripting? - if {condition} then … fi
4. What is the use of break command?
5. What is the use of continue command in shell scripting?
6. Tell me the Syntax of "Case statement" in Linux shell scripting?
7. What is the basic syntax of while loop in shell scripting?
8. How to make a shell script executable?
9. What is the use of "#!/bin/bash" ?
10. What is the syntax of for loop in shell script?
11. How to debug a shell script?
12. How compare the strings in shell script?
13. What are the Special Variables set by Bourne shell for command line arguments?
14. How to test files in a shell script?
15. How to perform arithmetic operation?
16. Write the Basic Syntax of do-while statement?
17. How to define functions in shell scripting?
18. How to use bc (bash calculator) in a shell script?

Conclusion:

Thus the implementation of basic commands and shell program using Command syntax, Substitutions, Expansion, Simple functions, Patterns and Loops was implemented successfully.

EX.NO:2	IMPLEMENTATION OF UNIX SYSTEM CALLS

Objectives:

To write a program to simulate various UNIX commands in C.
a.ls and grep command
b.Process system calls of UNIX operating system: fork(),exec(),getpid,exit,wait.
c.I/O System calls :Open(), Create(), Read(), Write().

Learning Outcomes:

After the completion of this experiment, student will be able to

- Simulate the simple list directory 'ls' command in UNIX using the inbuilt system calls to create this functionality.
- Simulate grep command.
- Create a new process called child process
- Both the child and parent continue to execute the instructions following fork call.
- The child can start execution before the parent or vice-versa.
- To familiarize you with the input output system calls in unix.
- Create a new process called child process
- Both the child and parent continue to execute the instructions following fork call.
- The child can start execution before the parent or vice-versa.

Problem Statement:

A program to implement the following system calls

- ls
- grep
- fork()
- exec()
- wait()
- exit()
- Open()
- Create()
- Read()
- Write()

System and software tools required:

Software Required : TURBO C version 3 (or) GCC version 3.3.4

Operating System : WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required : Minimum Requirement: Pentium III or Pentium IV with 256 RAM and
40 GB Hard disk

Algorithm:

1. Store path of current working directory using getcwd system call.
2. Scan directory of the stored path using scandir system call and sort the resultant array of structure.
3. Display dname member for all entries if it is not a hidden file.
4. Stop

Fork()

1. Declare a variable x to be shared by both child and parent.
2. Create a child process using fork system call.
3. If return value is -1 then
   a. Print "Process creation unsuccessful"
   b. Terminate using exit system call.
4. If return value is 0 then
   a. Print "Child process"
   b. Print process id of the child using getpid system call
   c. Print value of x
   d. Print process id of the parent using getppid system call
5. Otherwise
   a. Print "Parent process"
   b. Print process id of the parent using getpid system call
   c. Print value of x
   d. Print process id of the shell using getppid system call.

Create()

1. Declare a character buffer buf to store 100 bytes.
2. Get the new filename as command line argument.
3. Create a file with the given name using open system call with O_CREAT and O_TRUNC options.
4. Check the file descriptor.
   a) If file creation is unsuccessful, then stop.
5. Get input from the console until user types Ctrl+D
   a) Read 100 bytes (max.) from console and store onto buf using read system call
   b) Write length of buf onto file using write system call.
6. Close the file using close system call.
7. Stop

Read()

1. Declare a character buffer buf to store 100 bytes.
2. Get existing filename as command line argument.
3. Open the file for reading using open system call with O_RDONLY option.
4. Check the file descriptor.
   a) If file does not exist, then stop.
5. Read until end-of-file using read system call.
   a) Read 100 bytes (max.) from file and print it
6. Close the file using close system call.
7. Stop

Write():

1. Declare a character buffer buf to store 100 bytes.
2. Get exisiting filename as command line argument.
3. Create a file with the given name using open system call with O_APPEND option.
4. Check the file descriptor.
    a) If value is negative, then stop.

5. Get input from the console until user types Ctrl+D
    a) Read 100 bytes (max.) from console and store onto buf using read system call
    b) Write length of buf onto file using write system call.
6. Close the file using close system call.
7. Stop

Execution of the Program:

```
$ cc list.c -o list    //Compiling the Program

$ ./list           //Executing the program
```

Sample Coding:

```
/* ls command simulation - list.c */

#include <stdio.h>

#include <dirent.h>

main()

{
        struct dirent **namelist;
        int n,i;

        char pathname[100];

        getcwd(pathname);

        n = scandir(pathname, &namelist, 0, alphasort);

        if(n < 0) printf("Error\n");

        else
        for(i=0; i<n; i++) if(namelist[i]- >d_name[0] != '.')

        printf("%-20s", namelist[i]->d_name);

}
```

Sample Output:

```
                                consumer.
a.out           cmdpipe.c       c
dirlist.c       ex6a.c          ex6b.c
ex6c.c          ex6d.c          exec.c
fappend.c       fcfs.c          fcreate.c
fork.c          fread.c         Hello
```

List	list.c	pri.c
producer.c	rr.c	simls.c
sjf.c	stat.c	wait.c

Test cases:

- Simulate ls run for default directory
- Simulate ls run for sub directory

Execution of the Program:

$ cc fork.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
/* Process creation - fork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
 #include <sys/types.h>
main()
{
pid_t pid;
int x = 5;
pid = fork();
x++; if (pid < 0)
          {
          printf("Process creation error");
          exit(-1);
        }
  else if (pid
     == 0)
        {
          printf("Child process:");
          printf("\nProcess id is %d", getpid());
          printf("\nValue of x is %d", x);
          printf("\nProcess id of parent is %d\n", getppid());
          }
else {
          printf("\nParent process:"); printf("\nProcess
          id is %d", getpid()); printf("\nValue of x is
          %d", x); printf("\nProcess id of shell is %d\n",
          getppid()); }

}
```

Sample Output:

Child process:

Process id is 19499

Value of x is 6

Process id of parent is 19498

Parent process:

Process id is 19498

Value of x is 6

Process id of shell is 3266

Test cases:

Create Child Process Successfully, with positive values confirmed.

Unsuccessful creation confirmed with negative values.

Sample Program :
Create()

```
fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC,
0644); if(fd < 0)
{
printf("File creation
problem\n"); exit(-1);
}
printf("Press Ctrl+D at end in a new line:\n");
while((n = read(0, buf, sizeof(buf))) > 0)
{
len = strlen(buf);
write(fd, buf, len);
}
Close(fd);
}
```

Read()

```
fd = open(argv[1],
O_RDONLY); if(fd == -1)
{
printf("%s file does not exist\n",
argv[1]);
exit(-1);
}
printf("Contents of the file %s is : \n",argv[1]);
while(read(fd, buf, sizeof(buf)) > 0)
printf("%s", buf);
close(fd);

}
```

Write()

```
fd = open(argv[1], O_APPEND|O_WRONLY|O_CREAT,0644);
if (fd < 0)
```

```
{
perror(argv[1]);
exit(-1);
}
while((n = read(0, buf, sizeof(buf))) > 0)
{
len = strlen(buf);
write(fd, buf, len);
}
close(fd);
}
```

Pre-lab questions:

1. Write a command to print the lines that has the the pattern "july" in all the files in a particular directory?
2. Write a command to print the lines that has the word "july" in all the files in a directory and also suppress the filename in the output
3. Write a command to print the lines that has the word "july" while ignoring the case.
4. Write a Unix command to display the lines in a file that do not contain the word "july"?
5. Write a command to print the lines that starts with the word "start"?
6. Write a command to print the lines which end with the word "end"?
7. Write a command to select only those lines containing "july" as a whole word?
8. You are supposed to print the content of directory in long format listing, showing hidden/dot files. How will you achieve this?
9. Write an ls command to display the hidden files and directories?
10. Write an ls command to display the inode number of file?
11. Write an ls command to sort the files in ascending order of modification time?
12. Write an ls command to print the files in a specific directory?
13. Write an ls command to display files in columns?
14. What is the purpose of system calls?
15. When a process creates a new process using the fork() operation, which of the following state is shared between the parent process and the child process?
16. a. Stack b. Heap c. Shared memory segments
17. State the types of system calls.
18. What is the purpose of fork(),exit(),wait() system calls.
19. List out unix system calls for file manipulation.
20. List out system calls for Device manipulation.
21. List out system calls for Communication
22. List out the different input output system calls.
23. What is the purpose of I/O system calls?
24. Is it possible to see information about a process while it is being executed?
25. What is pid?

Conclusion:

Thus the C program to implement various system calls was done and the output was verified.

EX.NO:3	Simulation and Analysis of non Preemptive and Preemptive scheduling algorithm

Objectives:

To implement non Preemptive and Preemptive scheduling algorithm and display Gantt Chart,
turnaround time and waiting time in C.

a. FCFS Non Preemptive scheduling algorithm
b. SJF Non Preemptive scheduling algorithm
c. Priority Non Preemptive scheduling algorithm
d. RR Preemptive scheduling algorithm

Learning Outcomes:

After the completion of this experiment, student will be able to

- Schedule the processes or jobs to the CPU
- Will be able to create Gantt Chart and Calculate the Average waiting time and turnaround time
- recollect the looping concepts in C

Problem Statement:

The program to perform scheduling for the processes using non preemptive and preemptive methods to display the following

1. Gantt chart
2. Average waiting time
3. Average turnaround time
4. Processes in the Queue

System and Software tools required:

Software Required **:** TURBO C version 3 (or) GCC version 3.3.4

Operating System **:** WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required **:** Minimum Requirement: Pentium III or Pentium IV with 256 RAM and
40 GB Hard disk

a.FCFS NON PREEMPTIVE SCHEDULING ALGORITHM

Algorithm:

1. Create the number of process.
2. Get the ID and Service time for each process.

3. Initially, waiting time of first process is zero and Total time for the first process is the starting time of that process.
4. Calculate the Total time and Processing time for the remaining processes.
5. Waiting time of one process is the Total time of the previous process.
6. Total time of process is calculated by adding Waiting time and Service time.
7. Total waiting time is calculated by adding the waiting time for lack process.
8. Total turnaround time is calculated by adding all total time of each process.
9. Calculate Average waiting time by dividing the total waiting time by total number of process.
10. Calculate Average turnaround time by dividing the total time by the number of process.
11. Display the result.

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
 main()
{
        scan the pids;
        scan the bursts;
        for (counter=0 ; not end of process_queue ; counter++) {
                print "dispatching the process : process_queue[counter]" ;
        }
}
```

Sample Output:

```
Enter the total no of processes
3
Enter the process id
1
2
3

Enter the burst time
5
3
7
p[1]=5
p[2]=3
p[3]=7
Gantt Chart

-------------------------------------------------------------------------------
|       p[1]    |       p[2]    |       p[3]    |
```

---
0 5 8 15
Scheduling.....

Test cases:

Enter three processes with the burst time of your own

Enter a maximum of 10 processes.

Enter four processes P1, P2, P3, and P4 with 6,4,5,2 as the duration of each process.

Enter 7 processes with increasing burst time and display the waiting time of the 7th process separately.

B. SJF NON PREEMPTIVE SCHEDULING ALGORITHM

Algorithm:

1. Define an array of structure process with members pid, btime, wtime & ttime.
2. Get length of the ready queue, i.e., number of process (say n).
3. Obtain btime for each process.
4. Sort the processes according to their btime in ascending order.
    a. If two process have same btime, then FCFS is used to resolve the tie.
5. The wtime for first process is 0.
6. Compute wtime and ttime for each process as:
    a. wtime i+1= wtime i+ btime i b. ttime i = wtime i + btime i
7. Compute average waiting time awat and average turnaround time atur.
8. Display btime,ttime and wtime for each process.
9. Display GANTT chart for the above scheduling.
10. Display awat and atur.
11. Stop.

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
 void sort_processess
{
/*
 *sorts the processess in the
 *increasing order of burst time
 */
}
```

```
main ()
{

        scan the pids;
        scan the bursts;

        sort_processess ();
        for (counter=0 ; not end of process_queue ; counter++) {
                print "dispatching the process : process_queue[counter]" ;
        }
}
```

Sample Output:

Enter the number of process:3
Enter the process and their respective burst time:

PROCESS	BURSTTIME
P1	20
P2	4
P3	2

The chart for shortest job first.....

```
 _____ _____ _____
|  P3|  P2|  P1|
0    2.000000  6.000000  26.000000
```

Test cases:

Enter three processes with the burst time of your own

Enter a maximum of 10 processes.

Enter four processes P1, P2, P3, and P4 with 6,4,5,2 as the duration of each process and find which process gets executed first.
Enter 7 processes with increasing burst time and display the waiting time of the 7th process separately.

C.PRIORITY NON PREEMPTIVE SCHEDULING ALGORITHM

Algorithm:

1. Get the number of process, burst time and priority.
2. Using for loop i=0 to n-1 do step 1 to 6.
3. If i=0,wait time=0,T[0]=b[0];
4. T[i]=T[i-1]+b[i] and wt[i]=T[i]-b[i].
5. Total waiting time is calculated by adding the waiting time for lack process.
6. Total turnaround time is calculated by adding all total time of each process.

7. Calculate Average waiting time by dividing the total waiting time by total number of process.
8. Calculate Average turnaround time by dividing the total time by the number of process.
9. Display the result.

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
main ()
{
        scan the pids;
        scan the bursts;
        scan the priorities;
        sort the processess in the increasing order of priority;
        for (counter=0 ; not end of process_queue ; counter++) {
                print "dispatching the process : process_queue[counter]" ;
        }
}
```

Sample Output:

PRIORITY SCHEDULING

Process	Burst time	Priority	Waiting time	Turnaround time
P[1]	4	1	0	4
P[2]	5	2	4	9
P[3]	9	3	9	18

GANTT CHART:

```
----------------------
| P1 | P2   | P3 |
----------------------
0     4 9      18
```

Test cases:

Enter three processes with the burst time of your own

Enter four processes P1, P2, P3, and P4 with 6,4,5,2 as the duration of each process and its Priority as 4,2,3,1 respectively.

Enter 7 processes with increasing burst time and display the waiting time of the 7th process separately.

Enter a maximum of 10 processes.

D.ROUND ROBIN PREEMPTIVE SCHEDULING ALGORITHM

Algorithm:

1. Initialize all the structure elements
2. Receive inputs from the user to fill process id,burst time and arrival time.
3. Calculate the waiting time for all the process id.
   i) The waiting time for first instance of a process is
      calculated as: a[i].waittime=count + a[i].arrivt
   ii) The waiting time for the rest of the instances of the process is calculated as:
      a) If the time quantum is greater than the remaining burst time then waiting time is calculated as:
         a[i].waittime=count + tq
      b) Else if the time quantum is greater than the remaining burst time then waiting
         time is calculated as:
         a[i].waittime=count - remaining burst time
4. Calculate the average waiting time and average turnaround time
5. Print the results of the step 4

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
main ()
{
        scan the pids;
        scan the bursts;
        /*
         * assumption is that processess and their bursts are
         * stored in different arrays
         */
        for (counter=0 ;; counter++) {
                /* if burst is 0, need not schedule that process */
                if (burst[counter] == 0 ) {
                continue ;
                }

                /* quantum is assumed as 1 by default*/
                print "dispatching the process : pid[counter]" ;
```

```
                /* refresh the burst */
                burst[count] = burst[count]-1;
                print "remaining burst for pid[counter] : burst[counter];

                /*
                 *if all the pid's bursts are 0,
                 *it means that all processess have been
                 completed */
                for(flush=0; flush < number of processess;
                        flush++) { check if burst[flush] == 0;
                        if all the burst entries are 0, goto finish;

                /*one cycle over,continue from the first
                process*/ if(counter== max number of
                process) {
                        counter=0;
                        print "one of cycle scheduling over, starting the next round";
                }
        }

finish:
print "all processess have been dispatched sccessfully"

}
```

Sample Output:

```
ROUND ROBIN SCHEDULING
Enter number of processes:3
Enter burst time for P[1]:25
Enter burst time for P[2]:16
Enter burst time for P[3]:7
Enter time quantum:15
Process no      Burst time      Wait time   Turnaround time
 P[1]             25             22          47
 P[2]             16             32          48
 P[3]             7              30          37
```

Test cases:

Enter three processes with the burst time of your own

Enter 6 processes with decreasing burst time and display the waiting time of the 6th process separately.

Enter four processes P1, P2, P3, and P4 with 6,4,5,2 as the duration of each process and its Priority as 4,2,3,1 respectively.

Enter 5 processes with increasing burst time and display the waiting time of the 5th process separately.

Enter a maximum of 15 processes.

Pre- Lab Questions

1. What are the different ways for scheduling a job in operating systems?
2. What are the different types of Real-Time Scheduling?
3. What is a long term scheduler & short term schedulers?
4. Define non-preemptive scheduling.
5. Define dispatcher and its functions?
6. Define dispatch latency?
7. Define CPU scheduling.
8. What is turnaround time?
9. Define Preemptive Scheduling.
10. Difference between preemptive and non preemptive scheduling.
11. How does SJF Scheduling algorithm works.
12. Is non-pre-emptive scheduling frequently used in a computer? Why?
13. What type of scheduling is there in RTOS?
14. What is meant by Input Queue?
15. Define Process and what are the states of process?
16. What is meant by Creating a Process?
17. What is means by Resuming a Process?
18. What is meant by Suspending a Process?
19. What is meant by Context Switching?
20. What is meant by PSW?
21. Define Mutual Exclusion.
22. Why is round robin algorithm considered better than first come first served algorithm?
23. What is context switching?

Conclusion

Thus the C program to implement non Preemptive and Preemptive scheduling algorithm was done and the Gantt Chart, turnaround time and waiting time was displayed.

EX.NO:4	Implementation of producer-consumer using Semaphores

Objectives:

To write a C Program for the Implementation of Producer Consumer Algorithm using Semaphore.

Description

- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time.

Learning Outcomes:

After the completion of the experiment, Student will be able to

- Understand the producer consumer algorithm.
- Write solution to the problem using semaphores.

Problem Statement:

The program to implement synchronization between more than one process using producer consumer algorithm.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

Implementation of producer code

```
 void Producer()
{
do
{  //wait until empty > 0
wait(Empty);
 wait(mutex);
add()
signal(mutex);
 signal(Full);
}
while(TRUE);
}
```

Implementation of consumer code

```
void Producer()
{
do
{  //wait until empty > 0
wait(full);
wait(mutex);
consume()
signal(mutex);
signal(empty);
}
while(TRUE);
 }
```

Sample Output:

```
1.Producer
2.Consumer
3.Exit
Enter your choice:1
Producer produces the item 1
```

Test cases:

Enter the choice of producer and check for buffer full, empty and producing the item
Enter the choice of consumer and check for buffer full, empty and producing the item

Pre- Lab Questions:

1. What are semaphores?
2. Mention the categories of semaphores.
3. How would you initialize a semaphore set?
4. What are the two main operations of semaphores?
5. How can you perform a operation on semaphore?
6. List out the importance of message queue?
7. What is semctl()?
8. How would you perform control operations in a semaphore?
9. List the parameters of semaphore control operation.
10. What is the producer-consumer problem in OS algorithm?
11. Is producer-consumer problem deadlock?
12. Which semaphore is used in producer-consumer problem?

Conclusion:

Thus the C program to perform Producer consumer problem using semaphores was performed successfully.

EX.NO:5	Implementation of Dining philosopher's algorithm to demonstrate Process Synchronization

Objectives:

To implement Dining philosopher's algorithm using semaphore.

Description:

1. Initialize the semaphores for each fork to 1 (indicating that they are available).

2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.

3. For each philosopher process, create a separate thread that executes the following code:

- While true:
    - Think for a random amount of time.
    - Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
    - Attempt to acquire the semaphore for the fork to the left.
- If successful, attempt to acquire the semaphore for the fork to the right.
- If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
- If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.

4. Run the philosopher threads concurrently.

Learning Outcomes:

After the completion of the experiment, Student will be able to

- Understand the process synchronization.
- Write solution to the Dining philosopher's algorithm using process synchronization s.

Problem Statement:

The program to implement the achieve synchronization between more than one process.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
process P[i]
 while true do
  {  THINK;
     PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
     EAT;
     PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

Sample Output

1.One can eat at a time

2.Two can eat at a time

3.Exit

Enter your choice: 1

Allow one philosopher to eat at any time

P 3 is granted to eat

P 3 is waiting

P 5 is waiting

P 0 is waiting

Test cases:

Enter the option to execute.

Initially allow one philosopher to eat at any time and the status of all others are reported

Option one Allow two philosopher to eat at any time and the status is reported

Pre-lab questions:

1. Which function is used to create the threads?
2. What is the use of sem_init() function?
3. Why is the sem_wait () function used twice in the philos function?
4. What is the main challenge in the dining philosophers problem?
5. How can the dining philosopher's problem be solved?
6. Can the dining philosopher's problem have multiple solutions?
7. What is the dining philosopher's problem also called?
8. What is process synchronisation?

Conclusion: Thus the C program to implement the dining philosopher's problem was implemented successfully.

EX.NO:6	Simulation of Banker's algorithm for Deadlock Avoidance

Objectives:

To implement of safety algorithm for deadlock avoidance using C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the safe state of processes.
- Learn how deadlocks can be avoided.

Problem Statement:

The program to implement the Safety algorithm for deadlock avoidance.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System :WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required :Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
for(1 to n resources)
   {
   calculate the need matrix using

    need[t][j] = MAX[t][j]-allocated[t][j];

   }
   if(count== no of resources)
   {
    Allocate the process number;

   print allocation matrix using

     avail[j]+=allocated[t][j];
   }
```

Sample Output:

Enter the number of resources : 3
Enter the max instances of each resources

a= 10
b= 7
c= 7

Enter the number of processes : 3

Enter the Allocation matrix
    a b c
P[0] 3 2 1
P[1] 1 1 2
P[2] 4 1 2

Enter the Max matrix
    a b c
P[0] 4 4 4
P[1] 3 4 5
P[2] 5 2 4

< P[2] P[0] P[1] >

Test cases:

Enter the number of resources and maximum instances of each resources.

Enter the number of processes and allocation matrix of resources for each process with respect to instances.

Enter the maximum matrix of resources and check the safe state of processes.

Pre-lab questions:

1. Define Deadlock.
2. What are the necessary conditions for deadlock?
3. Define mutual exclusion.
4. Define safe state.
5. Mention the deadlock prevention mechanisms.
6. What are the deadlock avoidance algorithms?
7. What are the methods to break a deadlock?
8. What are the issues to be addressed to deal with the preemption of deadlock?
9. Differentiate preemption and no-preemption.
10. What is hold and wait?

Conclusion:

Thus the C program to implement the safety algorithm for deadlock avoidance was implemented successfully.

EX.NO:7	Analysis and simulation of Memory Allocation and Management Techniques

A.Memory Allocation Techniques

Objectives:

To implement the memory Allocation concept using C.

Description:

In this the memory is divided in two parts and process is fit into it. The process which is best suited in to it is placed in the particular memory where it suits. We have to check memory partition. If it suits, its status should be changed.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the memory allocation types-MVT,MFT.
- Learn how a process can be allocated to a memory block.

Problem Statement:

The program to implement the memory Allocation strategies using

- Multiprogram Variable Task
- Multiprogram Fixed Task

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System :WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required :Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Algorithm :

1. Get the total memory size.
2. Allocate memory for os.
3. Allocate total memory to the pages.

4. 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++;
   } }
5. Display the wastage of memory.

Sample code

MULTIPROGRAM VARIABLE TASK (MVT)

```
for(i=0;i<n;i++)
{
printf("Enter size of page%d:",i+1);
scanf("%d",&s[i]);
}
tot=tot-m;
for(i=0;i<n;i++)
{
if(tot>=s[i])
{
printf("Allocate page %d\n",i+1);
tot=tot-s[i];
}
else
printf("process p%d is blocked\n",i+1);
}
printf("External Fragmentation is=%d",tot);
```

MULTIPROGRAM FIXED TASK (MFT)

```
for(i=0;i<n;i++)
{
printf("Enter size of page%d:",i+1);
scanf("%d",&s[i]);
}
tot=tot-m;
```

```
for(i=0;i<n;i++)
{
if(tot>=s[i])
{
printf("Allocate page %d\n",i+1);
tot=tot-s[i];
}
else
printf("process p%d is blocked\n",i+1);
}
printf("External Fragmentation is=%d",tot);
```

Sample input:

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

MFT

PROCESS MEMORY ALLOCATED

1 400

2 275

Sample output:

MFT

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 (i Bytes) --	275
Enter memory required for process 2 (i Bytes) --	400
Enter memory required for process 3 (i Bytes) --	290

Enter memory required for process 4 (i Bytes) --	293
Enter memory required for process 5 (i Bytes) --	100
No. of Blocks available in memory -	3

MVT

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

Total Internal Fragmentation is 42

Total External Fragmentation is 100

Conclusion:

Thus the C program to implement the different types of memory allocation schemes.

B.Memory Management Techniques

Objectives:

To implement the memory management concept using C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand the memory management types- Best fit, worst fit and First fit.
- Learn how a process can be fitted into a memory block.

Problem Statement:

The program to implement the memory management strategies.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4
Operating System :WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU
Computers Required :Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Algorithm :

1. Get the segment size, number of process to be allocated and their corresponding sizes.
2. Get the options.
    a. If the option is '1' call best fit function.
    b. If the option is '2' call worst fit function.
    c. If the option is '3' call first fit function; Otherwise exit.
3. For first fit, allocate the process to first possible segment which is free and set the slap as '1'. So that none of process to be allocated to segment which is already allocated and vice versa.
4. For best fit , do the following steps,
    a. Sorts the segments according to their sizes.

b. Allocate the process to the segment which is equal to or slightly greater than the process size and set the flag as the ‘1’ .
c. So that none of the process to be allocated to the segment which is already allocated and vice versa. Stop the program.

5. For worst fit, choose largest memory block and fit the process.

Sample Coding:

Worst Fit

```
Enter number of process, memory blocks and its size.
for(i=1;i<=nb;i++)
{
 //names of the memory blocks to be entered//
}
printf("Enter the size of the files :-\n");
for(i=1 to number of files)
{
//Files size to be entered//
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
//compare file size and memory blocks
if(bf[j]!=1)
{
temp=b[j]-f[i];
if(temp>=0) { ff[i]=j;
 break;
} } }
//fit into the blocks of largest size
frag[i]=temp; bf[ff[i]]=1;
```

Best Fit

```
//Enter number of process, its size, memory blocks and its size.
for(i=1;i<=nf;i++)
{
//Files size to be entered
```

```
}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
// compare file size and memory
 blocks if(bf[j]!=1)
} }
```

First Fit

```
//Enter number of process, its size, memory blocks and its size.
{
//Files size to be entered
}
 for(i=1;i<=nf;i++)
 {
for(j=1;j<=nb;j++)
 {
 if(bf[j]!=1)
{
 // compare file size and memory blocks; fit into the very first memory block.
}
```

Sample Output:

Worst Fit

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

ANSWER

File No	File Size	Block No	Block Size	Fragment
1	1	1	5	4
2	4	3	7	3

Pre-Lab Questions

1. What is the significance of memory allocation?
2. What are the different types of memory management schemes?
3. Mention the concept of first fit.
4. What is best fit?
5. What is worst fit?
6. What is fragmentation?
7. Which memory management scheme is considered to be more effective?
8. Is it necessary that the number of files should be equal to number of memory blocks?
9. What will happen if you could not fit the files into the blocks given?

Conclusion:

Thus the C program to implement the different types of memory management schemes.

EX.NO:8	Implementation of Page Replacement Techniques

Objectives:

To write a C program to implement page replacement algorithm.

a. FIFO

b. LRU

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how the page faults are calculated.
- Know how the pages are replaced in memory using FIFO method and LRU Method.
- Learn about the blocks, frame and how virtual memories are used.

Problem Statement:

The program to implement first in first out page replacement and LRU for replacing the pages in the virtual memory and calculating the number of page faults.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4
Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU
Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

a. FIFO PAGE REPLACEMENT:

Algorithm:

1. Start the program
2. Read the block size and compare with the pages in the block
3. When matches found in the block set hit to one, when the page is not present in the block increment page fault count by one
4. Store the page in the block by getting index of the block through mod function
5. Display the number of page fault count, after reading new page the contents of the block will be displayed

6. End the process.

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
fifo ()
{i =0;
        frame_to_sw
        ap = 0;
        while (i != end of the page list )
        {
                if(reference_string[i] is not in memory)
                {
                        page_fault++;
                }

                frame[frame_to_swap]= reference_string[i];
                frame_to_swap++;
                print "current pages in memory: reference_string[]" ;

                if( frame_to_swap== no of page frames)
                {
                        frame_to_swap=0;
                }
        }}
main ()
{
        scan number of page frames;
        scan number of pages;
        scan reference_string of each page;
        fifo ();
        print "No. of page faults: page_fault "
}
```

Sample Output:

```
Enter the no of elements on reference string10
Enter the reference string
1
2
3
4
5
6
```

7
8
9
2
Enter the no of frames3
1 0 0
1 2 0
1 2 3
4 2 3
4 5 3
4 5 6
7 5 6
7 8 6
7 8 9
2 8 9

No of page fault is: 10

Test cases:

Enter varying reference strings

Input the maximum and minimum frame numbers and compare their page faults

Enter repetitive reference strings.

Consider the page reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 and find how many page faults would occur for the FIFO replacement algorithms assuming three frames

B. LRU PAGE REPLACEMENT:

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

Execution of the Program:

$ cc pgm1.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
fifo ()
{i =0; frame_to_swap = 0;
        while (i != end of the page list )
        {
                if(reference_string[i] is not in memory)
                {
                        page_fault++;
                }
                frame[frame_to_swap]= reference_string[i];
                frame_to_swap++;
                print "current pages in memory: reference_string[]" ;
                if( frame_to_swap== no of page frames)

                {
                        frame_to_swap=0;
                }
        }
}
main ()
{
        scan number of page frames;
        scan number of pages;
        scan reference_string of each page;
        fifo ();
        print "No. of page faults: page_fault "
}
```

Sample Output:

```
Enter no of pages: 10
Enter the reference string: 7 5 9 4 3 7 9 6 2 1
Enter no of frames: 3
    7
    7   5
    7   5   9
    4   5   9
    4   3   9
```

4 3 7
9 3 7
9 6 7
9 6 2
1 6 2

The no of page faults is 10

Test cases:

Enter varying reference strings

Input the maximum and minimum frame numbers and compare their page faults

Enter repetitive reference strings.

Consider the page reference string: 1,5,7,5,6,8,9,4,3,7,4 and find how many page faults would occur for the LRU replacement algorithms assuming three frames

Pre-lab Questions:

1. What is page fault?
2. What is the purpose of page replacement algorithm?
3. State FIFO and LRU.
4. Define optimal page replacement.
5. What are the criteria for best page replacement algorithm?
6. State FIFO page replacement technique.
7. State LRU page replacement technique.
8. State Optimal page replacement technique.
9. What is virtual memory; if OS did not have that concept what are the problems that can arise?
10. Why paging is used?
11. Define Demand Paging, Page fault interrupt, and Trashing?
12. Explain Segmentation with paging?
13. Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs?
14. Explain FIFO Memory Management algorithm with Example

Conclusion:

Thus the C program to implement FIFO and LRU page replacement was implemented successfully and the output was verified.

EX.NO:9	Simulation of Disk Scheduling Algorithms

Objectives:

To simulate disk scheduling algorithms- FCFS, SCAN, SSTF, C-SCAN, LOOK and C – LOOK algorithm.

.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how disk can be scheduled.
- Learn how disk is scheduled using FCFS, SCAN, SSTF, C-SCAN, Look and C-Look methods.

Problem Statement:

The program to implement the following disk scheduling methods.

a) FCFS
b) SSTF
c) SCAN
d) C-SCAN
e) Look and
f) C-Look

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Algorithm

1. Mark the 'head' as the initial position of disk head.
   Let request array represent an array storing indexes of track that have been requested in ascending order of their time of arrival.
2. The corresponding algorithm starts it process of execution

- FCFS- It services the IO requests in the order in which they arrive. There is no starvation in this algorithm, every request is serviced.
- SCAN- the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path,and then it turns backand moves in the reverse direction satisfying requests coming in its path
- SSTF - Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction.
- C-SCAN- the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.
- Look- the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists.
- C-Look- the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.

3. The same process repeats.

a. (First Come First Serve (FCFS))

Sample code

```
for(i=0;i<n;i++)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }
```

a) SSTF

Sample code

```
while(count!=n)
   {
      int min=1000,d,index;
      for(i=0;i<n;i++)
      {
        d=abs(RQ[i]-initial);
        if(min>d)
        {
           min=d;
```

```
            index=i;
          }
                }
    TotalHeadMoment=TotalHeadMoment+min;
    initial=RQ[index];
    // 1000 is for max
    // you can use any number
    RQ[index]=1000;
    count++;
}
```

b) SCAN

Sample code

```
for(i=0;i<n;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(RQ[j]>RQ[j+1])
        {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
        }

    }
}
int index;
for(i=0;i<n;i++)
{
    if(initial<RQ[i])
    {
        index=i;
        break;
    }
}
```

```
// if movement is towards high value
if(move==1)
{
   for(i=index;i<n;i++)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }
   //  last movement for max size
   TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
   initial = size-1;
   for(i=index-1;i>=0;i--)
   {
       TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
       initial=RQ[i];
            }
}
// if movement is towards low value
else
{
   for(i=index-1;i>=0;i--)
   {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }
   //  last movement for min size
   TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
   initial =0;
   for(i=index;i<n;i++)
   {
       TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
       initial=RQ[i];

   }
}
   printf("Total head movement is %d",TotalHeadMoment);
return 0;
```

```
}
```

d.C-SCAN DISK SCHEDULING ALGORITHM

Sample code

```
    if(t[j]>t[j+1])
{
temp=t[j];
t[j]=t[j+1];
t[j+1]=temp;
}
for(i=0;i<=n+2;i++)
if(t[i]==h)
j=i;break;
p=0;
while(t[j]!=tot-1)
{
atr[p]=t[j];
j++;
p++;
}
atr[p]=t[j];
p++;
i=0;
while(p!=(n+3) && t[i]!=t[h])
{
atr[p]=t[i];
i++;
p++;
}
for(j=0;j<n+2;j++)
{
if(atr[j]>atr[j+1])
d[j]=atr[j]-atr[j+1];
else
d[j]=atr[j+1]-atr[j];
sum+=d[j]
```

```
}
```

Look

Sample code

```
for(i=0;i<n;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(RQ[j]>RQ[j+1])
         {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
         }

      }
   }

   int index;
   for(i=0;i<n;i++)
   {
      if(initial<RQ[i])
      {
         index=i;
         break;
      }
   }

   // if movement is towards high value
   if(move==1)
   {
      for(i=index;i<n;i++)
      {
         TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];
      }
```

```
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];

    }
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];

    }
}

printf("Total head movement is %d",TotalHeadMoment);
}
```

e) C-Look

Sample code

```
for(i=0;i<n;i++)
    {
        for( j=0;j<n-i-1;j++)
        {
            if(RQ[j]>RQ[j+1])
            {
                int temp;
```

```
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
        }
    }
}

int index;
for(i=0;i<n;i++)
{
    if(initial<RQ[i])
    {
        index=i;
        break;
    }
}
   // if movement is towards high value
if(move==1)
{
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    for( i=0;i<index;i++)
    {
         TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];

    }
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
```

```
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }

    for(i=n-1;i>=index;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];

    }
}
```

Sample Input:

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Enter the head movement direction for high 1 and for low 0

1

Sample output:

FCFS- Total head movement is 644

SSTF - Total head movement is 236

SCAN- Total head movement is 337

C-SCAN- Total head movement is 382

Look - Total head movement is 299

C-Look- Total head movement is 322

Test cases:

Enter the number of request

Enter the requests sequence

Enter initial head position

Enter the head movement direction for high 1 and for low 0

The Total head movement is calculated and displayed.

Pre-lab Questions:

1. What is the purpose of disk scheduling algorithm?
2. What is seek time?
3. List the ways in which disk can be scheduled?
4. Which algorithm of disk scheduling selects the request with the least seek time from the current head positions?
5. What is track in disk?
6. What is sector in disk?
7. Define Shortest Seek Time First (SSTE)
8. What is rotational latency?
9. What is the advantage of C-SCAN algorithm over SCAN algorithm?
10. Which disk scheduling algorithm has highest rotational latency? Why?

Conclusion:

Thus the C program to implement disk scheduling was implemented successfully and the output was verified.

Ex.No:10	Implementation of File organization Techniques and File Allocation Strategies

A.FILE ORGANIZATION TECHNIQUES

Objectives:

To simulate various file organization technique like single level directory, two level directory, hierarchical and DAG.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Understand how free disk spaces can be allocated.
- Learn how file organization is done in single level and two level directory as well as hierarchical and DAG technique.
- Will be able to compare between different file organization methods.

Problem Statement:

The program to implement the following file organization methods for organizing spaces to the file.

1. Single level directory,
2. Two level directory,
3. Hierarchical and
4. General Graph Directory

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4
Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU
Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Algorithm

1. Read the directory name
2. Read the number of files
3. Read the individual files
4. Declare the root directory
5. Define and display the file

A) SINGLE LEVEL DIRECTORY:

Sample code :

```
if(i==dir.fcnt)
printf("File %s not found",f);
else
dir.fcnt--;
break;
case 3: printf("\nEnter the name of the file -- ");
scanf("%s",f);
for(i=0;i<dir.fcnt;i++)
{
if(strcmp(f, dir.fname[i])==0)
{
printf("File %s is found ", f);
break;
} }
if(i==dir.fcnt)
printf("File %s not found",f);
break;
Case 4: if(dir.fcnt==0)
printf("\nDirectory Empty");
else
{
printf("\nThe Files are -- ");
for(i=0;i<dir.fcnt;i++)
printf("\t%s",dir.fname[i]);
}
```

Sample output:

Enter name of directory -- CSE
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1
Enter the name of the file -- A
1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit Enter your choice – 1

2. TWO LEVEL DIRECTORY

Sample code :

```
void main()
if(i==dcnt)
printf("Directory %s not found",d);
break;
case 3: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter name of the file -- ");
scanf("%s",f); for(k=0;k<dir[i].fcnt;k++)
{
if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is deleted ",f); dir[i].fcnt--;
strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]);
goto jmp;
} }
printf("File %s not found",f);
goto jmp;
} }
printf("Directory %s not found",d);
jmp : break;
case 4: printf("\nEnter name of the directory -- ");
scanf("%s",d);
for(i=0;i<dcnt;i++)
{
if(strcmp(d,dir[i].dname)==0)
{
printf("Enter the name of the file -- ");
scanf("%s",f); for(k=0;k<dir[i].fcnt;k++)
{

if(strcmp(f, dir[i].fname[k])==0)
{
printf("File %s is found ",f); goto jmp1;
} }
printf("File %s not found",f); goto jmp1;
} }
printf("Directory %s ot found",d); jmp1: break; case 5: if(dcnt==0)
printf("\nNo Directory's ");
else
{
printf("\nDirectory\tFiles"); for(i=0;i<dcnt;i++)
```

```
{ printf("\n%s\t\t",dir[i].dname); for(k=0;k<dir[i].fcnt;k++)
printf("\t%s",dir[i].fname[k]);
} }
break;
default:exit(0);
}}
```

Sample output

1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1
Enter name of directory -- DIR1 Directory created
26
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 1
Enter name of directory -- DIR2 Directory created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2
Enter name of the directory – DIR1 Enter name of the file -- A1
File created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice -- 2
Enter name of the directory – DIR1
Enter name of the file -- A2 File created
1. Create Directory 2. Create File 3. Delete File
4. Search File 5. Display 6. Exit Enter your choice – 6

3. HIERARCHICAL DIRECTORY:

Sample code:

```
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i,gap;
if(*root==NULL)
{
(*root)=(node *)malloc(sizeof(node));
printf("Enter name of dir/file(under %s) :",dname);
fflush(stdin);
gets((*root)->name);
printf("enter 1 for Dir/2 forfile :");
scanf("%d",&(*root)->ftype);
(*root)->level=lev;
(*root)->y=50+lev*50;
(*root)->x=x;
(*root)->lx=lx;
(*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
{
```

```
printf("No of sub directories/files(for %s):",(*root)->name); scanf("%d",&(*root)->nc);
if((*root)->nc==0)
gap=rx-lx;
else gap=(rx-lx)/(*root)->nc;
for(i=0;i<(*root)->nc;i++)
create(&((*root)->link[i]),lev+1,(*root)->name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else (*root)->nc=0;
} }
display(node *root)
{
int i;
settextstyle(2,0,4);
settextjustify(1,1);
setfillstyle(1,BLUE);
setcolor(14); if(root!=NULL)
{
for(i=0;i<root->nc;i++)
{
line(root->x,root->y,root->link[i]->x,root->link[i]->y);
}
if(root->ftype==1) bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0); else
fillellipse(root->x,root->y,20,20);
outtextxy(root->x,root->y,root->name); for(i=0;i<root->nc;i++)
{
display(root->link[i]);
} } }
```

Sample output

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

4.GENERAL GRAPH DIRECTORY:

Sample code:

```
void main()
{
int gd=DETECT,gm; root=NULL; clrscr();
create(&root,0,"root",0,639,320); read_links();
clrscr(); initgraph(&gd,&gm,"c:\\tc\\BGI"); draw_link_lines();
display(root); getch(); closegraph();
}
read_links()
{
```

```
int i;
printf("how many links"); scanf("%d",&nofl); for(i=0;i<nofl;i++)
{
printf("File/dir:");
fflush(stdin); gets(L[i].from); printf("user name:"); fflush(stdin); gets(L[i].to);
} }
draw_link_lines()
{
int i,x1,y1,x2,y2; for(i=0;i<nofl;i++)
{
search(root,L[i].from,&x1,&y1); search(root,L[i].to,&x2,&y2);
setcolor(LIGHTGREEN); setlinestyle(3,0,1); line(x1,y1,x2,y2);
setcolor(YELLOW); setlinestyle(0,0,1);
} }
search(node *root,char *s,int *x,int *y)
{
int i; if(root!=NULL)
{
if(strcmpi(root->name,s)==0)
{
*x=root->x;
*y=root->y;
return;
}
else{
for(i=0;i<root->nc;i++)
search(root->link[i],s,x,y);
} } }
create(node **root,int lev,char *dname,int lx,int rx,int x)
{
int i,gap; if(*root==NULL)
{
(*root)=(node *)malloc(sizeof(node));
printf("enter name of dir/file(under %s):",dname);
fflush(stdin);
gets((*root)->name);
printf("enter 1 for dir/ 2 for file:");
scanf("%d",&(*root)->ftype);
(*root)->level=lev; (*root)->y=50+lev*50; (*root)->x=x;
(*root)->lx=lx; (*root)->rx=rx;
for(i=0;i<5;i++)
(*root)->link[i]=NULL;
if((*root)->ftype==1)
```

```
{
printf("no of sub directories /files (for %s):",(*root)->name);
scanf("%d",&(*root)->nc);
if((*root)->nc==0)
gap=rx-lx;
else{
gap=(rx-lx)/(*root)->nc; for(i=0;i<(*root)->nc;i++)
create( & ( (*root)->link[i] ) , lev+1 ,
(*root)->name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else
(*root)->nc=0;
} }  /* displays the constructed tree in graphics mode */ display(node *root)
{
int i; settextstyle(2,0,4); settextjustify(1,1); setfillstyle(1,BLUE); setcolor(14);
if(root
!=NULL)
{
for(i=0;i<root->nc;i++)
{
line(root->x,root->y,root->link[i]->x,root->link[i]->y);
}
if(root->ftype==1) bar3d(root->x-20,root->y-10,root-
>x+20,root->y+10,0,0);
else
fillellipse(root->x,root->y,20,20); outtextxy(root->x,root->y,root-
>name); for(i=0;i<root->nc;i++)
{
display(root->link[i]);
}}}
```

Sample output:

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

No of subdirectories /files (for USER 1): 2

Enter Name of dir/file (under USER1): VB

Test cases:

Enter the directory name

Enter the file count and file name

Process the file organization

Display the corresponding file organization type.

Conclusion:

Thus the C program to implement file organization techniques was implemented successfully and the output was verified.

B.File Allocation Strategies

Objectives:

To implement file allocation on free disk space in a contiguous, linked and indexed manner
in C.

Learning Outcomes:

After the completion of this experiment, student will be able to

- Learn how file allocation is done in contiguous, linked and indexed manner.
- Will be able to compare between contiguous, linked and indexed file allocation methods.

Problem Statement:

The program to implement the following file allocation methods for allocating spaces to the file to be written based on the size of the file.

System and Software tools required:

Software Required: TURBO C version 3 (or) GCC version 3.3.4

Operating System: WINDOWS 2000 / XP / NT OR LINUX OR UBUNTU

Computers Required: Minimum Requirement: Pentium III or Pentium IV with 256 RAM and 40 GB hard disk

Algorithm:

1. Assume no. of blocks in the disk as 20 and all are free.
2. Display the status of disk blocks before allocation.

3. For each file to be allocated:
   a. Get the filename, start address and file length
   b. If start + length > 20, then goto step 2.
   c. Check to see whether any block in the range (start, start + length-1) is allocated. If so, then go to step 2.
   d. Allocate blocks to the file contiguously from start block to start + length – 1.
4. Display directory entries.
5. Display status of disk blocks after allocation
6. Stop

Execution of the Program:

$ cc filename.c //Compiling the Program

$ ./a.out //Executing the program

Sample Coding:

```
main ()
{
        scan total number of blocks;
        print block stats;
        scan filename, length and start_address;

        if (length !> total blocks)
        {
                abort;
        }

        if (any block in therequired range is already allocated)
        {
                abort;
        }
        allocate requested blocks;
        print current block stats and file stats;
}
```

Sample Output:

Disk space before allocation:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

18 19 Enter File name (max 3 char) : ls Enter

start block: 3

Enter no. of blocks: 4

Allocation done

Any more allocation (1. yes / 2. no)? : 1

Enter File name (max 3 char) : cp

Enter start block: 14

Enter no. of blocks: 3

Allocation done

Disk space after allocation:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
mv	mv		ls	ls	ls	ls	ps	ps	ps	tr	tr	tr		cp	cp	cp			

Test cases:

Enter blocks of varying size and check how allocation is done.

Enter larger size files and check whether the disk space overlaps.

Consider the disk spaces as both maximum and minimum and compare the results of both.

Enter the block size as both high and low values and show how the disk space is utilized by comparing.

Pre-lab questions:

1. Define File.
2. State the structure of file.
3. What are the ways in which the file can be accessed?
4. How the disk space is allocated ti the file?
5. State contiguous, linked and indexed allocation.
6. Differentiate cmp command from diff command.
7. What is a reference string?
8. Define seek time and latency time.
9. What are the allocation methods of a disk space?
10. What are the advantages & disadvantages of Contiguous allocation of disk space?
11. What are the advantages of Linked allocation?
12. Define directory?
13. Describe the general directory structure
14. List the different types of directory structures
15. Which of the directory structures is efficient? Why?
16. Which directory structure does not provide user-level isolation and protection?
17. What is the advantage of hierarchical directory structure?

Conclusion:

Thus the C program to implement file allocation on free disk space in a contiguous, linked and indexed manner was implemented successfully and the output was verified.