

Mahatma Gandhi Mission's
COLLEGE OF ENGINEERING AND TECHNOLOGY
AY 2023-2024

Department : Computer Engineering.

Class : SE

Subject : Operating System

Experiment List

| Serial No. | Name of experiment |
|-------------------|--|
| 1. | Study of basic LINUX commands. |
| 2. | Study and implement shell programs in Linux. |
| 3. | Implement any one basic commands of linux like ls, cp, mv and others using kernel APIs. |
| 4. | a. Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call. b. Explore wait and waitpid before termination of process. |
| 5. | a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. b. Write a program to demonstrate the concept of preemptive scheduling algorithms. |
| 6. | a. Write a C program to implement solution of Producer consumer problem through Semaphore. |
| 7. | a. Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm b. Write a program demonstrate the concept of Dining Philosopher's Problem |
| 8 | a. Write a program to demonstrate the concept of MVT and MFT memory management techniques b. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc. |
| 9 | a. Write a program to demonstrate the concept of demand paging for simulation of Virtual Memory implementation b. Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU etc |
| 10 | a. Write a C program to simulate File allocation strategies typically sequential, indexed and linked files b. Write a C program to simulate file organization of multi-level directory structure. c. Write a program in C to do disk scheduling - FCFS, SCAN, C-SCAN |

Experiment No: 01

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: Study of basic LINUX commands.

For eg: (mkdir, chdir, cat, ls, chown, chmod, chgrp, ps etc. system calls: open, read, write, close, getpid, setpid, getuid, getgid, getegid, geteuid. sort, grep, awk, etc.)

Theory:

Basic commands

There are a number of Linux commands which the user should become familiar with from the outset:

passwd :

This command allows you to change your login *password*. You are prompted to enter your current password, and then prompted (twice) to enter your new password.

cd

This command, as in DOS, changes directories. Example: **cd maindir** to move into the **maindir** directory, **cd ..** to move to the directory above, or **cd ~** to move to your root directory.

pwd

This command tells you which directory you are currently working in.

mv

The "move" command is how you rename files. Example: **mv oldfile.txt newfile.txt**

cp

Allows you to copy one or more files. Example: **cp myfile.c backup.c**

rm

Deletes a file. Example: **rm janfiles.***

more

Like **cat** but displays a file one page at a time. Example: **more long_file.txt**

wc

Counts the number of lines, words, and characters in a file. Example: **wc essay.rtf**

tail -n

Displays the last *n* lines of a file. Example: **tail -5 myfile**

head -n

Displays the first *n* lines of a file. Example: **head -5 myfile**

man

The **most important** *Unix* command! It displays the manual pages for a chosen *Unix* command. Example: **man ls**

man -k

Displays all *Unix* commands related to a given keyword. Example: **man -k date** will list all *Unix* commands whose man pages contain a reference to the word date.

date

Shows the current time and date.

mkdir

Creates a new directory, located below the present directory. (Use **pwd** first to check where you are!) Example: **mkdir new_dir**

chdir

Changes to another directory. Example: **chdir /home/user/www** will change the directory you are in to `/home/user/www`. Wildcards are also permitted.

rmdir

Deletes a directory. Example: **rmdir old_dir**

cat

Sends the contents of a file to *stdout* (usually the display screen). Example:
cat index.html

ls

This gives you a listing of all files in a directory. You can't tell which are files and which are directories.

chown

The **chown** command allows you to change the user and/or group ownership of a given file, directory, or symbolic link.

chown [OPTIONS] USER[:GROUP] FILE(s)

- [OPTIONS] – the command can be used with or without additional options.
- [USER] – the username or the numeric user ID of the new owner of a file.
- [:] – use the colon when changing a group of a file.
- [GROUP] – changing the group ownership of a file is optional.
- FILE – the target file.

Superuser permissions are necessary to execute the **chown** command.

chmod

The **chmod** command is used to change the access mode of a file.

`chmod [reference][operator][mode] file..`

The references are used to distinguish the users to whom the permissions apply i.e. they are list of letters that specifies whom to give permissions. The references are represented by one or more of the following letters:

| Reference | Class | Description |
|-----------|-------|-------------|
|-----------|-------|-------------|

| | | |
|---|-------|--------------|
| u | owner | file's owner |
|---|-------|--------------|

| | | |
|---|-------|---|
| g | group | users who are members of the file's group |
|---|-------|---|

| | | |
|---|--------|--|
| o | others | users who are neither the file's owner nor members of the file's group |
|---|--------|--|

| | | |
|---|-----|-------------------------------------|
| a | all | All three of the above, same as ugo |
|---|-----|-------------------------------------|

The operator is used to specify how the modes of a file should be adjusted. The following operators are accepted:

| Operator | Description |
|----------|-------------|
|----------|-------------|

| | |
|---|---|
| + | Adds the specified modes to the specified classes |
|---|---|

| | |
|---|--|
| - | Removes the specified modes from the specified classes |
|---|--|

| | |
|---|--|
| = | The modes specified are to be made the exact modes for the specified classes |
|---|--|

The modes indicate which permissions are to be granted or removed from the specified classes. There are three basic modes which correspond to the basic permissions:

| | |
|---|------------------------------|
| r | Permission to read the file. |
|---|------------------------------|

| | |
|---|---|
| w | Permission to write (or delete) the file. |
|---|---|

| | |
|---|--|
| x | Permission to execute the file, or, in the case of a directory, search it. |
|---|--|

chgrp

chgrp command in Linux is used to change the group ownership of a file or directory.

`chgrp [OPTION]... GROUP FILE...`

`chgrp [OPTION]... -reference=RFILE FILE...`

First we need to have administrator permission to add or delete groups. We can Login as root for this purpose or using *sudo*. In order to add a new group we can use:

`sudo addgroup geeksforgeeks`

Example : To change the group ownership of a file.

`sudo chgrp geeksforgeeks abc.txt`

ps

The ps command is used to view currently running processes on the system. It helps us to determine which process is doing what in our system, how much memory it is using, how much CPU space it occupies, user ID, command name, etc.

`ps [options]`

| Option | Description |
|--------|-------------|
|--------|-------------|

| | |
|----|--|
| -a | Displays all processes on a terminal, with the exception of group leaders. |
|----|--|

- c Displays scheduler data.
- d Displays all processes with the exception of session leaders.
- e Displays all processes.
- f Displays a full listing.
- glist Displays data for the list of group leader IDs.
- j Displays the process group ID and session ID.
- l Displays a long listing
- plist Displays data for the list of process IDs.
- slist Displays data for the list of session leader IDs.
- tlist Displays data for the list of terminals.
- ulist Displays data for the list of usernames.

Conclusion :

Questions:

1) Explain the difference between Linux and Unix ?

2) Give example of Linux os and Unix os?

3) How to create folder on desktop?

4) What is operating system?

5) Give five advantages of Linux?

6) Give five advantages of Unix?

7) What is the use of CP command?

Experiment No. 2

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: To study and implement shell programs in Linux.

Theory : Shell script defined as:

"Shell Script is **series of command** written in **plain text file**. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file."

Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in OS there is special program called Shell. Shell accepts your instruction or commands in English and translate it into computers native binary language.

A shell is a program constructed of shell commands (\$shell, \$path, ls, pwd, mkdir, etc..) Shell is an environment for user interaction. But it is not a part of kernel. Shell is just like as BAT files in MS-DOS. By default, Bash shell is default shell for Linux. Shells are CASE SENSITIVE. Shells allows interaction with kernel. Shells allow one to create functions and pass arguments to them.

Types of shells:

1. Bourne shell.
2. C shell and tc shell.
3. korn shell.

1. **Bourne Shell** : The Bourne shell was one of the major shells used in early versions and became a de facto standard. It was written by Stephen Bourne at Bell Labs. Every Unix-like system has at least one shell compatible with the Bourne shell. The Bourne shell program name is "sh" and it is typically located in the file system hierarchy at /bin/sh.
2. **C shell and tc shell**: The **C shell** was developed by **Bill Joy** for the **Berkeley Software Distribution**. Its syntax is modelled after the **C** programming language. It is used primarily for interactive terminal use, but less frequently for scripting and operating system control. **C** shell has many interactive commands. Developers have written large parts of the Linux operating system in the C and C++ languages. Using C syntax as a model, Bill Joy at Berkeley University developed the "C-shell," csh, in 1978. Ken Greer, working at Carnegie-Mellon University, took csh concepts a step forward with a new shell, tcsh, which Linux systems now offer. Tcsh fixed problems in csh and added command completion, in which the shell makes educated "guesses" as you type, based on your system's directory structure and files. Tcsh does not run bash scripts, as the two have substantial differences.
3. **Korn Shell** : David Korn developed the Korn shell, or ksh, about the time tcsh was introduced. Ksh is compatible with sh and bash. Ksh improves on the Bourne shell by

adding floating-point arithmetic, job control, command aliasing and command completion. AT&T held proprietary rights to ksh until 2000, when it became open source.

Steps to write shell script:

(1) Use any editor like vi or gedit to write shell script.

(2) After writing shell script set execute permission for your script as follows

syntax:

chmod permission your-script-name

Examples:

\$ chmod +x your-script-name

\$ chmod 755 your-script-name

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

(3) Execute your script as

syntax:

bash your-script-name

sh your-script-name

./your-script-name

Examples:

\$ bash bar

\$ sh bar

\$./bar

Write shell scripts to do the following:

- a. Display OS version, release number, kernel version

```
#!/bin/sh
Cat /etc/os-release
Lsb_release -a
Hostnamectl
Uname -r
```

- b. Display top 10 processes in descending order

```
#!/bin/sh
$ps -eo size,command --sort -size |head
```

- c. Display processes with highest memory usage.

```
#!/bin/sh
Top -b -o +%MEM | head -17
```

- d. Display current logged in user and log name.

w command : Show who is logged on and what they are doing on Linux

who command : Display information about Linux users who are currently logged in

whoami command : Find out who you are currently logged in as on Linux

id command : See user and group information for the specified USER account on Linux

- e. Display current shell, home directory, operating system type, current path setting, current working directory

```
#!/bin/sh
echo "$SHELL"
echo "$HOME"
echo "$PATH"
echo "$pwd"
```

Conclusion:

Questions:

1) What are the types of Shells?

2) What is Shell?

3) Explain Korn shell?

4) What are the steps to write shell script?

5) What is the use of “#!/bin/bash” ?

6) What is Shell Script and why it is required?

7) Write A Shell Script To Get The Current Date, Time, Username And Current Working Directory ?

Experiment No: 3

Name of Student: _____

Roll NO: _____ **Date of Performance:** _____

Staff Signature: _____ **Marks:** _____

Experiment Name: Implement any one basic commands of linux like ls, cp, mv and others using kernel APIs

Theory:

Linux

Linux has gained in popularity over the years due it being open source hence, based on a UNIX like design, and ported to more platforms compared to other competing operating systems. It is an operating system, as indicated, that resembles a UNIX OS – a stable multi-user multi-tasking operating system, and that has been assembled as a free and open-source software for development and distribution. Meaning that any individual or company has the permission to use, imitate, study and alter the Linux operating system in any way they desire.

Kernel space and Userspace

Kernel Space: the kernel is found in an elevated system state, which includes a protected memory space and full access to the device's hardware. This system state and memory space is altogether referred to as kernel-space. Within kernel space the core access to the hardware and system services are managed and provided as a service to the rest of the system.

User Space: the user's applications are carried out in the user-space, where they can reach a subset of the machine's available resources via kernel system calls. By using the core services provided the kernel, a user level application can be created like a game or office productivity software for example.

What makes the Linux Kernel different from other Classic Unix Kernels?

Significant differences exist between the Linux kernel and the Classic Unix kernels; as listed below:

1. Linux supports dynamic loading of kernel modules.
2. The Linux kernel is preemptive.
3. Linux has a symmetrical multiprocessor support.
4. Linux is free due to its open software nature.
5. Linux ignores some standard Unix features that the kernel developers call "poorly designed."
6. Linux provides an object-oriented device model with device classes, hot-pluggable events, and a user-space device file-system
7. The Linux kernel fails to differentiate between threads and normal processes.

Components of the Linux Kernel

A kernel is simply a resource manager; the resource being managed may be a process, memory or hardware device. It manages and arbitrates access to the resource between multiple competing users. The Linux kernel exists in the kernel space, below the userspace, which is where the user's applications are executed. For the user space to communicate with the kernel space, a GNU C Library is incorporated which provides a forum for the system call interface to connect to the kernel space and allow transition back to the userspace.

The architectural perspective of the Linux kernel consists of: System call interface, Process Management, the Virtual File system, Memory Management, Network Stack, Architecture and the Device Drivers.

1. System call interface; is a thin layer that is used to undertake function calls from user space into the kernel. This interface may be architecture dependent
2. Process management; is mainly there to execute the processes. These are referred to as the thread in a kernel and are representing an individual virtualization of the particular processor
3. Memory management; memory is managed in what are known as pages for efficiency. Linux includes the methods in which to manage the available memory as well as the hardware mechanisms for physical and virtual mappings. Swap space is also provided
4. Virtual file system; it provides a standard interface abstraction for the file systems. It provides a switching layer between the system call interface and the file systems supported by the kernel.
5. Network stack; is designed as a layered architecture modeled after the particular protocols.
6. Device drivers; a significant part of the source code in the Linux kernel is found in the device drivers that make a particular hardware device usable.
7. Architecture-dependent code; those elements that depend on the architecture on which they run, hence must consider the architectural design for normal operation and efficiency.

Interfaces

System calls and Interrupts

Applications pass information to the kernel through system calls. A library contains functions that the applications work with. The libraries then, through the system call interface, instruct the kernel to perform a task that the application wants.

Interrupts offer a way through which the Linux kernel manages the systems' hardware. If hardware has to communicate with a system, an interrupt on the processor does the trick, and this is passed on to the Linux kernel.

a) ls

```
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/param.h>
#include <stdio.h>
#define FALSE 0
#define TRUE 1
extern int alphasort();
```

```

char pathname[MAXPATHLEN];
main() {
int count,i;
struct dirent **files;
int file_select();
if (getwd(pathname) == NULL )
{
printf("Error getting pathn");
exit(0);
}
printf("Current Working Directory = %s\n",pathname);
count = scandir(pathname, &files, file_select, alphasort);
if (count <= 0)
{
printf("No files in this directory\n");
exit(0);
}
printf("Number of files = %d\n",count);
for (i=1;i<count 1; i)
printf("%s \n",files[i-1]->d_name);
}
int file_select(struct direct *entry)
{
if ((strcmp(entry->d_name, ".") == 0) ||(strcmp(entry->d_name, "..") == 0))
return (FALSE);
else
return (TRUE);
}

```

Output:

```

Student@ubuntu:~$ gcc list.c
Student@ubuntu:~$ ./a.out
Current working directory=/home/student/
Number of files=57

```

b) cp

```

#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
void main()
{
int fd1,fd2;
char buf[60];
fd1=open("f2",O_RDWR);
fd2=open("f6",O_RDWR);
read(fd1,buf,sizeof(buf));

```

```
write(fd2,buf,sizeof(buf));
close(fd1);
close(fd2);
}
```

c) mv

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
main( int argc,char *argv[] )
{
int i,fd1,fd2;
char *file1,*file2,buf[2];
file1=argv[1];
file2=argv[2];
printf("file1=%s file2=%s",file1,file2);
fd1=open(file1,O_RDONLY,0777);
fd2=creat(file2,0777);
while(i=read(fd1,buf,1)>0)
write(fd2,buf,1);
remove(file1);
close(fd1);
close(fd2);
}
```

Output:

```
student@ubuntu:~$gcc -o mvp.out mvp.c
student@ubuntu:~$cat > ff
hello
hai
student@ubuntu:~$./mvp.out ff ff1
student@ubuntu:~$cat ff
cat:ff:No such file or directory
student@ubuntu:~$cat ff1
hello
hai
```

Conclusion:

Questions:

1) What are APIs?

2) What are the different types of APIs?

3) Give examples for each of the API types.

4) What is difference between API and function call?

5) Why use APIs rather than native system calls?

Experiment No: 4

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: Linux Process

4. a. Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.
4.b. Explore wait and waitpid before termination of process.

Theory:

Fork system call is used for creating a new process, which is called *child process*, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

Both getppid() and getpid() are inbuilt functions defined in **unistd.h** library.

getppid() : returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of the getppid function call, this function returns the process ID of the parent process. Otherwise, this function returns a value of 1 which is the process id for **init** process.

Syntax:

pid_t getppid(void);

Return type: getppid() returns the process ID of the parent of the current process. It never throws any error therefore is always successful.

getpid() : returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

Syntax:

pid_t getpid(void);

Return type: getpid() returns the process ID of the current process. It never throws any error therefore is always successful.

wait() API()

- The wait() function suspends the execution of the calling process until one of its child terminates or was stopped by a signal or resumed by a signal.

- In the case of `fork()` API, a child process is created which has its life cycle and runs independently.
- If the parent process happens to terminate before the child process, the child process becomes orphan and gets re-parented by the Init process.
- If the child process terminates and the parent process does not remove its entry is still there in the process control block(PCB), it is considered as a Zombie process.
- Using `wait()`, parents process can get the exit status of the child process and can remove its entry and can avoid zombie process.
- If the parent process calls `wait()` API, and the child has already terminated it returns immediately without blocking the caller, and the return value is -1.
- If the parent process calls `wait()` and none of the child processes has terminated, the parent process is blocked indefinitely. Thus `wait()` API is a blocking call.
- On success, it returns the process ID(PID) of the terminated child.

Syntax:

- `pid_t wait (int *status);`

Return Value:

- It returns -1 in case of failure that is there is no child process.
- It returns the PID of the child terminated if there is a child and anyone has terminated.
- If non of the child has terminated, return nothing and the caller is blocked.

`waitpid()`

- The `waitpid()` API suspends the execution of the caller process until a child specified by PID argument has either terminated or signaled to stop or resume.
- The call `wait()` is equivalent to `waitpid (-1, &status, 0)`.
- By default, `waitpid()` waits only for the terminated child, not for the signaled child but this can be changed by the use of options.
- The most important option is `WNOHANG`, which makes `waitpid()` a non-blocking call.

Syntax:

```
int waitpid( int pid, int * stat_var, int options);
```

Return Value:

- It returns -1 in the case of failure that is there is no child process.
- It returns PID of the child terminated on success and
- Return 0 if the child exists but not terminated or stopped(used with `WNOHANG`)

a)

```

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

int main(void) {
    pid_t pid = fork();

    if(pid == 0) {
        printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
        exit(EXIT_SUCCESS);
    }
    else if(pid > 0) {
        printf("Parent => PID: %d\n", getpid());
        printf("Waiting for child process to finish.\n");
        wait(NULL);
        printf("Child process finished.\n");
    }
    else {
        printf("Unable to create child process.\n");
    }

    return EXIT_SUCCESS;
}

```

b)

```

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

int main(void) {

    pid_t childPid; // the child process that the execution will soon run inside of.
    childPid = fork();

    if(childPid == 0) // fork succeeded
    {
        // Do something
        exit(0);
    }

    else if(childPid < 0) // fork failed
    {
        // log the error
    }
}

```

```

}

else // Main (parent) process after fork succeeds
{
    int returnStatus;
    waitpid(childPid, &returnStatus, 0); // Parent process waits here for child to terminate.

    if (returnStatus == 0) // Verify child process terminated without error.
    {
        printf("The child process terminated normally.");
    }

    if (returnStatus == 1)
    {
        printf("The child process terminated with an error!.");
    }
}

```

Conclusion:

Questions:

1) What is a system call?

2) Explain fork() system call.

3) What is the difference between wait() and waitpid() system call?

4) What does getpid() and getppid() system call do?

Experiment No: 5

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name : To implement Process scheduling algorithms

Theory :

Process Scheduling Mechanisms:

A multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time-multiplexing. Part of the reason for using multiprogramming is that the operating system itself is implemented as one or more processes, so there must be a way for the operating system and application processes to share the CPU. Another main reason is the need for processes to perform I/O operations in the normal course of computation. Since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions, multiprogramming systems allocate the CPU to another process whenever a process invokes an I/O operation

1. The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
2. Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.
3. Scheduling Queues
Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

Goals for Scheduling

Make sure your scheduling strategy is good enough with the following criteria:

Utilization/Efficiency: keep the CPU busy 100% of the time with useful work

- **Throughput:** Maximize the number of jobs processed per hour.
- **Turnaround time:** From the time of submission to the time of completion, minimize the time batch users must wait for output
- **Waiting time:** Sum of times spent in ready queue - Minimize this
- **Response Time:** Time from submission till the first response is produced, minimize response time for interactive users

There are four major scheduling algorithms here which are following

- First Come First Serve (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Round Robin(RR) Scheduling
- Shortest Remaining Time First (SSTF)

1. First Come First Serve (FCFS):

FCFS also termed as First-In-First-Out i.e. allocate the CPU in order in which the process arrive. When the CPU is free, it is allowed to process, which is occupying the front of the queue. Once this process goes into running state, its PCB is removed from the queue. This algorithm is non-preemptive.

Advantage:

1. Suitable for batch system.
2. It is simple to understand and code.

Disadvantage:

1. Waiting time can be large if short request wait behind the long process.
2. It is not suitable for time sharing system where it is important that each user should get the CPU for equal amount of time interval.

Example:

| Process | Arrival Time | Execute Time | Service Time |
|---------|--------------|--------------|--------------|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



Wait time of each process is following :

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

Average Wait Time: $(0+4+6+13) / 4 = 5.55$

2. Shortest Job First (SJF):

Shortest job next (SJN), also known as Shortest Job First (SJF) or Shortest Process Next (SPN), is a scheduling process that selects the waiting process with the smallest execution time to execute next. SJN is a non-pre-emptive algorithm. Shortest time remaining is a pre-emptive variant of SJN.

Shortest job next is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added. Highest response ratio next is similar but provides a solution to this problem.

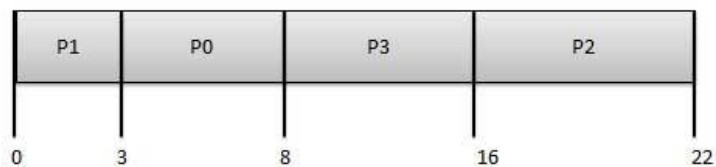
Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution. While it is not possible to perfectly predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.

Shortest job next can be effectively used with interactive processes which generally follow a pattern of alternating between waiting for a command and executing it. If the execution burst of a process is regarded as a separate "job", past behaviour can indicate which process to run next, based on an estimate of its running time.

Shortest job next is used in specialized environments where accurate estimates of running time are available. Estimating the running time of queued processes is sometimes done using a technique called aging.

Example:

| Process | Arrival Time | Execute Time | Service Time |
|---------|--------------|--------------|--------------|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 3 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



Wait time of each process is following :

| Process | Wait Time : Service Time - Arrival Time |
|---------|---|
| P0 | $3 - 0 = 3$ |
| P1 | $0 - 0 = 0$ |
| P2 | $16 - 2 = 14$ |
| P3 | $8 - 3 = 5$ |

Average Wait Time: $(3+0+14+5) / 4 = 5.50$

3. Round Robin Scheduling:

Round robin is the scheduling algorithm used by the CPU during execution of the process. Round robin is designed specifically for time sharing systems. It is similar to first come first serve scheduling algorithm but the pre-emption is the added functionality to switch between the processes.

A small unit of time also known as time slice or quantum is set/defined. The ready queue works like circular queue. All processes in this algorithm are kept in the circular queue also known as ready queue. Each New process is added to the tail of the ready/circular queue.

By using this algorithm, CPU makes sure, time slices (any natural number) are assigned to each process in equal portions and in circular order, dealing with all process without any priority.

It is also known as cyclic executive.

The main advantage of round robin algorithm over first come first serve algorithm is that it is starvation free. Every process will be executed by CPU for fixed interval of time (which is set as time slice). So in this way no process left waiting for its turn to be executed by the CPU.

Round robin algorithm is simple and easy to implement. The name round robin comes from the principle known as round robin in which every person takes equal share of something in turn.

Pseudo Code:

- * CPU scheduler picks the process from the circular/ready queue, set a timer to interrupt it after 1 time slice / quantum and dispatches it.

- * If process has burst time less than 1 time slice/quantum

- > Process will leave the CPU after the completion
- > CPU will proceed with the next process in the ready queue / circular queue.

else If process has burst time longer than 1 time slice/quantum

- > Timer will be stopped. It cause interruption to the OS.
- > Executed process is then placed at the tail of the circular / ready queue by applying the context switch
- > CPU scheduler then proceeds by selecting the next process in the ready queue.

Here, User can calculate the average turnaround time and average waiting time along with the starting and finishing time of each process

Turnaround time : Its the total time taken by the process between starting and the completion

Waiting time : Its the time for which process is ready to run but not executed by CPU scheduler

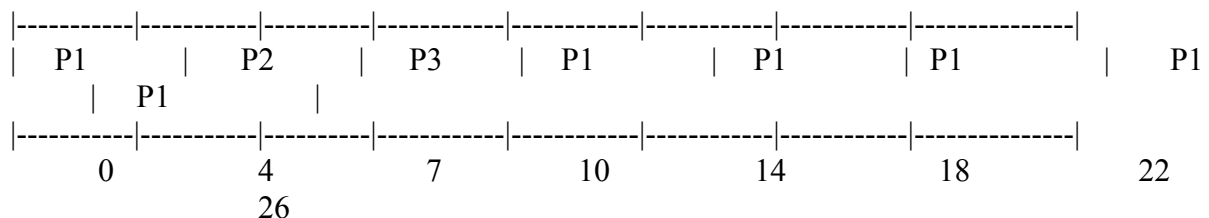
Example:

We have three processes:

| | Burst time | Waiting time | Turnaround time |
|----|------------|--------------|-----------------|
| P1 | 24 | 6 | 30 |
| P2 | 3 | 4 | 7 |
| P3 | 3 | 7 | 10 |

So here we can see the turnaround time for the process 1 is 30 while 7 and 10 for 2nd and 3rd process. A Gantt chart is a chart which shows the start and finish times of all the processes .

Gantt chart for the round robin algorithm is



To write the program to implement CPU scheduling algorithm for first come first serve scheduling.

Program:

```
#include<stdio.h>
struct process
{
int pid; int bt; int wt,tt;
}p[10];

int main()
{
int i,n,totwt,tottt,avg1,avg2;

printf("enter the no of process \n");
scanf("%d",&n);
for(i=1;i<=n;i++)
{

p[i].pid=i;

printf("enter the burst time n");
scanf("%d",&p[i].bt);

}

p[1].wt=0; p[1].tt=p[1].bt+p[1].wt; i=2;
```

```

while(i<=n)
{
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
i++;
}
i=1;
totwt=totwt=0;
printf("\n processid \t bt\t wt\t tt\n");
while(i<=n){
printf("\n\t%d \t%d \t%d \t%d",p[i].pid,p[i].bt,p[i].wt,p[i].tt);
totwt=p[i].wt+totwt;
totwt=p[i].tt+totwt;
i++;
}
avg1=totwt/n; avg2=totwt/n;
printf("\navg1=%d \t avg2=%d\t",avg1,avg2);
return 0;
}
Output:

```

```

enter the no of process
3
enter the burst time n5
enter the burst time n6
enter the burst time n7

processid      bt      wt      tt

      1        5        0        5
      2        6        5       11
      3        7       11       18
avg1=5   avg2=11

```

To write a program to implement cpu scheduling for Round Robin Scheduling.

ALGORITHM:

1. Get the number of process and their burst time.
2. Initialize the array for Round Robin circular queue as '0'.
3. The burst time of each process is divided and the quotients are stored on the round Robin array.
4. According to the array value the waiting time for each process and the average time are calculated as line the other scheduling.
5. The waiting time for each process and average times are displayed.
6. Stop the program.

Program:

```
#include<stdio.h>
struct process
{
int pid,bt,tt,wt;
};
int main()
{
struct process x[10],p[30];
int i,j,k,tot=0,m,n;
float wtime=0.0,tottime=0.0,a1,a2;
printf("\nEnter the number of process:\t");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
x[i].pid=i;
printf("\nEnter the Burst Time:\t");
scanf("%d",&x[i].bt);
tot=tot+x[i].bt;
}
printf("\nTotal Burst Time:\t%d",tot);
p[0].tt=0;
k=1;
printf("\nEnter the Time Slice:\t"); scanf("%d",&m);
for(j=1;j<=tot;j++)
{
for(i=1;i<=n;i++)
{
if(x[i].bt !=0)
{
p[k].pid=i;
if(x[i].bt-m<0)
{
p[k].wt=p[k-1].tt;
p[k].bt=x[i].bt;
p[k].tt=p[k].wt+x[i].bt;
x[i].bt=0; k++;
}
else
{
p[k].wt=p[k-1].tt;
p[k].tt=p[k].wt+m;
x[i].bt=x[i].bt-m; k++;
}
}
}
}
```

```

}
printf("\nProcess id \tw\t \ttt");
for(i=1;i<k;i++)
{
printf("\n\t%d \t%d \t%d",p[i].pid,p[i].wt,p[i].tt);
wtime=wtime+p[i].wt;
tottime=tottime+p[i].tt;
a1=wtime/n;
a2=tottime/n;
}
printf("\n\nAverage Waiting Time:\t%f",a1);
printf("\n\nAverage TurnAround Time:\t%f",a2);
return 0;
}

```

```

Enter the number of process:    3

Enter the Burst Time:    3

Enter the Burst Time:    5

Enter the Burst Time:    4

Total Burst Time:    12
Enter the Time Slice:    2

Process id      wt      tt
      1          0       2
      2          2       4
      3          4       6
      1          6       7
      2          7       9
      3          9      11
      2         11      12

Average Waiting Time:    13.000000

Average TurnAround Time:    17.000000

...Program finished with exit code 0
Press ENTER to exit console.

```

b) Write a program to demonstrate the concept of preemptive scheduling algorithms

Pre-emptive priority

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

```
struct process
```

```
{
```

```
    char process_name;
```

```
    int arrival_time, burst_time, ct, waiting_time, turnaround_time, priority;
```

```
    int status;
```

```
}process_queue[10];
```

```
int limit;
```

```
void Arrival_Time_Sorting()
```

```

{
    struct process temp;
    int i, j;
    for(i = 0; i < limit - 1; i++)
    {
        for(j = i + 1; j < limit; j++)
        {
            if(process_queue[i].arrival_time > process_queue[j].arrival_time)
            {
                temp = process_queue[i];
                process_queue[i] = process_queue[j];
                process_queue[j] = temp;
            }
        }
    }
}

void main()
{
    int i, time = 0, burst_time = 0, largest;
    char c;
    float wait_time = 0, turnaround_time = 0, average_waiting_time,
average_turnaround_time;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &limit);
    for(i = 0, c = 'A'; i < limit; i++, c++)
    {
        process_queue[i].process_name = c;
        printf("\nEnter Details For Process[%C]:\n", process_queue[i].process_name);
        printf("Enter Arrival Time:\t");
        scanf("%d", &process_queue[i].arrival_time );
        printf("Enter Burst Time:\t");
        scanf("%d", &process_queue[i].burst_time);
        printf("Enter Priority:\t");
        scanf("%d", &process_queue[i].priority);
        process_queue[i].status = 0;
        burst_time = burst_time + process_queue[i].burst_time;
    }
    Arrival_Time_Sorting();
    process_queue[9].priority = -9999;
    printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time");
    for(time = process_queue[0].arrival_time; time < burst_time;)
    {
        largest = 9;
        for(i = 0; i < limit; i++)
        {
            if(process_queue[i].arrival_time <= time && process_queue[i].status != 1 &&
process_queue[i].priority > process_queue[largest].priority)
            {
                largest = i;
            }
        }
    }
}

```

```

    }
}
time = time + process_queue[largest].burst_time;
process_queue[largest].ct = time;
process_queue[largest].waiting_time = process_queue[largest].ct -
process_queue[largest].arrival_time - process_queue[largest].burst_time;
process_queue[largest].turnaround_time = process_queue[largest].ct -
process_queue[largest].arrival_time;
process_queue[largest].status = 1;
wait_time = wait_time + process_queue[largest].waiting_time;
turnaround_time = turnaround_time + process_queue[largest].turnaround_time;
printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d", process_queue[largest].process_name,
process_queue[largest].arrival_time, process_queue[largest].burst_time,
process_queue[largest].priority, process_queue[largest].waiting_time);
}
average_waiting_time = wait_time / limit;
average_turnaround_time = turnaround_time / limit;
printf("\n\nAverage waiting time:\t%f\n", average_waiting_time);
printf("Average Turnaround Time:\t%f\n", average_turnaround_time);
}

```

```

Terminal File Edit View Search Terminal Help
tushar@tusharsoni:~/Desktop$ gcc test.c
tushar@tusharsoni:~/Desktop$ ./a.out

Enter Total Number of Processes: 3

Enter Details For Process[A]:
Enter Arrival Time: 1
Enter Burst Time: 23
Enter Priority: 2

Enter Details For Process[B]:
Enter Arrival Time: 2
Enter Burst Time: 54
Enter Priority: 1

Enter Details For Process[C]:
Enter Arrival Time: 3
Enter Burst Time: 12
Enter Priority: 3

Process Name  Arrival Time  Burst Time  Priority  Waiting Time
A             1             23          2         0
C             3             12          3         21
B             2             54          1         34

Average waiting time: 18.333334
Average Turnaround Time: 48.000000
tushar@tusharsoni:~/Desktop$

```

Pre-emptive SJF

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

```

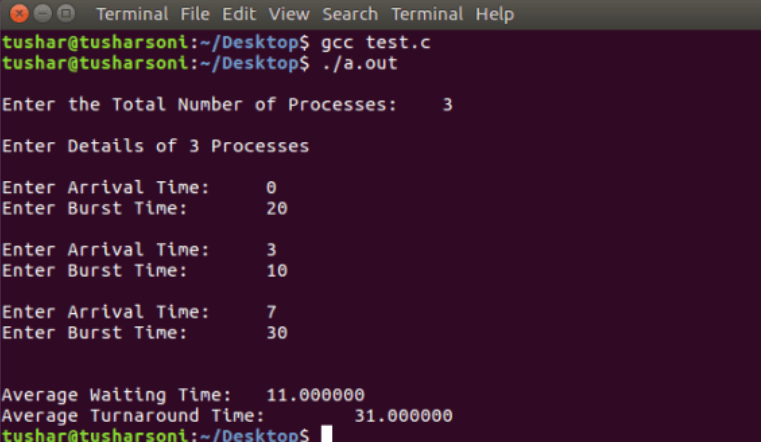
int main()
{
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, limit;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;
    printf("\nEnter the Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Details of %d Processes\n", limit);
    for(i = 0; i < limit; i++)

```

```

{
    printf("\nEnter Arrival Time:\t");
    scanf("%d", &arrival_time[i]);
    printf("Enter Burst Time:\t");
    scanf("%d", &burst_time[i]);
    temp[i] = burst_time[i];
}
burst_time[9] = 9999;
for(time = 0; count != limit; time++)
{
    smallest = 9;
    for(i = 0; i < limit; i++)
    {
        if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest] &&
burst_time[i] > 0)
        {
            smallest = i;
        }
    }
    burst_time[smallest]--;
    if(burst_time[smallest] == 0)
    {
        count++;
        end = time + 1;
        wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
        turnaround_time = turnaround_time + end - arrival_time[smallest];
    }
}
average_waiting_time = wait_time / limit;
average_turnaround_time = turnaround_time / limit;
printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
printf("Average Turnaround Time:\t%lf\n", average_turnaround_time);
return 0;
}

```



```

Terminal File Edit View Search Terminal Help
tushar@tusharsoni:~/Desktop$ gcc test.c
tushar@tusharsoni:~/Desktop$ ./a.out
Enter the Total Number of Processes: 3
Enter Details of 3 Processes
Enter Arrival Time: 0
Enter Burst Time: 20
Enter Arrival Time: 3
Enter Burst Time: 10
Enter Arrival Time: 7
Enter Burst Time: 30
Average Waiting Time: 11.000000
Average Turnaround Time: 31.000000
tushar@tusharsoni:~/Desktop$

```

Conclusion:

Questions:

1) What are different criteria for CPU Scheduling?

2) What are the different CPU Scheduling Algorithms?

3) What is difference between preemptive and non-preemptive algorithms?

4) What is the drawback of Shortest job first scheduling algorithm?

5) Why aging technique is used?

Experiment No. 6

Name of Student: _____

Roll NO: _____ **Date of Performance:** _____

Staff Signature: _____ **Marks:** _____

Experiment Name: Study and simulate producer-consumer problem using Semaphores

AIM: To Write a C program to simulate producer-consumer problem using semaphores.

Theory: Producer consumer problem is a synchronization problem. There is a fixed size buffer where the producer produces items and that is consumed by a consumer process. One solution to the producer consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Program:

```
#include<stdio.h>
void main()
{
int buffer[10], bufsize, in, out, produce, consume, choice=0;
in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice:   =");
scanf("%d", &choice);
switch(choice)
{
case 1: if((in+1)%bufsize==out)
printf("\nBuffer is Full");
else
{
printf("\nEnter the value: ");
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
}
break;
case 2: if(in == out)
```

```

printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
break;
}
}
}
}

```

Output:

```

1. Produce      2. Consume      3. Exit
Enter your choice:    =1

Enter the value: 100

1. Produce      2. Consume      3. Exit
Enter your choice:    =1

Enter the value: 400

1. Produce      2. Consume      3. Exit
Enter your choice:    =2

The consumed value is 100
1. Produce      2. Consume      3. Exit
Enter your choice:    =2

The consumed value is 400
1. Produce      2. Consume      3. Exit
Enter your choice:    =2

Buffer is Empty

```

Conclusion:

Questions:

1. Differentiate between a monitor, semaphore and a binary semaphore?

2. Define Semaphore.

3. Write Test and set method.

4. Define critical Section.

5. What are three properties to satisfy to overcome critical section problem.

Experiment No. 7

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: 7.a To study and implement deadlock avoidance using Banker's algorithm.

Theory :

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The algorithm was developed in the design process for the operating system and originally described (in Dutch) in EWD108. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Banker's Algorithm is a deadlock avoidance algorithm that checks for safe or unsafe state of a System after allocating resources to a process.

When a new process enters into system, it must declare maximum no. of instances of each resource that it may need. After requesting operating system run banker's algorithm to check whether after allocating requested resources, system goes into deadlock state or not. If yes then it will deny the request of resources made by process else it allocates resources to that process.

Safe or Unsafe State:- A system is in Safe state if its all process finish its execution or if any process is unable to acquire its all requested resources then system will be in Unsafe state.

Following **Data structures** are used to implement the Banker's Algorithm:

Let '**n**' be the number of processes in the system and '**m**' be the number of resources types.

Available:

- It is a 1-d array of size '**m**' indicating the number of available resources of each type. Available[j] = k means there are '**k**' instances of resource type **R_j**

Max:

- It is a 2-d array of size '**n*m**' that defines the maximum demand of each process in a system.
- Max[i, j] = k means process **P_i** may request at most '**k**' instances of resource type **R_j**.

Allocation:

- It is a 2-d array of size ' $n \times m$ ' that defines the number of resources of each type currently allocated to each process.
- $\text{Allocation}[i, j] = k$ means process P_i is currently allocated ' k ' instances of resource type R_j

Need:

- It is a 2-d array of size ' $n \times m$ ' that indicates the remaining resource need of each process.
- $\text{Need}[i, j] = k$ means process P_i currently need ' k ' instances of resource type R_j for its execution.
- $\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$

Allocation_i specifies the resources currently allocated to process P_i and Need_i specifies the additional resources that process P_i may still request to complete its task.

Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.

Initialize: Work = Available

Finish[i] = false; for $i=1, 2, 3, 4, \dots, n$

2) Find an i such that both

a) Finish[i] = false

b) $\text{Need}_i \leq \text{Work}$

if no such i exists goto step (4)

3) $\text{Work} = \text{Work} + \text{Allocation}[i]$

Finish[i] = true

goto step (2)

4) if Finish[i] = true for all i

then the system is in a safe state

Program:

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

```
int main()
```

```
{
```

```
    // P0, P1, P2, P3, P4 are the Process names here
```

```
    int n, m, i, j, k;
```

```
    n = 5; // Number of processes
```

```
    m = 3; // Number of resources
```

```
    int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
```

```
                      { 2, 0, 0 }, // P1
```

```
                      { 3, 0, 2 }, // P2
```

```
                      { 2, 1, 1 }, // P3
```

```
                      { 0, 0, 2 } }; // P4
```

```

int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                  { 3, 2, 2 }, // P1
                  { 9, 0, 2 }, // P2
                  { 2, 2, 2 }, // P3
                  { 4, 3, 3 } }; // P4

```

```

int avail[3] = { 3, 3, 2 }; // Available Resources

```

```

int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
    f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
    for (i = 0; i < n; i++) {
        if (f[i] == 0) {

            int flag = 0;
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]){
                    flag = 1;
                    break;
                }
            }

            if (flag == 0) {
                ans[ind++] = i;
                for (y = 0; y < m; y++)
                    avail[y] += alloc[i][y];
                f[i] = 1;
            }
        }
    }
}

```

```

printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
    printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);

```

```

return (0);

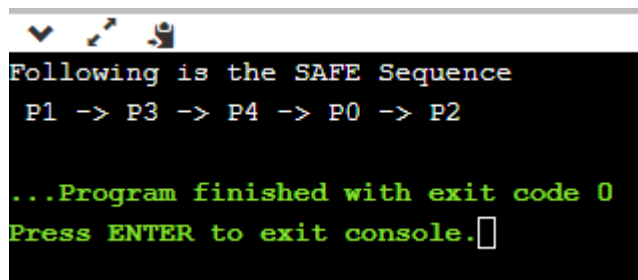
```

```

}

```

Output:



```
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2

...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment Name: 7.b Study and simulate the concept of Dining-philosophers problem.

Theory:

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

PROGRAM

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

```

{
printf("\nAll are hungry..\nDead lock stage will occur");
printf("\nExiting..");
}
else
{
for(i=0;i<howhung;i++)
{
printf("Enter philosopher %d position: ",(i+1));
scanf("%d", &hu[i]); status[hu[i]]=2;
}
do
{
printf("1.One can eat at a time\t2.Two can eat at a time\t3.Exit\nEnter your choice:");
scanf("%d", &cho);
switch(cho)
{
case 1: one();
break;
case 2: two();
break;
case 3: exit(0);
default: printf("\nInvalid option..");
}
}while(1);
}
}
one()
{
int pos=0, x, i;
printf("\nAllow one philosopher to eat at any time\n");
for(i=0;i<howhung; i++, pos++)
{ printf("\nP %d is granted to eat", philname[hu[pos]]);
for(x=pos;x<howhung;x++)
printf("\nP %d is waiting", philname[hu[x]]);
}
}
two()
{
int i, j, s=0, t, r, x;
printf("\n Allow two philosophers to eat at same time\n");
for(i=0;i<howhung;i++)
{ for(j=i+1;j<howhung;j++)
{ if(abs(hu[i]-hu[j])>=1&& abs(hu[i]-hu[j])!=4)
{ printf("\n\ncombination %d \n", (s+1));
t=hu[i];
r=hu[j];
s++;
printf("\nP %d and P %d are granted to eat", philname[hu[i]], philname[hu[j]]);
for(x=0;x<howhung;x++)

```



```

{
if((hu[x]!=t)&&(hu[x]!=r))
printf("\nP %d is waiting", philname[hu[x]]);
}
}
}
}
}
}
}
}

```

INPUT AND OUTPUT

DINING PHILOSOPHER PROBLEM

Enter the total no. of philosophers: 5

How many are hungry : 3

Enter philosopher 1 position: 2

Enter philosopher 2 position: 4

Enter philosopher 3 position: 5

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

P 5 is granted to eat

P 5 is waiting

P 0 is waiting

P 0 is granted to eat

P 0 is waiting

1. One can eat at a time

2. Two can eat at a time

3. Exit Enter your choice: 2

Allow two philosophers to eat at same time

combination 1

P 3 and P 5 are granted to eat

P 0 is waiting

combination 2

P 3 and P 0 are granted to eat

P 5 is waiting

combination 3

P 5 and P 0 are granted to eat

P 3 is waiting

1.One can eat at a time

2.Two can eat at a time

3.Exit Enter your choice: 3

Conclusion:

Questions:

1. Define resource. Give examples.

2. What is deadlock?

3. What are the conditions to be satisfied for the deadlock to occur?

4. How can be the resource allocation graph used to identify a deadlock situation?

5. How is Banker's algorithm useful over resource allocation graph technique?

6. Differentiate between deadlock avoidance and deadlock prevention?

7. Define clearly the dining-philosophers problem?

8. Identify the scenarios in the dining-philosophers problem that leads to the deadlock situations?

Experiment No: 8

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: 8 a. Write a program to demonstrate the concept of MVT and MFT memory management techniques
b. Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.

Theory:

MVT :

MVT stands for multiprogramming with variable number of tasks. Multiprogramming is a technique to execute number of programs simultaneously by a single processor. This is one of the memory management techniques. To eliminate the same of the problems with fixed partitions, an approach known as dynamic partitioning developed. In this technique, partitions are created dynamically, so that each process is loaded into partition of exactly the same size at that process. This scheme suffers from external fragmentation.

MFT:

MFT stands for multiprogramming with fixed no of tasks.

MFT is the one of the memory management technique. In this technique, main memory is divided into no of static partitions at the system generated time. A process may be loaded into a partition of equal or greater size. The partition sizes are depending on o.s. in this memory management scheme the o.s occupies the low memory, and the rest of the main memory is available for user space. This scheme suffers from internal as well as external fragmentation

PROGRAM

MFT MEMORY MANAGEMENT TECHNIQUE

```
#include<stdio.h>
#include<conio.h>
main()
{
int ms, bs, nob, ef,n, mp[10],tif=0;
int i,p=0;
clrscr();
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
```

```

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

```

INPUT

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

OUTPUT

| | PROCESS | MEMORY-REQUIRED | ALLOCATED | INTERNAL-FRAGMENTATION |
|---|---------|-----------------|-----------|------------------------|
| 1 | 275 | YES | 25 | |

| | | | |
|---|-----|-----|-------|
| 2 | 400 | NO | ----- |
| 3 | 290 | YES | 10 |
| 4 | 293 | YES | 7 |

Memory is Full, Remaining Processes cannot be accommodated

Total Internal Fragmentation is 42

Total External Fragmentation is 100

MVT MEMORY MANAGEMENT TECHNIQUE

```
#include<stdio.h>
#include<conio.h>
main()
{
int ms,mp[10],i, temp,n=0;
char ch = 'y';
clrscr();
printf("\nEnter the total memory available (in Bytes)-- ");
scanf("%d",&ms);
temp=ms;
for(i=0;ch=='y';i++,n++)
{
printf("\nEnter memory required for process %d (in Bytes) -- ",i+1);
scanf("%d",&mp[i]);
if(mp[i]<=temp)
{
printf("\nMemory is allocated for Process %d ",i+1);
temp = temp - mp[i];
}
else
{
printf("\nMemory is Full");
break;
}
printf("\nDo you want to continue(y/n) -- ");
scanf(" %c", &ch);
}
printf("\n\nTotal Memory Available -- %d", ms);
printf("\n\n\tPROCESS\t\tMEMORY ALLOCATED ");
for(i=0;i<n;i++)
printf("\n \t%d\t\t\t%d",i+1,mp[i]);
printf("\n\nTotal Memory Allocated is %d",ms-temp);
printf("\nTotal External Fragmentation is %d",temp);
getch();
}
```

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

OUTPUT

Memory is Full
Total Memory Available -- 1000
PROCESS MEMORY-ALLOCATED
1 400
2 275
Total Memory Allocated is 675

Total External Fragmentation is 325

Experiment Name: 8.b Study and simulate the following contiguous memory allocation Techniques a) First fit b) Best fit c) Worst fit

Theory:

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

First Fit Algorithm

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Now allocate processes
if(block size \geq process size)
//allocate the process
else
//move on to next block
4. Display the processes with the blocks that are allocated to a respective process.
5. Stop.

Program:

```
#include<stdio.h>
void main()
{
    int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

    for(i = 0; i < 10; i++)
    {
        flags[i] = 0;
        allocation[i] = -1;
    }

    printf("Enter no. of blocks: ");
    scanf("%d", &bno);

    printf("\nEnter size of each block: ");
    for(i = 0; i < bno; i++)
        scanf("%d", &bsize[i]);

    printf("\nEnter no. of processes: ");
    scanf("%d", &pno);

    printf("\nEnter size of each process: ");
    for(i = 0; i < pno; i++)
        scanf("%d", &psize[i]);
    for(i = 0; i < pno; i++) //allocation as per first fit
        for(j = 0; j < bno; j++)
            if(flags[j] == 0 && bsize[j] >= psize[i])
            {
                allocation[j] = i;
                flags[j] = 1;
                break;
            }

    //display allocation details
    printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
    for(i = 0; i < bno; i++)
    {
        printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
        if(flags[i] == 1)
            printf("%d\t\t%d", allocation[i]+1, psize[allocation[i]]);
        else
            printf("Not allocated");
    }
}
```

Output:


```

Enter no. of blocks: 5

Enter size of each block: 100
500
200
300
600

Enter no. of processes: 4

Enter size of each process: 212
4177
112
426

Block no.      size      process no.      size
1              100      Not allocated
2              500       1              212
3              200       3              112
4              300      Not allocated
5              600       4              426

```

Best Fit Algorithm

In the case of the best fit memory allocation scheme, the operating system searches for the empty memory block. When the operating system finds the memory block with minimum wastage of memory, it is allocated to the process. This scheme is considered as the best approach as it results in most optimised memory allocation. However, finding the best fit memory allocation may be time-consuming.

Best Fit Algorithm

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the above definition.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

Program:

```

#include <stdio.h>
void main()
{
    // declaration and initialization of variables
    int fragment[20],b[20],p[20],i,j,nb,np,tem,low=9999;
    static int barray[20],parray[20];
    printf("Memory Management Scheme - Best Fit");
    //input number of processes
    printf("Enter the number of processes:");
    scanf("%d",&np);
    //input number of blocks
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    //input number of blocks
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)

```

```

        {
            printf("Block no.:%d:",i);
            scanf("%d",&b[i]);
        }
        //input the size of process
        printf("\nEnter the size of the processes :-\n");
        for(i=1;i<=np;i++)
    {
        printf("Process no.:%d:",i);
        scanf("%d",&p[i]);
    }
    for(i=1;i<=np;i++)
    {
        for(j=1;j<=nb;j++)
        {
            if(barray[j]!=1)
            {
                tem=b[j]-p[i];
                if(tem>=0)
                if(low>tem)
                {
                    parray[i]=j;
                    low=tem;
                }
            }
        }

        fragment[i]=low;
        barray[parray[i]]=1;
        low=10000;
    }
    //Print the result
    printf("\nProcess_number \tProcess_size\tBlock_number \tBlock_size\tFragment");
    for(i=1;i<=np && parray[i]!=0;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}

```

Output:

```

Memory Management Scheme - Best FitEnter the number of processes:5

Enter the number of blocks:5

Enter the size of the blocks:-
Block no.1:7
Block no.2:11
Block no.3:13
Block no.4:15
Block no.5:17

Enter the size of the processes :-
Process no.1:5
Process no.2:8
Process no.3:10
Process no.4:12
Process no.5:16

```

| Process_number | Process_size | Block_number | Block_size | Fragment |
|----------------|--------------|--------------|------------|----------|
| 1 | 5 | 1 | 7 | 2 |
| 2 | 8 | 2 | 11 | 3 |
| 3 | 10 | 3 | 13 | 3 |
| 4 | 12 | 4 | 15 | 3 |
| 5 | 16 | 5 | 17 | 1 |

Worst Fit Algorithm

The Worst Fit Memory Allocation Algorithm allocates the **largest free partition** available in the memory that is sufficient enough to hold the process within the system. It searches the complete memory for available free partitions and allocates the process to the memory partition which is the largest out of all. This algorithm is not recommended to be implemented in the real world as it has many disadvantages. A process entering first may be allocated the largest memory space but if another process of larger memory requirement is to be allocated, space cannot be found. This is a serious drawback here.

Program:

```

#include<stdio.h>
#define max 25
void main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
    static int bf[max],ff[max];
    printf("\n\tMemory Management Scheme - Worst Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++) {printf("Block %d:",i);scanf("%d",&b[i]);}
    printf("Enter the size of the files :-\n");

```

```

for(i=1;i<=nf;i++) {printf("File %d:",i);scanf("%d",&f[i]);}
for(i=1;i<=nf;i++)
{
for(j=1;j<=nb;j++)
{
if(bf[j]!=1) //if bf[j] is not allocated
{
temp=b[j]-f[i];
if(temp>=0)
if(highest<temp)
{
ff[i]=j;
highest=temp;
}
}
}
frag[i]=highest;
bf[ff[i]]=1;
highest=0;
}
printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
for(i=1;i<=nf;i++)
printf("\n%d\t%d\t%d\t%d\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
}

```

Output:

```

Memory Management Scheme - Worst Fit
Enter the number of blocks:4
Enter the number of files:4

Enter the size of the blocks:-
Block 1:45
Block 2:24
Block 3:10
Block 4:67
Enter the size of the files :-
File 1:23
File 2:8
File 3:56
File 4:18

File_no:      File_size :      Block_no:      Block_size:      Fragement
1             23             4             67             44
2             8             1             45             37
3             56             0             1             0
4             18             2             24             6

```

Conclusion:

Questions:

1. What is the purpose of memory management unit?

2. Differentiate between logical address and physical address?

3. What are the different types of address binding techniques?

4. What is the basic idea behind contiguous memory allocation?

5. How is dynamic memory allocation useful in multiprogramming operating systems?

6. Differentiate between equal sized and unequal sized MFT schemes?

7. What is the advantage of MVT memory management scheme over MFT?

8. What is external fragmentation?

9. What is Internal fragmentation?

Experiment No: 9

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: 9.a Write a program to demonstrate the concept of demand paging for simulation of Virtual Memory implementation.

Virtual Memory

Virtual Memory is a storage scheme that provides user an illusion of having a very big main memory. This is done by treating a part of secondary memory as the main memory.

In this scheme, User can load the bigger size processes than the available main memory by having the illusion that the memory is available to load the process.

Instead of loading one big process in the main memory, the Operating System loads the different parts of more than one process in the main memory.

By doing this, the degree of multiprogramming will be increased and therefore, the CPU utilization will also be increased.

How Virtual Memory Works?

In modern word, virtual memory has become quite common these days. In this scheme, whenever some pages needs to be loaded in the main memory for the execution and the memory is not available for those many pages, then in that case, instead of stopping the pages from entering in the main memory, the OS search for the RAM area that are least used in the recent times or that are not referenced and copy that into the secondary memory to make the space for the new pages in the main memory.

Since all this procedure happens automatically, therefore it makes the computer feel like it is having the unlimited RAM.

Demand Paging

Demand Paging is a popular method of virtual memory management. In demand paging, the pages of a process which are least used, get stored in the secondary memory.

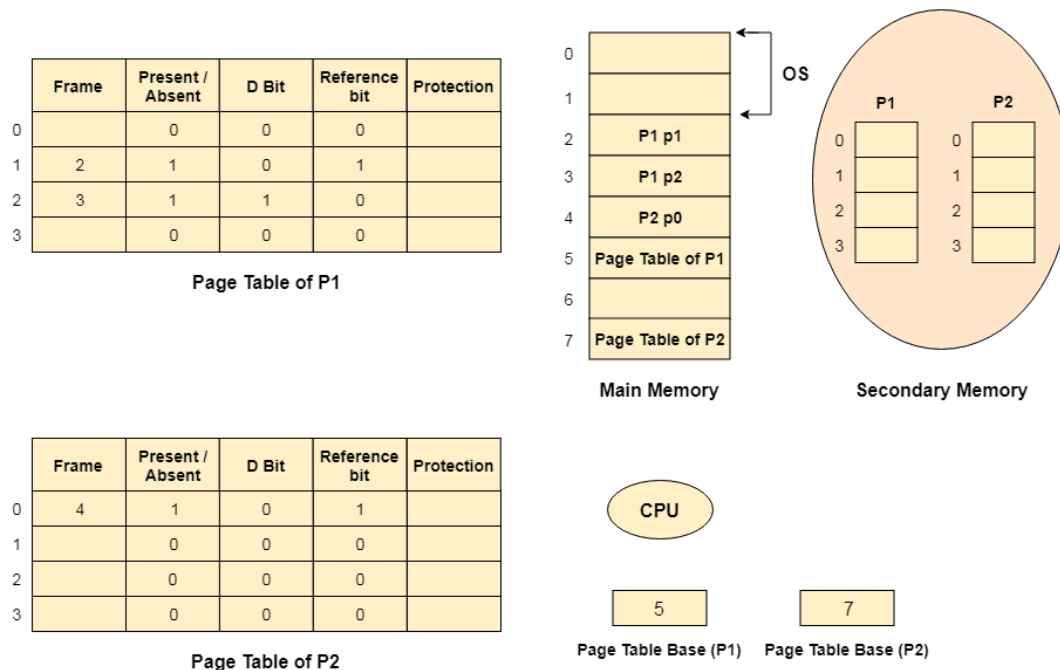
A page is copied to the main memory when its demand is made or page fault occurs. There are various page replacement algorithms which are used to determine the pages which will be replaced. We will discuss each one of them later in detail.

Snapshot of a virtual memory management system

Let us assume 2 processes, P1 and P2, contains 4 pages each. Each page size is 1 KB. The main memory contains 8 frame of 1 KB each. The OS resides in the first two partitions. In the third partition, 1st page of P1 is stored and the other frames are also shown as filled with the different pages of processes in the main memory.

The page tables of both the pages are 1 KB size each and therefore they can be fit in one frame each. The page tables of both the processes contain various information that is also shown in the image.

The CPU contains a register which contains the base address of page table that is 5 in the case of P1 and 7 in the case of P2. This page table base address will be added to the page number of the Logical address when it comes to accessing the actual corresponding entry.



Advantages of Virtual Memory

1. The degree of Multiprogramming will be increased.
2. User can run large application with less real RAM.
3. There is no need to buy more memory RAMs.

Disadvantages of Virtual Memory

1. The system becomes slower since swapping takes time.
2. It takes more time in switching between applications.
3. The user will have the lesser hard disk space for its use.

Experiment Name: 9.b To study and implement page replacement algorithms for memory management

Theory:

Page replacement takes the following approach. IF no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in

memory. We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
 - i. If there is a free frame, use it.
 - ii. If there is no free frame, use a page replacement algorithm to select a victim frame.
 - iii. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.

- **First-in, first-out (FIFO):**

The simplest page-replacement algorithm is a FIFO algorithm. The first-in, first-out (FIFO) page replacement algorithm is a low-overhead algorithm that requires little bookkeeping on the part of the operating system. The idea is obvious from the name – the operating system keeps track of all the pages in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a page needs to be replaced, the page at the front of the queue (the oldest page) is selected. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences Belady's anomaly.

Algorithm:

1. Start
2. Get input as memory block to be added to cache
3. Consider an element of the array
4. If cache is not full, add element to the cache array
5. If cache is full, check if element is already present
6. If it is hit is incremented
7. If not, element is added to cache removing first element (which is in first).
8. Repeat step 3 to 7 for remaining elements
9. Display the cache at very instance of step 8
10. Print hit ratio
11. End.

Example:

reference string

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | | 2 | 2 | 4 | 4 | 4 | 0 | | | 0 | 0 | | | 7 | 7 | 7 |
| | 0 | 0 | 0 | | 3 | 3 | 3 | 2 | 2 | 2 | | | 1 | 1 | | | 1 | 0 | 0 |
| | | 1 | 1 | | 1 | 0 | 0 | 0 | 3 | 3 | | | 3 | 2 | | | 2 | 2 | 1 |

page frames

- **Least recently used (LRU):**

The least recently used (LRU) page replacement algorithm, though similar in name to NRU, differs in the fact that LRU keeps track of page usage over a short period of time, while NRU just looks at the usage in the last clock interval. LRU works on the idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too. While LRU can provide near-optimal performance in theory (almost as good as Adaptive Replacement Cache), it is rather expensive to implement in practice. There are a few implementation methods for this algorithm that try to reduce the cost yet keep as much of the performance as possible.

One important advantage of the LRU algorithm is that it is amenable to full statistical analysis. It has been proven, for example, that LRU can never result in more than N-times more page faults than OPT algorithm, where N is proportional to the number of pages in the managed pool.

Algorithm:

1. Start
2. Get input as memory block to be added to cache
3. Consider an element of the array
4. If cache is not full, add element to the cache array
5. If cache is full, check if element is already present
6. If it is hit is incremented
7. If not, element is added to cache removing least recently used element
8. Repeat step 3 to 7 for remaining elements
9. Display the cache at very instance of step 8
10. Print hit ratio
11. End

Example:

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|--|---|--|---|---|---|---|--|--|---|--|---|--|---|--|--|
| 7 | 7 | 7 | 2 | | 2 | | 4 | 4 | 4 | 0 | | | 1 | | 1 | | 1 | | |
| | 0 | 0 | 0 | | 0 | | 0 | 0 | 3 | 3 | | | 3 | | 0 | | 0 | | |
| | | 1 | 1 | | 3 | | 3 | 2 | 2 | 2 | | | 2 | | 2 | | 7 | | |

page frames

FIFO

This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

Program:

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

```

{
    int referenceString[10], pageFaults = 0, m, n, s, pages, frames;
    printf("\nEnter the number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter reference string values:\n");
    for(m = 0; m < pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &referenceString[m]);
    }
    printf("\n What are the total number of frames:\t");
    {
        scanf("%d", &frames);
    }
    int temp[frames];
    for(m = 0; m < frames; m++)
    {
        temp[m] = -1;
    }
    for(m = 0; m < pages; m++)
    {
        s = 0;
        for(n = 0; n < frames; n++)
        {
            if(referenceString[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
            temp[m] = referenceString[m];
        }
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = referenceString[m];
        }
        printf("\n");
        for(n = 0; n < frames; n++)
        {
            printf("%d\t", temp[n]);
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}

```

Output:

```

Enter the number of Pages:      5

Enter reference string values:
Value No. [1]:  4
Value No. [2]:  3
Value No. [3]:  2
Value No. [4]:  5
Value No. [5]:  4

What are the total number of frames:  3

4      -1      -1
4      3      -1
4      3      2
5      3      2
5      4      2
Total Page Faults:      5

...Program finished with exit code 0
Press ENTER to exit console.

```

LRU

Least Recently Used (LRU) page replacement algorithm works on the concept that the pages that are heavily used in previous instructions are likely to be used heavily in next instructions. And the page that are used very less are likely to be used less in future. Whenever a page fault occurs, the page that is least recently used is removed from the memory frames. Page fault occurs when a referenced page is not found in the memory frames.

Program:

```

#include<stdio.h>
int main()
{
    int frames[10], temp[10], pages[10];
    int total_pages, m, n, position, k, l, total_frames;
    int a = 0, b = 0, page_fault = 0;
    printf("\nEnter Total Number of Frames:\t");
    scanf("%d", &total_frames);
    for(m = 0; m < total_frames; m++)
    {
        frames[m] = -1;
    }
    printf("Enter Total Number of Pages:\t");
    scanf("%d", &total_pages);
    printf("Enter Values for Reference String:\n");
    for(m = 0; m < total_pages; m++)
    {
        printf("Value No. [%d]:\t", m + 1);
        scanf("%d", &pages[m]);
    }
}

```

```

for(n = 0; n < total_pages; n++)
{
    a = 0, b = 0;
    for(m = 0; m < total_frames; m++)
    {
        if(frames[m] == pages[n])
        {
            a = 1;
            b = 1;
            break;
        }
    };
    if(a == 0)
    {
        for(m = 0; m < total_frames; m++)
        {
            if(frames[m] == -1)
            {
                frames[m] = pages[n];
                b = 1;
                break;
            }
        }
        page_fault++;
    }
    if(b == 0)
    {
        for(m = 0; m < total_frames; m++)
        {
            temp[m] = 0;
        }
        for(k = n - 1, l = 1; l <= total_frames - 1; l++, k--)
        {
            for(m = 0; m < total_frames; m++)
            {
                if(frames[m] == pages[k])
                {
                    temp[m] = 1;
                }
            }
        }
        for(m = 0; m < total_frames; m++)
        {
            if(temp[m] == 0)
            position = m;
        }
        frames[position] = pages[n];
        page_fault++;
    }
}
printf("\n");
for(m = 0; m < total_frames; m++)

```

```

        {
            printf("%d\t", frames[m]);
        }
    }
    printf("\nTotal Number of Page Faults:\t%d\n", page_fault);
    return 0;
}

```

Output:

```

Enter Total Number of Frames:  3
Enter Total Number of Pages:  6
Enter Values for Reference String:
Value No.[1]:  2
Value No.[2]:  4
Value No.[3]:  3
Value No.[4]:  2
Value No.[5]:  1
Value No.[6]:  7

2      -1      -1
2      4      -1
2      4      3
2      4      3
2      1      3
2      1      7
Total Number of Page Faults:  5

```

Optimal

This algorithm is also known as **Clairvoyant Replacement Algorithm**. As per the optimal page replacement technique, the page with the highest label should be removed first. When a page needs to be swapped into the memory, the OS will swap out the page which is not required to be used in the near future. This page replacement algorithm is a little unreliable to implement and, therefore, it cannot be implemented in a general-purpose operating system.

Program:

```

#include<stdio.h>
int main()
{
    int reference_string[25], frames[25], interval[25];
    int pages, total_frames, page_faults = 0;
    int m, n, temp, flag, found;
    int position, maximum_interval, previous_frame = -1;
    printf("\nEnter Total Number of Pages:\t");
    scanf("%d", &pages);
    printf("\nEnter Values of Reference String\n");
    for(m = 0; m < pages; m++)
    {
        printf("Value No.[%d]:\t", m + 1);
        scanf("%d", &reference_string[m]);
    }
}

```

```

printf("\nEnter Total Number of Frames:\t");
scanf("%d", &total_frames);
for(m = 0; m < total_frames; m++)
{
    frames[m] = -1;
}
for(m = 0; m < pages; m++)
{
    flag = 0;
    for(n = 0; n < total_frames; n++)
    {
        if(frames[n] == reference_string[m])
        {
            flag = 1;
            printf("\t");
            break;
        }
    }
    if(flag == 0)
    {
        if (previous_frame == total_frames - 1)
        {
            for(n = 0; n < total_frames; n++)
            {
                for(temp = m + 1; temp < pages; temp++)
                {
                    interval[n] = 0;
                    if (frames[n] == reference_string[temp])
                    {
                        interval[n] = temp - m;
                        break;
                    }
                }
            }
            found = 0;
            for(n = 0; n < total_frames; n++)
            {
                if(interval[n] == 0)
                {
                    position = n;
                    found = 1;
                    break;
                }
            }
        }
        else
        {
            position = ++previous_frame;
            found = 1;
        }
    }
}

```

```

    if(found == 0)
    {
        maximum_interval = interval[0];
        position = 0;
        for(n = 1; n < total_frames; n++)
        {
            if(maximum_interval < interval[n])
            {
                maximum_interval = interval[n];
                position = n;
            }
        }
        frames[position] = reference_string[m];
        printf("FAULT\t");
        page_faults++;
    }
    for(n = 0; n < total_frames; n++)
    {
        if(frames[n] != -1)
        {
            printf("%d\t", frames[n]);
        }
    }
    printf("\n");
}
printf("\nTotal Number of Page Faults:\t%d\n", page_faults);
return 0;
}

```

Output:

```

Enter Total Number of Pages:    6

Enter Values of Reference String
Value No.[1]:    1
Value No.[2]:    2
Value No.[3]:    3
Value No.[4]:    4
Value No.[5]:    2
Value No.[6]:    1

Enter Total Number of Frames:    3
FAULT    1
FAULT    1        2
FAULT    1        2        3
FAULT    1        2        4
        1        2        4
        1        2        4

Total Number of Page Faults:    4

```


Conclusion:

Questions:

1. Define the concept of virtual memory?

2. What is the purpose of page replacement?

3. Define the general process of page replacement?

4. List out the various page replacement techniques?

5. What is page fault?

6. Define the concept of thrashing? What is the scenario that leads to the situation of thrashing?

7. Define the concept of thrashing? What is the scenario that leads to the situation of thrashing?

Experiment No: 10

Name of Student: _____

Roll NO: _____ Date of Performance: _____

Staff Signature: _____ Marks: _____

Experiment Name: 10.a Study and Simulate all file allocation strategies a) Sequential b) Indexed c) Linked.

Theory:

Sequential method

The Sequential File Allocation or Contiguous File Allocation Method has an easy memory access advantage over the other two file allocation methods. In the contiguous File Allocation, the file is stored in sequential memory blocks and they are next to each other. So, when we have to search some files we look into the directory (directory has the starting block address of each file) and reach the starting block where the file starts and from there we will just read the next blocks in order to access the complete file. This access method also allows us to directly access the blocks of the memory as we can calculate easily where our required information is located.

Sequential File Allocation Program Algorithm:

- STEP 1: Start the program.
- STEP 2: Gather information about the number of files.
- STEP 3: Gather the memory requirement of each file.
- STEP 4: Allocate the memory to the file in a sequential manner.
- STEP 5: Select any random location from the available location.
- STEP 6: Check if the location that is selected is free or not.
- STEP 7: If the location is allocated set the flag = 1.
- STEP 8: Print the file number, length, and the block allocated.
- STEP 9: Gather information if more files have to be stored.
- STEP 10: If yes, then go to STEP 2.
- STEP 11: If no, Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
void recurse(int files[]){
    int flag = 0, startBlock, len, j, k, ch;
    printf("Enter the starting block and the length of the files: ");
    scanf("%d%d", &startBlock, &len);
    for (j=startBlock; j<(startBlock+len); j++){
        if (files[j] == 0)
            flag++;
    }
    if(len == flag){
        for (int k=startBlock; k<(startBlock+len); k++){
```

```

        if (files[k] == 0){
            files[k] = 1;
            printf("%d\t%d\n", k, files[k]);
        }
    }
    if (k != (startBlock+len-1))
        printf("The file is allocated to the disk\n");
}
else
    printf("The file is not allocated to the disk\n");

printf("Do you want to enter more files?\n");
printf("Press 1 for YES, 0 for NO: ");
scanf("%d", &ch);
if (ch == 1)
    recurse(files);
else
    exit(0);
return;
}

int main()
{
    int files[50];
    for(int i=0;i<50;i++)
        files[i]=0;
    printf("Files Allocated are :\n");
    recurse(files);
    return 0;
}

```

Output:

```

Files Allocated are :
Enter the starting block and the length of the files: 10 5
10      1
11      1
12      1
13      1
14      1
The file is allocated to the disk
Do you want to enter more files?
Press 1 for YES, 0 for NO: 0

...Program finished with exit code 0
Press ENTER to exit console.

```

Indexed Method

Indexed allocation supports both sequential and direct access files. The file indexes are not physically stored as a part of the file allocation table. Whenever the file size increases, we can easily add some more blocks to the index. In this strategy, the file allocation table contains a

single entry for each file. The entry consisting of one index block, the index blocks having the pointers to the other blocks. No external fragmentation.

Algorithm for Indexed File Allocation:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer. If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

Program:

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d----->%d : %d\n",ind,index[k],f[index[k]]);
```

```

}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
}

```

Output:

```

Enter the index block: 5
Enter no of blocks needed and no of files for the index 5 on the disk :
4
1 2 3 4
Allocated
File Indexed
5----->1 : 1
5----->2 : 1
5----->3 : 1
5----->4 : 1
Do you want to enter more file(Yes - 1/No - 0)0

```

Linked list method

It is easy to allocate the files because allocation is on an individual block basis. Each block contains a pointer to the next free block in the chain. Here also the file allocation table consisting of a single entry for each file. Using this strategy any free block can be added to a chain very easily. There is a link between one block to another block, that's why it is said to be linked allocation. We can avoid the external fragmentation.

Algorithm for Linked File Allocation:

- Step 1: Create a queue to hold all pages in memory
- Step 2: When the page is required replace the page at the head of the queue
- Step 3: Now the new page is inserted at the tail of the queue
- Step 4: Create a stack
- Step 5: When the page fault occurs replace page present at the bottom of the stack
- Step 6: Stop the allocation.

Program:

```

#include<stdio.h>
#include<stdlib.h>
void main()

```

```

{
int f[50], p,i, st, len, j, c, k, a;
for(i=0;i<50;i++)
f[i]=0;
printf("Enter how many blocks already allocated: ");
scanf("%d",&p);
printf("Enter blocks already allocated: ");
for(i=0;i<p;i++)
{
scanf("%d",&a);
f[a]=1;
}
x: printf("Enter index starting block and length: ");
scanf("%d%d", &st,&len);
k=len;
if(f[st]==0)
{
for(j=st;j<(st+k);j++)
{
if(f[j]==0)
{
f[j]=1;
printf("%d----->%d\n",j,f[j]);
}
}
else
{
printf("%d Block is already allocated \n",j);
k++;
}
}
else
printf("%d starting block is already allocated \n",st);
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
}

```

Output:

```

Enter how many blocks already allocated: 3
Enter blocks already allocated: 1 3 5
Enter index starting block and length: 2 3
2----->1
3 Block is already allocated
4----->1
5 Block is already allocated
6----->1
Do you want to enter more file(Yes - 1/No - 0)0

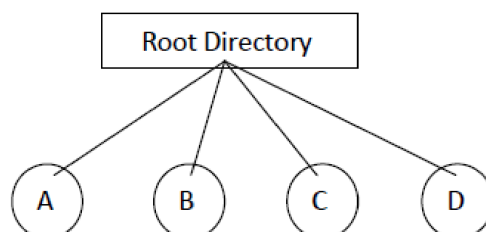
```

Experiment Name: 10.b Write a C program to simulate file organization of multi-level directory structure

THEORY:

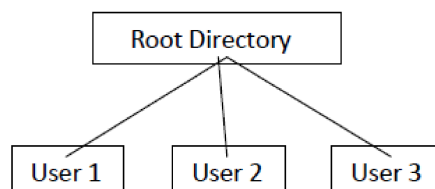
The directory contains information about the files, including attributes, location and ownership. Sometimes the directories consists of subdirectories also. The directory is itself a file, owned by the o.s and accessible by various file management routines.

a)Single Level Directories: It is the simplest of all directory structures, in this the directory system having only one directory, it consisting of the all files. Sometimes it is said to be root directory. The following diagram shows single level directory that contains four files (A, B, C, D).



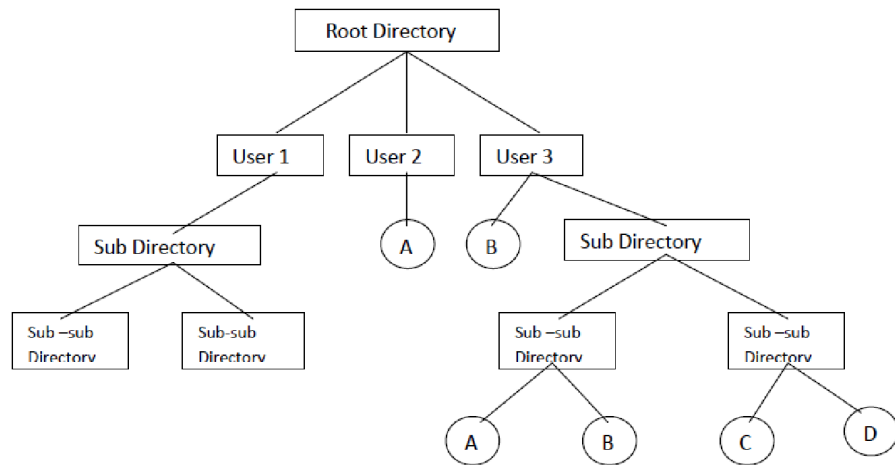
It has the simplicity and ability to locate files quickly. it is not used in the multi-user system, it is used on small embedded system.

b) Two Level Directory: The problem in single level directory is different users may be accidentally using the same names for their files. To avoid this problem, each user need a private directory. In this way names chosen by one user don't interface with names chosen by a different user. The following dig 2-level directory



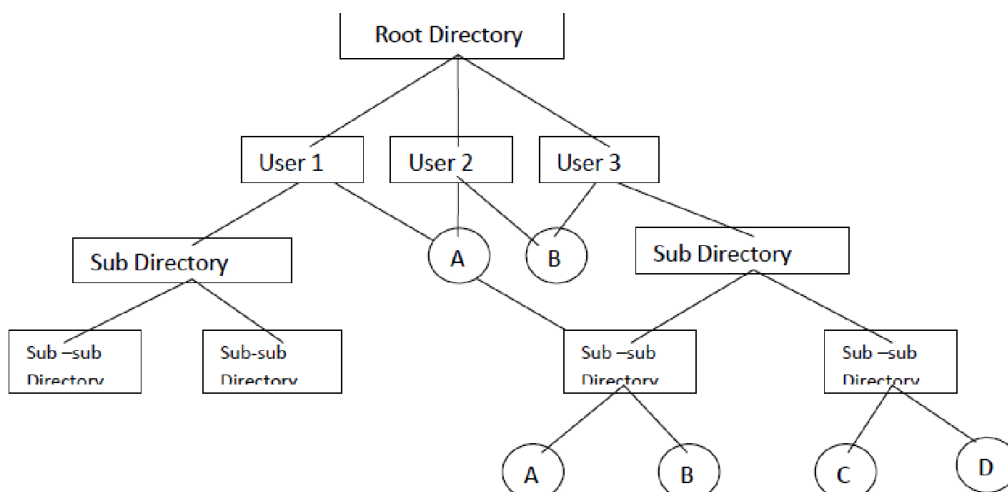
Here root directory is the first level directory it consisting of entries of user directory. User1, User2, User3 are the user levels of directories. A, B, C are the files.

c) Hierarchical Directory: The two level directories eliminate name conflicts among users but it is not satisfactory for users with a large no of files. To avoid this, create the subdirectory and load the same type of the files into the subdirectory. So, in this method each can have as many directories are needed.



This directory structure looks like tree, that's why it is also said to be tree-level directory structure

d) General graph Directory: When we add links to an existing tree structured directory, the tree structure is destroyed, resulting in a simple graph structure. This structure is used to traversing is easy and file sharing also possible.



```

#include<stdio.h>
#include<graphics.h>
struct tree_element
{
char name[20];
int x,y,ftype,lx,rx,nc,level;
struct tree_element *link[5];
};
typedef struct tree_element node;
void main()
{

```

```

int gd=DETECT,gm;
node *root;
root=NULL;
create(&root,0,"root",0,639,320);
initgraph(&gd,&gm,"c:\\tc\\BGI");
display(root);
closegraph();
}
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++)

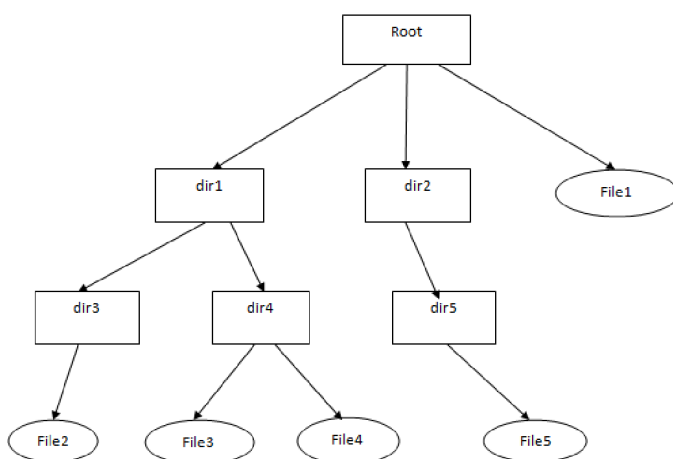
```

```

create(&((*root)->link[i]),lev+1,(*root)->name,lx+gap*i,lx+gap*i+gap,lx+gap*i+gap/2);
}
else
(*root)->nc=0;
}
}
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]);
}
}
}
}

```

Output:



Experiment Name: 10.c To study and implement disk scheduling algorithms

Theory:

In operating systems, seek time is very important. Since all device requests are linked in queues, the seek time is increased causing the system to slow down. Disk Scheduling Algorithms are used to reduce the total seek time of any request.

- **First Come -First Serve (FCFS)**

All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.

- **Shortest Seek Time First (SSTF)**

In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seems to be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to each other the other requests will never be handled since the distance will always be greater.

- **Scan**

This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks. Once again this is more optimal than the previous algorithm, but it is not the best.

FCFS

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,n,TotalHeadMoment=0,initial;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    // logic for FCFS disk scheduling
    for(i=0;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
```

```

        initial=RQ[i];
    }
    printf("Total head moment is %d",TotalHeadMoment);
    return 0;
}

```

Output:

```

Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Total head movement is 644

```

Scan disk Sheduling

```

#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);
    // logic for Scan disk scheduling
    /*logic for sort the request array */
    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;
}

```

Output:

```

Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 337

```

C-scan disk scheduling

```

#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);

    // logic for C-Scan disk scheduling

    /*logic for sort the request array */
    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);
    /*movement max to min disk */
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial=0;
    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];
    }
    // last movement for min size
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
    /*movement min to max disk */
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial =size-1;
    for(i=n-1;i>=index;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
}

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

```

Output:

```

Enter the number of Request
8
Enter the Requests Sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 382

```

Conclusion:

Questions:

1. Define file.

2. What are the different kinds of files?

3. How many types of file allocation methods are there? Briefly explain with example.

4. What file allocation strategy is most appropriate for random access files?

5. In how many ways we can access a file? What are they?

6. What file access pattern is particularly suited to chained file allocation on disk?

7. What is the purpose of file allocation strategies?

8. Identify ideal scenarios where sequential, indexed and linked file allocation strategies are most appropriate?

9. What are the disadvantages of sequential file allocation strategy?

10. What is an index block?

11. What is the file allocation strategy used in UNIX?

12. What are the attributes of a file?

13. Define directory?

14. Describe the general directory structure?

15. List the different types of directory structures?

16. Which of the directory structures is efficient? Why?

17. Which directory structure does not provide user-level isolation and protection?

18. What is the advantage of hierarchical directory structure?
