

218556: Operating Systems Laboratory

Second Year (2020 Course)
Semester - II

Teaching Scheme		Examination Scheme	
Practical :	2 Hrs. / Week	Practical	25 Marks
		Term Work	25 Marks



LABORATORY MANUAL

DEPARTMENT OF AI&ML Engineering

Samarth College of Engineering, Pune
2023-2024

INDEX

Sr. No	Title of Experiment	Page No.	Marks	Sign
1.	<p>A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.</p> <p>B. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit</p>			
2.	<p>Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.</p> <p>A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also, demonstrate zombie and orphan states.</p> <p>B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program that display array in reverse order.</p>			
3.	<p>Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.</p>			
4.	<p>A. Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex.</p>			

	B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.			
5.	Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.			
6.	Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.			
7.	<p>Inter process communication in Linux using following.</p> <p>A. FIFOS: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.</p> <p>B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.</p>			
8.	Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, and C-Look considering the initial head position moving away from the spindle.			

Certified that Mr/Miss _____ *of*

class _____ Sem _____ Roll no. _____ has completed the term work satisfactorily in
the subject _____ of the Department _____ of SGOI
College of Engineering Belhe, During academic year _____.

Staff Member

Prof. Shelake S. D.

Head of Dept.

Assignment No. 1

To study **Basic Linux Commands and Shell programming.**

- A. Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.
- B. Write a program to implement an address book with options given below:
 - a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit

OBJECTIVE:

To study

- 1. Basic Linux commands
- 2. Shell script

THEORY:

Linux Commands

The Linux command is a utility of the Linux operating system. All basic and advanced tasks can be done by executing commands. The commands are executed on the Linux terminal. The terminal is a command-line interface to interact with the system, which is similar to the command prompt in the Windows OS. Commands in Linux are case-sensitive

Basic Linux Commands

❖ mkdir Command

The mkdir command is used to create a new directory under any directory Syntax:

mkdir <directory name>

❖ rmdir Command

The rmdir command is used to delete a directory. Syntax:

`rmdir <directory name>`

❖ **cd Command**

The `cd` command is used to change the current directory. Syntax:

`cd <directory name>`

❖ **touch Command**

The `touch` command is used to create empty files. We can create multiple empty files by executing it once.

Syntax:

`touch <file name> touch`

`<file1> <file2>`

❖ **cat Command**

The `cat` command is a multi-purpose utility in the Linux system. It can be used to create a file, display content of the file, copy the content of one file to another file, and more.

Syntax:

`cat [OPTION]... [FILE]..`

To create a file, execute it as follows:

`cat > <file name>`

// Enter file content

Press "CTRL+ D" keys to save the file. To display the content of the file, execute it as follows:

`cat <file name>`

❖ **rm Command**

The `rm` command is used to remove a file. Syntax:

`rm <file name>`

❖ **ls Command**

The ls command is used to display a list of content of a directory. Syntax:

ls

❖ cat Command

The cat command is also used as a filter. To filter a file, it is used inside pipes.

Syntax:

cat <fileName> | cat or tac | cat or tac |

❖ grep Command

The grep is the most powerful and used filter in a Linux system. The 'grep' stands for "global regular expression print." It is useful for searching the content from a file. Generally, it is used with the pipe.

Syntax:

command | grep <searchWord>

❖ sed command

The sed command is also known as stream editor. It is used to edit files using a regular expression. It does not permanently edit files; instead, the edited content remains only on display. It does not affect the actual file.

Syntax:

command | sed 's/<oldWord>/<newWord>/'

❖ sort Command

The sort command is used to sort files in alphabetical order. Syntax:

sort <file name>

❖ cal Command

The cal command is used to display the current month's calendar with the current date highlighted.

Syntax:

cal<

❖ read command

The read command is used to read from a file descriptor. This command read up the total number of bytes from the specified file descriptor into the buffer. If the number or count is zero then this command may detect the errors. But on success, it returns the number of bytes read. Zero indicates the end of the file. If some errors found then it returns -1

Syntax:

read

❖ echo command

echo command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax:

echo [option] [string]

❖ test command

Test is used as part of the conditional execution of shell commands. test exits with the status determined by EXPRESSION. Placing the EXPRESSION between square brackets ([and]) is the same as testing the EXPRESSION with test. To see the exit status at the command prompt, echo the value "\$?". A value of 0 means the expression evaluated as true, and a value of 1 means the expression evaluated as false

Syntax:

test EXPRESSION

[EXPRESSION]

Shell Script: Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands), the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is known as shell script. Shell Script is series of command written in plain text file. This manual is meant as a brief introduction to features found in Bash.

Exit Status:

By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not.

- (1) If return value is zero (0), command is successful.
- (2) If return value is nonzero, command is not successful or some sort of error executing command/shell script.

This value is known as Exit Status. But how to find out exit status of command or shell script? Simple, to determine this exit Status you can use \$? special variable of shell.

For e.g. (This example assumes that unknow1file does not exist on your hard drive)

```
$ rm unknow1file
```

It will show error as follows

```
rm: cannot remove `unkowm1file': No such file or directory and after  
that if you give command
```

```
$ echo $?
```

it will print nonzero value to indicate error.

User defined variables (UDV)

To define UDV use following syntax Syntax:

```
variable name=value
```

'value' is assigned to given 'variable name' and Value must be on right side =

sign.

Example:

To define variable called n having value 10

```
$ n=10
```

To print or access UDV use following syntax Syntax:

```
$variablename
```

```
$ n=10
```

To print contains of variable 'n' type command as follows

```
$ echo $n
```

About Quotes

There are three types of quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclosed in double quotes removed meaning of that characters (except \ and \$).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	`Back quote` - To execute command

Example:

```
$ echo "Today is date"
```

Can't print message with today's date.

```
$ echo "Today is `date`".
```

Rules for Naming variable name (Both UDV and System Variable)

(1) Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows
HOME SYSTEM_VERSION

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error

```
$ no=10
```

But there will be problem for any of the following variable declaration:

```
$ no =10
```

```
$ no= 10
```

```
$ no = 10
```

(3) Variables are case-sensitive, just like filename in Linux.

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.

```
$ vech=
```

```
$ vech=""
```

Try to print it's value by issuing following command

```
$ echo $vech
```

Nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use ?,* etc, to name your variable names. Shell

Arithmetic

Use to perform arithmetic operations.

Syntax:

```
expr op1 math-operator op2
```

Examples:

```
$ expr 1 + 3
```

```
$ expr 2 - 1
```

```
$ expr 10 / 2
```

```
$ expr 20 % 3
```

```
$ expr 10 \* 3
```

```
$ echo `expr 6 + 3` Note:
```

expr 20 %3 - Remainder read as 20 mod 3 and remainder is 2.

expr 10 * 3 - Multiplication use * and not * since its wild card.

The read Statement

Use to get input (data from user) from keyboard and store (data) to

variable.

Syntax:

read variable1, variable2,...variableN

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

```
$ vi sayH #  
#Script to read your name from key-board #  
echo "Your first name please:" read  
fname  
echo "Hello $fname, Lets be friend!"
```

Variables in Shell

To process our data/information, data must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux (Shell), there are two types of variable:

- (1) System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
- (2) User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower letters.

You can see system variables by giving command like \$ set, some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
HOME=/home/vivek	Our home directory
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name

You can print any of the above variables contains as follows:

```
$ echo $HOME
```

test command or [expr]

test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

Syntax:

test expression OR [expression] Example:

Following script determine whether given argument number is positive.

```
if test $1 -gt 0 then
echo "$1 number is positive" fi
```

test or [expr] works with

1. Integer (Number without decimal point)
2. File types
- 3.Character strings

For Mathematics, use following operator in Shell Script

Operator in Shell Script	Meaning	Normal Arithmetical	But in Shell	
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

NOTE: == is equal, != is not equal. For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators:

Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 - a expression2	Logical AND
expression1 - o expression2	Logical OR

if condition

if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:

```
if condition
then
    command1 if condition is true or if exit status of
    condition is 0 (zero)
    ...
    ...
fi
```

Condition is defined as:

“Condition is nothing but comparison between two values.”

For compression you can use test or [expr] statements or even exist status can be also used.

Loops in Shell Scripts

Bash supports:

- 1) for loop
- 2) while loop

while :

The syntax of the while is:

```
while test-commands
do
commands
done
```

Execute *commands* as long as *test-commands* has an exit status of zero.

for :

The syntax of the for is: for

```
variable in list do
commands
done
```

Each white space-separated word in list is assigned to variable in turn and commands executed until list is exhausted.

The case Statement

The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

Syntax:

```
case $variable-name in
    pattern1)    command
                ...
                ..
                command;;
    pattern2)    command
                ...
                ..
                command;;
    patternN)    command
```



```

...
..
command;;
*) command
...
..
command;;

esac

```

The \$variable-name is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found.

Conclusion: Thus in shell script we can write series of commands and execute as a single program.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Answer the following questions. [Write Short Answer Below]

1. What are different types of shell & differentiate them.
2. Explain Exit status of a command.
3. Explain a) User define variables.
4. System variables.
5. What is man command?
6. Explain test command.

7. Explain how shell program get executed.
8. Explain syntax of if-else, for, while, case.
9. Explain use of functions in shell and show it practically.
10. Write a menu driven shell script to execute different commands with options.

Assignment No. 2

Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.

- A. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also, demonstrate zombie and orphan states. Write a program to implement an address book with options given below: a) Create address book. b) View address book. c) Insert a record. d) Delete a record. e) Modify a record. f) Exit
- B. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program that display array in reverse order.

OBJECTIVE:

To study

- Process control
- Zombie and orphan processes
- System calls: fork, execve, wait

THEORY:

Process

A process is the basic active entity in most operating-system models. A process is a program in execution in memory or in other words, an instance of a program in memory. Any program executed creates a process. A program can be a command, a shell script, or any binary executable or any application.

Process IDs

Each process in a Linux system is identified by its unique *process ID*, sometimes referred to as *pid*. Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

When referring to process IDs in a C or C++ program, always use the `pid_t` typedef, which is defined in `<sys/types.h>`. A program can obtain the process ID of the process it's running in with the `getpid()` system call, and it can obtain the process ID of its parent process with the `getppid()` system call.

Creating Process

Two common techniques are used for creating a new process.

1. using `system()` function
2. using `fork()` system calls

1. `system ()` function

The `system` function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, `system` creates a subprocess running the standard Bourne shell (`/bin/sh`) and hands the command to that shell for execution.

The `system` function returns the exit status of the shell command. If the shell itself cannot be run, `system` returns 127; if another error occurs, `system` returns `-1`.

2. `fork` system call

A process can create a new process by calling `fork`. The calling process becomes the parent, and the created process is called the child. The `fork` function copies the parent's memory image so that the new process receives a copy of the address space of the parent. Both processes continue at the instruction after the `fork` statement (executing in their respective memory images).

Synopsis

```
#include <unistd.h>

pid_t fork(void);
```

The fork function returns 0 to the child and returns the child's process ID to the parent. When fork fails, it returns -1.

Wait Function

When a process creates a child, both parent and child proceed with execution from the point of the fork. The parent can execute `wait` to block until the child finishes. The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.

Synopsis

```
#include <sys/wait.h>

pid_t wait(int *status);
```

If `wait` returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return -1.

Example:

```
pid_t childpid; childpid =
wait(NULL); if (childpid !=
-1)
printf("Waited for child with pid %ld\n", childpid);
```

Status values

The status argument of `wait` is a pointer to an integer variable. If it is not NULL, this function stores the return status of the child in this location. The child returns its status by calling `exit`, `_exit` or return from `main`.

A zero return value indicates `EXIT_SUCCESS`; any other value indicates `EXIT_FAILURE`.

POSIX specifies six macros for testing the child's return status. Each takes the status value returned by a child to wait as a parameter. Following are the two such macros:

Synopsis

```
#include <sys/wait.h> WIFEXITED(intstat_val)
WEXITSTATUS(intstat_val)
```

Exec system call

Used for new program execution within the existing process. The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent. The exec family of functions provides a facility for overlaying the process image of the calling process with a new image. The traditional way to use the fork–exec combination is for the child to execute (with an exec function) the new program while the parent continues to execute the original code. The exec system call is used after a fork system call by one of the two processes to replace the memory space with a new program. The exec system call loads a binary file into memory (destroying image of the program containing the exec system call) and go their separate ways. Within the exec family, there are functions that vary slightly in their capabilities.

Synopsis

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

```
char **environ;
```

Exec family

1. `intexecl(const char *path, const char *arg0, ... /*, char *(0) */);`
2. `intexecle (const char *path, const char *arg0, ... /*, char *(0), char *constenvp[] */);`
3. `intexeclp (const char *file, const char *arg0, ... /*, char *(0) */);`
4. `intexecv(const char *path, char *constargv[]);`
5. `intexecve (const char *path, char *constargv[], char *constenvp[]);`
6. `intexecvp (const char *file, char *constargv[]);`

1. Execl() and execlp():

execl()

It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL.

e.g. `execl("/bin/lis", "lis", "-l", NULL);`

execlp()

It does same job except that it will use environment variable PATH to determine which executable to process. Thus, a fully qualified path name would not have to be used. It can also take the fully qualified name as it also resolves explicitly.

e.g. `execlp("ls", "ls", "-l", NULL);`

2. Execv() and execvp()

execv()

It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string.

e.g. `char *argv[] = ("ls", "-l", NULL);`
`execv("/bin/ls", argv);`

execvp()

It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.

e.g. `execvp("ls", argv);`

3. execve()

It executes the program pointed to by filename.

`intexecve(const char *filename, char *constargv[], char *constenvp[]);`

Filename must be either a binary executable, or a script starting with a line of the form: `#!/bin/program`. argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form `key=value`, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

```
int main(int argc, char *argv[ ], char *envp[ ])
```

`execve()` does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

All exec functions return `-1` if unsuccessful. In case of success these functions never return to the calling function.

Process Termination

Normally, a process terminates in one of two ways. Either the executing program calls the exit function, or the program's main function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.

Zombie Processes

If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens, when a child process terminates and the parent is not calling wait? Does it simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, it becomes a zombie process.

A *zombie process* is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then calls wait. If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls wait, the child process becomes a zombie. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

Orphan Process

Orphan process is the process whose parent process is dead. That is, parent process is terminated, killed or exited but the child process is still alive.

In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically. Re-parenting means orphaned process is immediately adopted by special process. Thing to notice here is that even after re-parenting, the process still remains Orphan as the parent which created the process is dead.

A process can be orphaned either intentionally or unintentionally. Sometimes a parent process exits/terminates or crashes leaving the child process still running, and then they become orphans. Also, a process can be intentionally orphaned just to keep it running. For example when you need to run a job in the background which need any manual intervention and going to take long time, then you detach it from user session and leave it there. Same way, when you need to run a process in the background for infinite time, you need to do the same thing. Processes running in the background like this are known as daemon process.

Finding an Orphan Process

It is very easy to spot a Orphan process. Orphan process is a user process, which is having init process (process id 1) as parent. You can use this command in linux to find the orphan processes.

```
# ps -elf | head -1; ps -elf | awk '{if ($5 == 1 && $3 != "root") {print $0}}' | head
```

This will show you all the orphan processes running in your system. The output from this command confirms that they are orphan processes but does not mean that they are all useless.

Killing an Orphan Process

As orphan processes waste server resources, it is not advised to have lots of orphan processes running into the system. To kill a orphan process is same as killing a normal process.

```
# kill -15 <PID>
```

If that does not work then simply use # kill -9

<PID>

Daemon Process

A process runs in the background, rather than under the direct control of a user. They are usually initiated as background processes.

vfork

It is alternative of fork. Creates a new process when exec a new program.

Comparison with fork

1. Creates new process without fully copying the address space of the parent.
2. vfork guarantees that the child runs first, until the child calls exec or exit.
3. When child calls either of these two functions (exit, exec), the parent resumes.

Input

1. An integer array with specified size.
2. An integer array with specified size and number to search.

Output

1. Sorted array.
2. Status of number to be searched.

FAQs

- Is Orphan process different from a Zombie process?
- Are Orphan processes harmful for system?
- Is it bad to have Zombie processes on your system?
- How to find an Orphan Process?
- How to find a Zombie Process?
- What is common shared data between parent and child process?
- What are the contents of Process Control Block?

Practice Assignments

Example 1

Printing the Process ID

```
#include <stdio.h>
#include <unistd.h> int
main()
{
    printf("The process ID is %d\n", (int) getpid()); printf("The parent
    process ID is %d\n", (int) getppid()); return 0;
}
```

Example 2

Using the system call

```
#include <stdlib.h> int
main()
{
    int return_value; return_value=system("ls -l
    /"); return return_value;
}
```

Example 3

Using fork to duplicate a program's process

```
#include <stdio.h> #include
<unistd.h> #include
<sys/types.h> int main()
{
    pid_t child_pid;
    printf("The main program process ID is %d\n", (int) getpid()); child_pid=fork();
    if(child_pid!=0)
    {
        printf("This is the parent process ID, with id %d\n", (int) getpid()); printf("The child
        process ID is %d\n", (int) child_pid);
    }
    else
        printf("This is the child process ID, with id %d\n", (int) getpid()); return 0;
}
```

Example 4

Determining the exit status of a child

```
#include <stdio.h> #include <sys/types.h> #include <sys/wait.h> void
show_return_status(void)
{
pid_t childpid; int
status;
childpid = wait(&status); if (childpid == -1) perror("Failed to wait
for child");
else if (WIFEXITED(status))
printf("Child %ld terminated with return status %d\n", (long)childpid, WEXITSTATUS(status));
}
```

Example 5

A program that creates a child process to run ls -l

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h> int
main(void)
{
pid_t childpid;
childpid = fork(); if
(childpid == -1) {
perror("Failed to fork"); return 1;
}
if (childpid == 0) {
/* child code */
execl("/bin/ls", "ls", "-l", NULL); perror("Child failed
to exec ls"); return 1;
}
if (childpid != wait(NULL)) {
/* parent code */
perror("Parent failed to wait due to signal or error"); return 1;
}
return 0;
}
```

Example 6

Making a zombie process

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```

int main()
{
    pid_t child_pid;
    //create a child process
    child_pid=fork(); if(child_pid>0) {
    //This is a parent process. Sleep for a minute sleep(60)
    }
    else
    {
    //This is a child process. Exit immediately. exit(0);
    }
    return 0;
}

```

Example 7

Demonstration of fork system call

```

#include<stdio.h> #include
<stdlib.h> #include
<sys/types.h> #include
<unistd.h> int main()
{
    pid_t pid; char
    *msg; int n;
    printf("Program starts\n");
    pid=fork();
    switch(pid)
    {
    case -1:
    printf("Fork error\n");
    exit(-1);
    case 0:
    msg="This is the child process"; n=5;
    break;
    default:
    msg="This is the parent process"; n=3;
    break;
    }
    while(n>0)
    {

```

```
puts(msg);
sleep(1);
n--;
}
return 0;
}
```

Example 8

Demo of multi-process application using fork () system call

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

#define SIZE 1024

void do_child_proc(intpfd[2]); void
do_parent_proc(intpfd[2]); int main()
{
    intpfd[2];
    intret_val,nread;
    pid_tpid;
    ret_val=pipe(pfd);
    if(ret_val== -1)
    {
        perror("pipe error\n");
        exit(ret_val);
    }
    pid=fork();
    switch(pid)
    {
        case -1:
            printf("Fork error\n");
            exit(pid);
        case 0: do_child_proc(pfd);
            exit(0);
        default:
            do_parent_proc(pfd);
            exit(pid);
    }
    wait(NULL);
    return 0;
}
```

```

void do_child_proc(int pfd[2])
{
    int nread;
    char *buf=NULL;
    printf("5\n");
    close(pfd[1]);
    while(nread=(read(pfd[0],buf,size))!=0)
    printf("Child Read=%s\n",buf); close(pfd[0]);
    exit(0);
}

void do_parent_proc(int pfd[2])
{
    char ch;
    char *buf=NULL; close(pfd[0]);
    while(ch=getchar()!='\n') {
        printf("7\n");
        *buf=ch;
        buff++;
    }
    *buf='\0'; write(pfd[1],buf,strlen(buf)+1);
    close(pfd[1]);
}

```

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Assignment No. 3

Implement the C program for **CPU Scheduling Algorithms**: Shortest Job First (Preemptive) and Round Robin with different arrival time.

OBJECTIVE:

To study

- Preemptive and Non-Preemptive CPU scheduling
- Application and use of CPU scheduling Algorithm

THEORY:

Shortest Job First (Preemptive) What

is Shortest Job First?

This is an approach which considers the next CPU burst. Each process possess its next CPU burst. When CPU is available, the process having the smallest next CPU burst is allocated CPU.

It may happen that two or more processes have the same next CPU burst. Then which process to allocate will be decided as per FCFS scheduling.

Shortest job first(SJF) is a scheduling algorithm, that is used to schedule processes in an operating system. It is a very important topic in Scheduling when compared to round-robin and FCFS Scheduling.

There are two types of SJF

- Pre-emptive SJF
- Non-Preemptive SJF

These algorithms schedule processes in the order in which the shortest job is done first. It has a minimum average waiting time.

There are 3 factors to consider while solving SJF, they are

1. BURST Time

2. Average waiting time
3. Average turnaround time

Shortest Remaining Time First (SRTF) scheduling:

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Implementation Points:

- 1- Traverse until all process gets completely executed.
 - a) Find process with minimum remaining time at every single time lap.
 - b) Reduce its time by 1.
 - c) Check if its remaining time becomes 0
 - d) Increment the counter of process completion.
 - e) Completion time of current process = current_time +1;
 - e) Calculate waiting time for each completed process. $wt[i] = \text{Completion time} - \text{arrival_time_burst_time}$
 - f) Increment time lap by one.
- 2- Find turnaround time (waiting_time+burst_time).

Key Differences Between Preemptive and Non-Preemptive Scheduling:

1. In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.
2. The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.
3. In Preemptive Scheduling, there is the overhead of switching the process from the ready state to running state, vise-verse and maintaining the ready queue.

Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.

4. In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. , in the non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with small burst time may have to starve.
5. Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.
6. Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative which is not the case with Non-preemptive Scheduling.

Shortest Job First Advantages and Disadvantages:

Advantages

- This algorithm is simple to implement.
- Does not depend on any priority of the process. The smallest burst time is the higher priority consideration.
- It provides good CPU utilization than FCFS (First Come First Search).
- Waiting time and turnaround time of each process is reduced, reducing the average waiting time and turn around the time of the system as compared to FCFS.

Disadvantages

- Waiting time of some processes still high due to the long burst time of the processes, in case of non-preemptive scheduling.
- In the case of non-preemptive scheduling, it may act as a uni-processing operating system.
- In the case of preemptive scheduling, context switch is required.
- And in preemptive scheduling, turnaround time may get increased.

Preemptive SJF Example:

Process	Duration	Order	Arrival Time
P1	9	1	0
P2	2	2	2

Shortest Job First (Preemptive) Code Implementation:

Code:

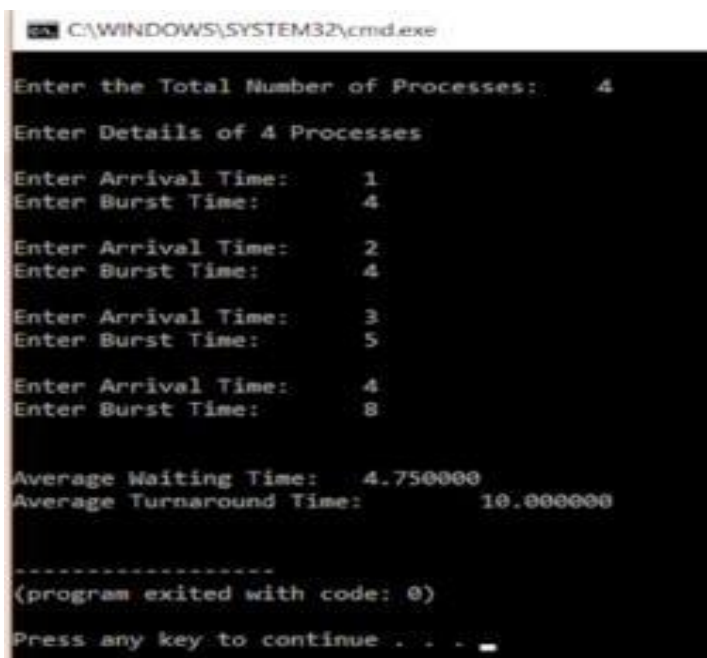
```
#include <stdio.h> int
main()
{
    int arrival_time[10], burst_time[10], temp[10]; int i,
    smallest, count = 0, time, limit;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time; printf("nEnter
the Total Number of Processes:t"); scanf("%d", &limit);
    printf("nEnter Details of %d Processesn", 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("\nAverage Waiting Time:t%lf\n",
average_waiting_time);
printf("Average Turnaround Time:t%lf\n", average_turnaround_time); return 0;
}

```

Output



```

C:\WINDOWS\SYSTEM32\cmd.exe
Enter the Total Number of Processes: 4
Enter Details of 4 Processes
Enter Arrival Time: 1
Enter Burst Time: 4
Enter Arrival Time: 2
Enter Burst Time: 4
Enter Arrival Time: 3
Enter Burst Time: 5
Enter Arrival Time: 4
Enter Burst Time: 8
Average Waiting Time: 4.750000
Average Turnaround Time: 10.000000
-----
(program exited with code: 0)
Press any key to continue . . .

```

Round Robin with different arrival time Round

Robin Scheduling

- The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns.
- Each process is assigned a fixed time slot in a cyclic way. Time Quantum
- It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.
- In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice.
- This algorithm also offers starvation free execution of processes.

Characteristics of Round Robin

- Round robin is a pre-emptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
- The process that is preempted is added to the end of the queue.
- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
- It is a real time algorithm which responds to the event within a specific time limit.
- Round robin is one of the oldest, fairest, and easiest algorithm.
- Widely used scheduling method in traditional OS.

Advantages of Round Robin

- It does not face any starvation issues or convoy effect.
- Each process gets equal priority to the fair allocation of CPU.
- It deals with all process without any priority.
- It is easy to implement the CPU Scheduling algorithm.

- Each new process is added to the end of the ready queue as the next process's arrival time is reached.
- Each process is executed in circular order that shares a fixed time slot or quantum.
- Every process gets an opportunity in the round-robin scheduling algorithm to reschedule after a given quantum period.
- This scheduling method does not depend upon burst time. That's why it is easily implementable on the system.

Disadvantages of Round Robin

- If the time quantum is lower, it takes more time on context switching between the processes.
- It does not provide any special priority to execute the most important process.
- The waiting time of a large process is higher due to the short time slot.
- The performance of the algorithm depends on the time quantum.
- The response time of the process is higher due to large slices to time quantum.
- Getting a correct time slot or quantum is quite difficult for all processes in the round-robin algorithm.

Example of Round Robin #1

Process Queue	Burst Time
P1	4
P2	3
P3	5

Queue Representation based on burst on time

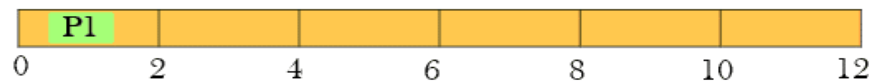
Now Time Slice = 2



- **Step 1)** The execution begins with process P1, which has burst time 4. Here, every process executes for 2 seconds. P2 and P3 are still in the waiting queue.



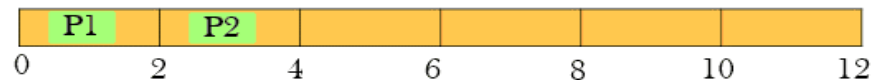
- Time Slice = 2



- **Step 2)** At time =2, P1 is added to the end of the Queue and P2 starts executing



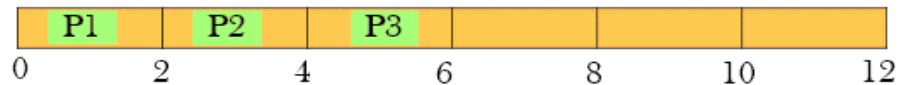
- Time Slice = 2



- **Step 3)** At time=4 , P2 is preempted and add at the end of the queue. P3 starts executing.



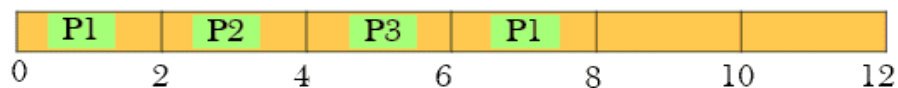
- Time Slice = 2



- **Step 4)** At time=6 , P3 is preempted and add at the end of the queue. P1 starts executing.



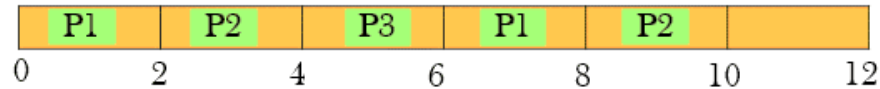
- Time Slice = 2



- **Step 5)** At time=8 , P1 has a burst time of 4. It has completed execution. P2 starts execution



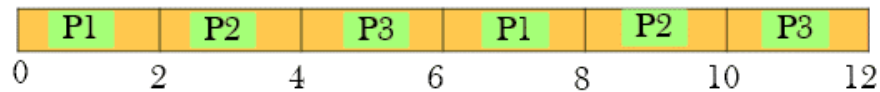
- Time Slice = 2



- **Step 6)** P2 has a burst time of 3. It has already executed for 2 interval. At time=9, P2 completes execution. Then, P3 starts execution till it completes.



- Time Slice = 2



- Now as all the process get equal time slice for execution let's calculate average waiting time(Service Time - Arrival Time).

Process Queue	Burst time	Waiting Time
P1	4	0+ 4= 4
P2	3	2+4= 6
P3	5	4+3= 7

Round Robin scheduling Code implementation

Code:

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

```
void main()
```

```
{
```

```
    // initialize the variable name
```

```
    int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
```

```
    float avg_wt, avg_tat;
```

```
    printf(" Total number of process in the system: "); scanf("%d",
    &NOP);
```

```
    y = NOP; // Assign the number of process to variable y
```



```

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1); printf(" Arrival time
is: \t"); // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time scanf("%d",
&bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time quantum
printf("Enter the Time Quantum for the process: \t"); scanf("%d",
&quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time "); for(sum=0, i = 0; y!=0; )
{
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
sum = sum + temp[i]; temp[i] =
0;
count=1;
}
else if(temp[i] > 0)
{
temp[i] = temp[i] - quant; sum
= sum + quant;
}
if(temp[i]==0 && count==1)
{
y--; //decrement the process no.
printf("\nProcess No[%d] \t\t %d\t\t\t %d\t\t\t %d", i+1, bt[i], sum- at[i], sum-at[i]-bt[i]);
wt = wt+sum-at[i]-bt[i]; tat
= tat+sum-at[i]; count =0;
}
}

```

```

if(i==NOP-1)
{
    i=0;
}
else if(at[i+1]<=sum)
{
    i++;
}
else
{
    i=0;
}
}
}

```

Output:

```

C:\Users\AMIT YADAV\Documents\amit.exe
Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is: 0
Burst time is: 8

Enter the Arrival and Burst time of the Process[2]
Arrival time is: 1
Burst time is: 5

Enter the Arrival and Burst time of the Process[3]
Arrival time is: 2
Burst time is: 10

Enter the Arrival and Burst time of the Process[4]
Arrival time is: 3
Burst time is: 11
Enter the Time Quantum for the process: 6

Process No      Burst Time      TAT      Waiting Time
Process No[2]   5              10       5
Process No[1]   8              25       17
Process No[3]   10             27       17
Process No[4]   11             31       20
Average Turn Around Time: 14.750000
Average Waiting Time: 23.250000

```

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

Assignment No. 4

Thread Synchronization

- A. Thread synchronization using counting semaphores. Application to demonstrate: producer-consumer problem with counting semaphores and mutex.
- B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.

OBJECTIVE:

To study

- Semaphores
- Mutex
- Producer-Consumer Problem
- Reader- Writer problem

THEORY:

Semaphores:

Semaphore is an integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.

The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. It is known as a counting semaphore or a general semaphore. Semaphores are the OS tools for synchronization.

Two types:

1. Binary Semaphore.
2. Counting Semaphore

Binary semaphore

Semaphores, which are restricted to the values 0 and 1 (or locked/unlocked, unavailable / available), are called binary semaphores and are used to implement locks.

It is a means of suspending active processes, which are later to be reactivated at such time conditions are right for it to continue. A binary semaphore is a pointer which when held by a process grants them exclusive use to their critical section. It is a (sort of) integer variable which can take the values 0 or 1 and be operated upon only by two commands termed in English wait and signal.

Counting semaphore

Semaphores which allow an arbitrary resource count are called counting semaphores.

A counting semaphore comprises:

An integer variable, initialized to a value K ($K \geq 0$). During operation it can assume any value $\leq K$, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

A counting semaphore can be implemented as follows:

- Initialize – initialize to non-negative integer
- Decrement (semWait)
 - Process executes this to receive a signal via semaphore.
 - If signal is not transmitted, process is suspended.
 - Decrements semaphore value
 - If value becomes negative , process is blocked
 - Otherwise it continues execution.
- Increment (semSignal)
 - Process executes it to transmit a signal via semaphore.
 - Increments semaphore value

- If value is less than or equal to zero, process blocked by semWait is unblocked

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Value of semaphore

- Positive
Indicates number of processes that can issue wait & immediately continue to execute.
- Zero
By initialization or because number of processes equal to initial semaphore value have issued a wait Next process to issue a wait is blocked.
- Negative
Indicates number of processes waiting to be unblocked Each signal unblocks one waiting process.

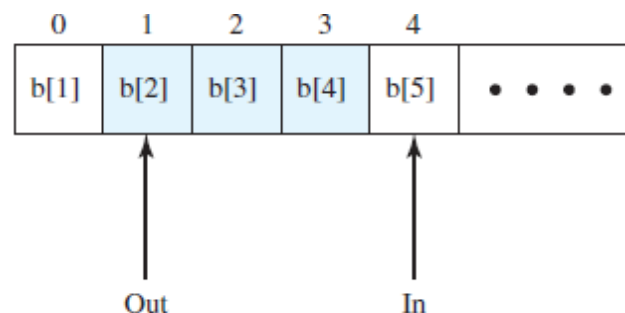
The Producer/Consumer Problem

There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

```
producer:
while (true) {
    /* produce item v */;
    b[in] = v;
    in++;
}

consumer:
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */;
}
```

Figure illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the



Note: Shaded area indicates portion of buffer that is occupied

Figure: Infinite buffer for producer/consumer problem

Solution for bounded buffer using counting semaphore

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

The Reader Writer Problem

In dealing with the design of synchronization and concurrency mechanisms, it is useful to be able to relate the problem at hand to known problems and to be able to test any solution in terms of its ability to solve these known problems. The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes

that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike.

In the readers/writers problem readers do not also write to the data area, nor do writers read the data area while writing.

For example, suppose that the shared area is a library catalog. Ordinary users of the library read the catalog to locate a book. One or more librarians are able to update the catalog. In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time, it is important to prevent writers from interfering with each other and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

Readers Have Priority

Figure is a solution using semaphores, showing one instance each of a reader and a writer; the solution does not change for multiple readers and writers. The writer process is simple. The semaphore wsem is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on wsem. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable read count is used to keep track of the number of readers, and the semaphore x is used to assure that read count is updated properly.


```

/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true){
        semWait (x);
        readcount++;
        if(readcount == 1)
            semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if(readcount == 0)
            semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true){
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader,writer);
}

```

POSIX Semaphores

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (`sem_post(3)`); and decrement the semaphore value by one (`sem_wait(3)`). If the value of a semaphore is currently zero, then a `sem_wait(3)` operation will block until the value becomes greater than zero.

Semaphore functions:

1. `sem_init()`

It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

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

2. sem_wait()

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

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

3. sem_post()

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

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

1. sem_unlink

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

```
int sem_unlink(const char *name)
```

All the above functions returns

0 : Success

-1 : Error

Mutex

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

1. pthread_mutex_init()

The function shall initialize the mutex referenced by `mutex` with attributes specified by `attr`. If `attr` is `NULL`, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restrict attr);
```

2. **pthread_mutex_lock()**

The mutex object referenced by `mutex` shall be locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

```
int pthread_mutex_lock(pthread_mutex_t * mutex);
```

3. **pthread_mutex_unlock()**

The function shall release the mutex object referenced by `mutex`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `mutex` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

```
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

4. **pthread_mutex_destroy()**

The function shall destroy the mutex object referenced by `mutex`; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

CONCLUSION:

Thus, we have implemented producer-consumer problem and Reader Writer Problem using 'C' in Linux.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

FAQ

1. Explain the concept of semaphore.
2. Explain wait and signal functions associated with semaphores.
3. What do binary and counting semaphores mean?
4. What is mutual exclusion?

Assignment No. 5

Implement the C program for Deadlock Avoidance Algorithm: **Bankers Algorithm.**

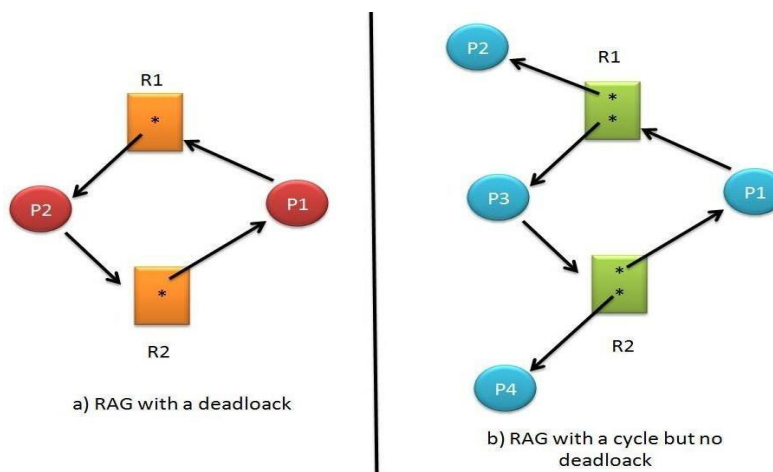
OBJECTIVE: To Study

- Deadlock
- Deadlock Avoidance Algorithm- Bankers Algorithm.

THEORY:

What is Deadlock?

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no interrupts condition is needed to prevent an otherwise deadlocked process from being awake.



Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

- ❖ Mutual exclusion condition- Each resource is either currently assigned to exactly one process or is available.
- ❖ Hold and wait condition- Processes currently holding resources granted earlier can request new resources.
- ❖ No preemption condition- Resources previously granted cannot be forcibly taken away from a process. The process holding them must explicitly release them.
- ❖ Circular wait condition- There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Banker's Algorithm in Operating System

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

Why Banker's algorithm is named so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S . If a person applies for a loan then the bank first subtracts the loan amount from the total

money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

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.
- $\text{Available}[j] = k$ 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.
- $\text{Max}[i, j] = k$ 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*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*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**
- $\text{Need}[i, j] = \text{Max}[i, j] - \text{Allocation}[i, j]$

Allocation specifies the resources currently allocated to process **P_i** and

Need 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....n  
2) Find an i such that both  
a) Finish[i] = false  
b) Needi ≤ Work  
if no such i exists goto step (4)  
3) Work = Work + Allocation[i]  
Finish[i] = true  
goto step (2)  
4) if Finish [i] = true for all i  
then the system is in a safe state
```

Resource-Request Algorithm

Let Request_i be the request array for process P_i. Request_i [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i \leq Need_i$

Goto step (2); otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq Available$

Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$Available = Available - Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

Example-

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Question1. What will be the content of the Need matrix?

$Need[i, j] = Max[i, j] - Allocation[i, j]$ So,
the content of Need Matrix is:

Process	Need		
	A	B	C
P ₀	7	4	3
P ₁	1	2	2
P ₂	6	0	0
P ₃	0	1	1
P ₄	4	3	1

Applying the Safety algorithm on the given system,

m=3, n=5 Step 1 of Safety Algo
 Work = Available
 Work =

3	3	2
---	---	---

 0 1 2 3 4
 Finish =

false	false	false	false	false
-------	-------	-------	-------	-------

For i = 0 Step 2
 Need₀ = 7, 4, 3 ✗
 Finish [0] is false and Need₀ > Work
 So P₀ must wait But Need ≤ Work

For i = 1 Step 2
 Need₁ = 1, 2, 2 ✓
 Finish [1] is false and Need₁ < Work
 So P₁ must be kept in safe sequence

Step 3
 Work = Work + Allocation₁
 Work =

5	3	2
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	false	false
-------	------	-------	-------	-------

For i = 2 Step 2
 Need₂ = 6, 0, 0 ✗
 Finish [2] is false and Need₂ > Work
 So P₂ must wait

For i = 3 Step 2
 Need₃ = 0, 1, 1 ✓
 Finish [3] is false and Need₃ < Work
 So P₃ must be kept in safe sequence

Step 3
 Work = Work + Allocation₃
 Work =

7	4	3
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	true	false
-------	------	-------	------	-------

For i = 4 Step 2
 Need₄ = 4, 3, 1 ✓
 Finish [4] is false and Need₄ < Work
 So P₄ must be kept in safe sequence

Step 3
 Work = Work + Allocation₄
 Work =

7	4	5
---	---	---

 0 1 2 3 4
 Finish =

false	true	false	true	true
-------	------	-------	------	------

For i = 0 Step 2
 Need₀ = 7, 4, 3 ✓
 Finish [0] is false and Need₀ < Work
 So P₀ must be kept in safe sequence

Step 3
 Work = Work + Allocation₀
 Work =

7	5	5
---	---	---

 0 1 2 3 4
 Finish =

true	true	false	true	true
------	------	-------	------	------

For i = 2 Step 2
 Need₂ = 6, 0, 0 ✓
 Finish [2] is false and Need₂ < Work
 So P₂ must be kept in safe sequence

Step 3
 Work = Work + Allocation₂
 Work =

10	5	7
----	---	---

 0 1 2 3 4
 Finish =

true	true	true	true	true
------	------	------	------	------

Step 4
 Finish [i] = true for 0 ≤ i ≤ n
 Hence the system is in Safe state

The safe sequence is P₁, P₃, P₄, P₀, P₂

What will happen if process P₁ requests one additional instance of resource type A and two instances of resource type C

A B C
Request₁ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1
1, 0, 2 1, 2, 2 ✓
Request₁ < Need₁

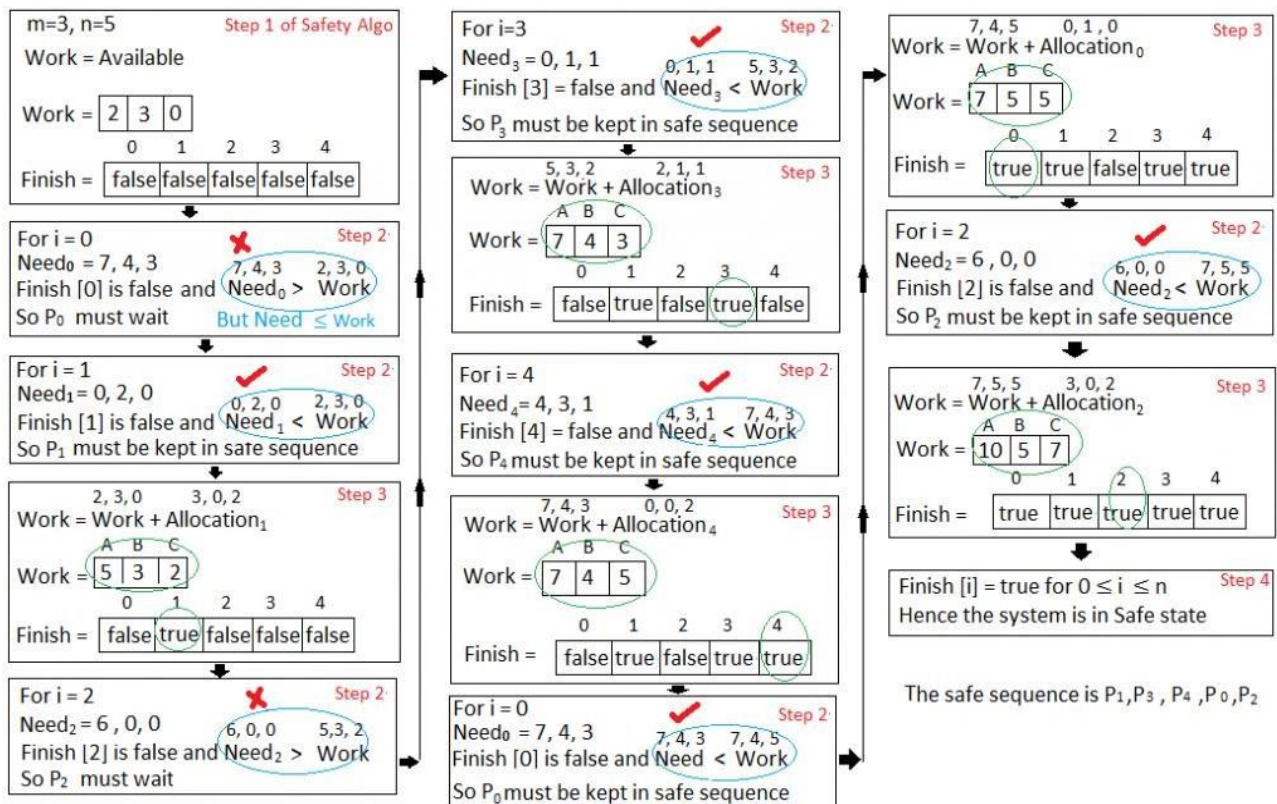
Step 2
1, 0, 2 3, 3, 2 ✓
Request₁ < Available

Step 3

Available = Available – Request₁
Allocation₁ = Allocation₁ + Request₁
Need₁ = Need₁ - Request₁

Process	Allocation	Need	Available
	A B C	A B C	A B C
P ₀	0 1 0	7 4 3	2 3 0
P ₁	3 0 2	0 2 0	
P ₂	3 0 2	6 0 0	
P ₃	2 1 1	0 1 1	
P ₄	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



CONCLUSION:

Thus, we have implemented Banker's Algorithm problem using 'C' in Linux.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

FAQ:

1. What is dead lock?
2. What are the necessary and sufficient conditions to occur deadlock?
3. What is deadlock avoidance and deadlock prevention techniques?

Assignment No. 6

Implement the C program for **Page Replacement Algorithms**: FCFS, LRU, and Optimal for frame size as minimum three.

OBJECTIVE:

To study

- Page Replacement
- Methods and Algorithms for Page Replacement

THEORY:

Page Replacement

A computer system has a limited amount of memory. Adding more memory physically is very costly. Therefore, most modern computers use a combination of both hardware and software to allow the computer to address more memory than the amount physically present on the system. This extra memory is actually called Virtual Memory.

Virtual Memory is a storage allocation scheme used by the Memory Management Unit (MMU) to compensate for the shortage of physical memory by transferring data from RAM to disk storage. It addresses secondary memory as though it is a part of the main memory. Virtual Memory makes the memory appear larger than actually present which helps in the execution of programs that are larger than the physical memory.

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault).

In Virtual Memory Management, Page Replacement Algorithms play an important role. The main objective of all the **Page replacement policies** is to decrease the maximum number of **page faults**.

Page Fault – It is basically a memory error, and it occurs when the current programs attempt to access the memory page for mapping into virtual address space, but it is unable to load into the physical memory then this is referred to as Page fault.

A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

Virtual Memory can be implemented using two methods:

- Paging
- Segmentation

Paging

Paging is a process of reading data from, and writing data to, the secondary storage. It is a memory management scheme that is used to retrieve processes from the secondary memory in the form of pages and store them in the primary memory. The main objective of paging is to divide each process in the form of pages of fixed size. These pages are stored in the main memory in frames. Pages of a process are only brought from the secondary memory to the main memory when they are needed.

When an executing process refers to a page, it is first searched in the main memory. If it is not present in the main memory, a page fault occurs.

Page Fault is the condition in which a running process refers to a page that is not loaded in the main memory.

In such a case, the OS has to bring the page from the secondary storage into the main memory. This may cause some pages in the main memory to be replaced due to limited storage. A Page Replacement Algorithm is required to decide which page needs to be replaced.

Basic Page Replacement Algorithm in OS

Page Replacement Algorithm decides which page to remove, also called swap out when a new page needs to be loaded into the main memory.

When the page that was selected for replacement was paged out, and referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

Page Replacement technique uses the following approach. If there is no free frame, then we will find the one that is not currently being used and then free it. A-frame can be freed by writing its content to swap space and then change the page table in order to indicate that the page is no longer in the memory.

1. First of all, find the location of the desired page on the disk.
2. Find a free Frame: a) If there is a free frame, then use it. b) If there is no free frame then make use of the page-replacement algorithm in order to select the victim frame. c) Then after that write the victim frame to the disk and then make the changes in the page table and frame table accordingly.
3. After that read the desired page into the newly freed frame and then change the page and frame tables.
4. Restart the process.

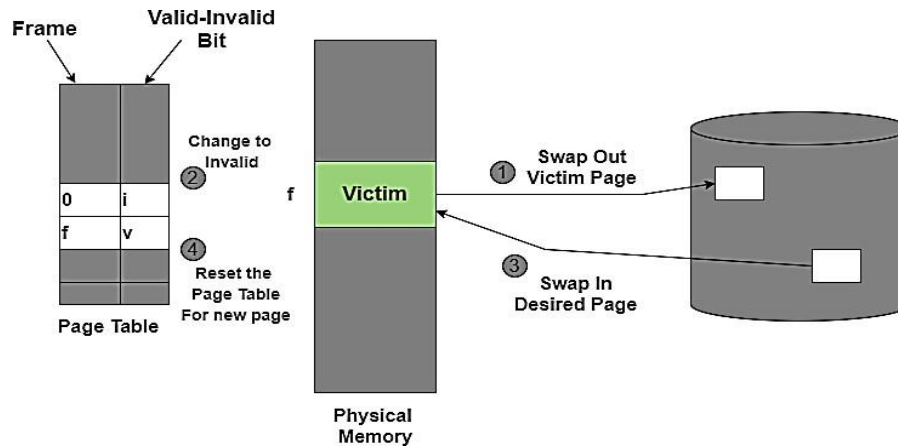
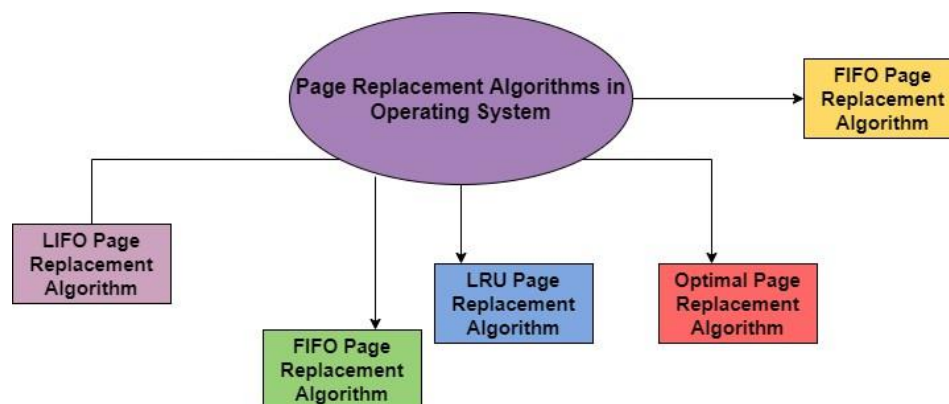


Figure: Page Replacement

Page Replacement Algorithms in OS

This algorithm helps to decide which pages must be swapped out from the main memory in order to create a room for the incoming page. This Algorithm wants the lowest page-fault rate.

Various Page Replacement algorithms used in the Operating system are as follows;



Some Page Replacement Algorithms:

- First In First Out (FIFO)
- Least Recently Used (LRU)
- Optimal Page Replacement

1. First In First Out (FIFO)

It is a very simple way of Page replacement and is referred to as First in First Out. This algorithm mainly replaces the oldest page that has been present in the main memory for the longest time.

- This algorithm is implemented by keeping the track of all the pages in the queue.
- As new pages are requested and are swapped in, they are added to the tail of a queue and the page which is at the head becomes the victim.
- This is not an effective way of page replacement but it can be used for small systems.

Advantages

- This algorithm is simple and easy to use.
- FIFO does not cause more overhead.

Disadvantages

- This algorithm does not make the use of the frequency of last used time rather it just replaces the Oldest Page.
- There is an increase in page faults as page frames increases.
- The performance of this algorithm is the worst.

ALGORITHM

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Form a queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault

8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

Code of FIFO Page Replacement Algorithm

```
#include<stdio.h> int
main()
{
    int i,j,n,a[50],frame[10],no,k,avail,count=0; printf("\n ENTER
    THE NUMBER OF PAGES:\n");
    scanf("%d",&n);
    printf("\n ENTER THE PAGE NUMBER :\n");
    for(i=1;i<=n;i++) scanf("%d",&a[i]);
    printf("\n ENTER THE NUMBER OF FRAMES :");
    scanf("%d",&no); for(i=0;i<no;i++)
    frame[i]= -1;
    j=0;
    printf("\tref string\t page frames\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\t\t",a[i]); avail=0;
        for(k=0;k<no;k++)
        if(frame[k]==a[i])
            avail=1;
        if (avail==0)
        {
            frame[j]=a[i];
```

```

        j=(j+1)%no; count++;
        for(k=0;k<no;k++)
            printf("%d\t",frame[k]);

    }

    printf("\n");

}

printf("Page Fault Is %d",count); return
0;

}

```

OUTPUT

ENTER THE NUMBER OF PAGES: 20
 ENTER THE PAGE NUMBER : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
 ENTER THE NUMBER OF FRAMES :3

	<u>ref string</u>	<u>page frames</u>	
7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1
0			
3	2	3	1
0	2	3	0
4	4	3	0
2	4	2	0
3	4	2	3
0	0	2	3
3			
2			
1	0	1	3
2	0	1	2
0			
1			
7	7	1	2
0	7	0	2
1	7	0	1

Page Fault Is 15

CONCLUSION:

Thus, we have implemented Page Replacement Algorithm using 'C' in Linux.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

Answer the following questions. [Write Short Answer Below]

1. What are different types Page Replacement Algorithm.
2. What is page fault.
3. What is demand paging.
4. Explain memory management.

Assignment No. 7

Inter Process Communication (IPC) in Linux using following.

- A. FIFOs: Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.
- B. Inter-process Communication using Shared Memory using System V. Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and writes the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

OBJECTIVE:

To study

- Communication (interface) between processes.
- FIFO method of process communication.
- Shared memory process of communication

THEORY:

Inter process communication in Linux First in first out (FIFO)

A FIFO (First In First Out) is a one-way flow of data. FIFOs have a name, so unrelated processes can share the FIFO. FIFO is a named pipe. Any process can open or close the FIFO. FIFOs are also called named pipes.

Properties:

1. After a FIFO is created, it can be opened for read or write.

2. Normally, opening a FIFO for read or write, it blocks until another process opens it for write or read.
1. A read gets as much data as it requests or as much data as the FIFO has, whichever is less.
2. A write to a FIFO is atomic, as long as the write does not exceed the capacity of the FIFO.
3. Two processes must open FIFO; one opens it as reader on one end, the other opens it as sender on the other end. The first/earlier opener has to wait until the second/later opener to come. This is somewhat like a hand shaking.

Creating a FIFO

A FIFO is created by the `mkfifo` function. Specify the path to the FIFO on the command line. For example, create a FIFO in `/tmp/fifo` by invoking this:

```
#include <sys/types.h>

#include <sys/stat.h>

intmkfifo(const char *pathname, mode_t mode);
```

pathname: a UNIX pathname (path and filename). The name of the FIFO

mode: the file permission bits. It specifies the pipe's owner, group, and world permissions, and a pipe must have a reader and a writer, the permissions must include both read and write permissions.

If the pipe cannot be created (for instance, if a file with that name already exists), `mkfifo` returns `-1`.

FIFO can also be created by the `mknod` system call,

e.g., `mknod("fifo1", S_IFIFO|0666, 0)` is same as `mkfifo("fifo1", 0666)`.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions like open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
Intfd = open (fifo_path, O_WRONLY); write
(fd, data, data_length);
close (fd);
```

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

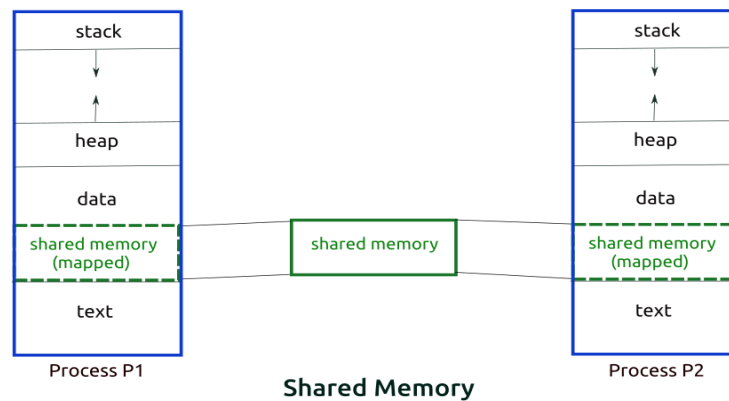
Close: To close an open FIFO, use close().

Unlink :To delete a created FIFO, use unlink().

Inter-process Communication using Shared Memory using System V.

Shared memory is one of the three inter-process communication (IPC) mechanisms available under Linux and other Unix-like systems. The other two IPC mechanisms are the message queues and semaphores. In case of shared memory, a shared memory segment is created by the kernel and mapped to the

data segment of the address space of a requesting process. A process can use the shared memory just like any other global variable in its address space.



In the inter-process communication mechanisms like the pipes, fifos and message queues, the work involved in sending data from one process to another is like this. Process *P1* makes a system call to send data to Process *P2*. The message is copied from the address space of the first process to the kernel space during the system call for sending the message. Then, the second process makes a system call to receive the message. The message is copied from the kernel space to the address space of the second process. The shared memory mechanism does away with this copying overhead. The first process simply writes data into the shared memory segment. As soon as it is written, the data becomes available to the second process. Shared memory is the fastest mechanism for inter-process communication.

We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this —

- Create the shared memory segment or use an already created shared memory segment (`shmget()`)
- Attach the process to the already created shared memory segment (`shmat()`)

- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())

Let us look at a few details of the system calls related to shared memory.

```
#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg)
```

The above system call creates or allocates a System V shared memory segment. The arguments that need to be passed are as follows –

The **first argument, key**, recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communication. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

The **second argument, size**, is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

The **third argument, shmflg**, specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

This call would return a valid shared memory identifier (used for further calls of shared memory) on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>

#include <sys/shm.h>

void * shmat(int shmid, const void *shmaddr, int shmflg)
```

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. The arguments that need to be passed are as follows –

shmid is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

shmaddr is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

shmflg is specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/types.h>

#include <sys/shm.h>

int shmdt(const void *shmaddr)
```

he above system call performs shared memory operation for System V shared memory segment of detaching the shared memory segment from the address space of the calling process. The argument that needs to be passed is –

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

```
#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

The above system call performs control operation for a System V shared memory segment. The following arguments needs to be passed –

The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)

- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from 'A' to 'E' each of 1023 bytes into the shared memory. Last byte signifies the end of buffer
- Read process would read from the shared memory and write to the standard output
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
- Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)
- Performs reading and writing process for a few times for simplification and also in order to avoid infinite loops and complicating the program

Code for write process (Writing into Shared Memory – File: shm_write.c)

```
/* Filename: shm_write.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h> #include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>

#define BUF_SIZE 1024 #define
SHM_KEY 0x1234

struct shmseg { int
    cnt;
    int complete;
    char buf[BUF_SIZE];
};
```

```

int fill_buffer(char * bufptr, int size);

int main(int argc, char *argv[]) { int
    shmidx, numtimes;
    struct shmseg *shmp; char
    *bufptr;
    int spaceavailable;
    shmidx = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT); if (shmidx ==
    -1) {
        perror("Shared memory");
        return 1;
    }

    // Attach to the segment to get a pointer to it. shmp =
    shmat(shmidx, NULL, 0);
    if (shmp == (void *) -1) { perror("Shared
        memory attach"); return 1;
    }

    /* Transfer blocks of data from buffer to shared memory */ bufptr =
    shmp->buf;
    spaceavailable = BUF_SIZE;
    for (numtimes = 0; numtimes < 5; numtimes++) { shmp->cnt
        = fill_buffer(bufptr, spaceavailable); shmp->complete =
        0;
        printf("Writing Process: Shared Memory Write: Wrote %d bytes\n", shmp-
        >cnt);
        bufptr = shmp->buf; spaceavailable =
        BUF_SIZE; sleep(3);
    }
    printf("Writing Process: Wrote %d times\n", numtimes); shmp-
    >complete = 1;

    if (shmdt(shmp) == -1) {
        perror("shmdt"); return 1;
    }

    if (shmctl(shmidx, IPC_RMID, 0) == -1) { perror("shmctl");
        return 1;
    }
    printf("Writing Process: Complete\n");
}

```

```

    return 0;
}

int fill_buffer(char * bufptr, int size) { static
    char ch = 'A';
    int filled_count;
    //printf("size is %d\n", size);
    memset(bufptr, ch, size - 1);
    bufptr[size-1] = '\0';
    if (ch > 122)
        ch = 65;
    if ( (ch >= 65) && (ch <= 122) ) { if (
        (ch >= 91) && (ch <= 96) ) {
        ch = 65;
    }
    }
    filled_count = strlen(bufptr);

    //printf("buffer count is: %d\n", filled_count);
    //printf("buffer filled is:%s\n", bufptr); ch++;
    return filled_count;
}

```

Compilation and Execution Steps

Writing Process: Shared Memory Write: Wrote 1023 bytes Writing
 Process: Shared Memory Write: Wrote 1023 bytes Writing Process:
 Shared Memory Write: Wrote 1023 bytes Writing Process: Shared
 Memory Write: Wrote 1023 bytes Writing Process: Shared Memory
 Write: Wrote 1023 bytes Writing Process: Wrote 5 times
 Writing Process: Complete

Code for read process (Reading from the Shared Memory and writing to the standard output – File: shm_read.c)

```

/* Filename: shm_read.c */
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/types.h>
#include<string.h>
#include<errno.h>

```

```

#include<stdlib.h>

#define BUF_SIZE 1024 #define SHM_KEY
0x1234 struct shmseg {
    int cnt;
    int complete;
    char buf[BUF_SIZE];
};
int main(int argc, char *argv[]) { int shmid;
    struct shmseg *shmp;
    shmid = shmget(SHM_KEY, sizeof(struct shmseg), 0644|IPC_CREAT); if (shmid == -1)
    {
        perror("Shared memory"); return 1;    }
    // Attach to the segment to get a pointer to it. shmp = shmat(shmid,
    NULL, 0);
    if (shmp == (void *) -1) { perror("Shared memory
    attach"); return 1;    }
    /* Transfer blocks of data from shared memory to stdout*/ while (shmp->complete != 1)
    {
        printf("segment contains : \n\"%s\"\n", shmp->buf); if (shmp->cnt == -1)
        {
            perror("read"); return 1;
        }
        printf("Reading Process: Shared Memory: Read %d bytes\n", shmp->cnt); sleep(3);
    }
    printf("Reading Process: Reading Done, Detaching Shared Memory\n"); if (shmdt(shmp) == -1)
    {
        perror("shmdt"); return 1;
    }
    printf("Reading Process: Complete\n"); return 0;
}

```

CONCLUSION:

Thus, we studied inter process communication using FIFOs

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

Assignment No. 8

Implement the C program for **Disk Scheduling Algorithms**: SSTF, SCAN, C- Look considering the initial head position moving away from the spindle.

OBJECTIVE:

This assignment covers the UNIX process control commonly called for process creation, program execution and process termination. Also covers process model, including process creation, process destruction, zombie and orphan processes.

THEORY:

❖ Disk Scheduling Algorithms

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

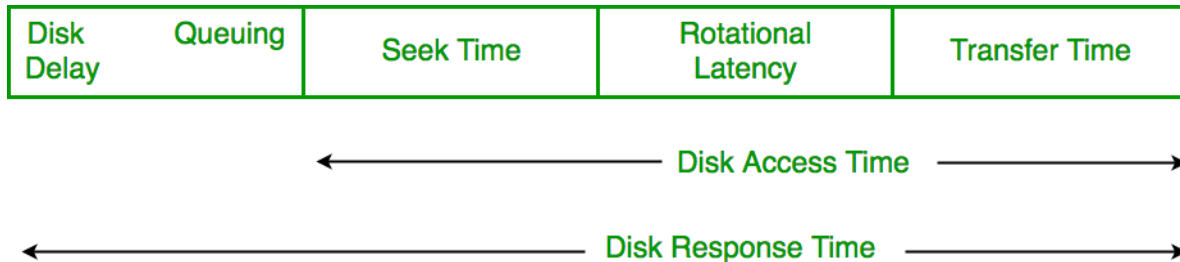
There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time**: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency**: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write

heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

- **Transfer Time:** Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time:** Disk Access Time is:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time



- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. Therefore, the disk scheduling algorithm that gives minimum variance response time is better.

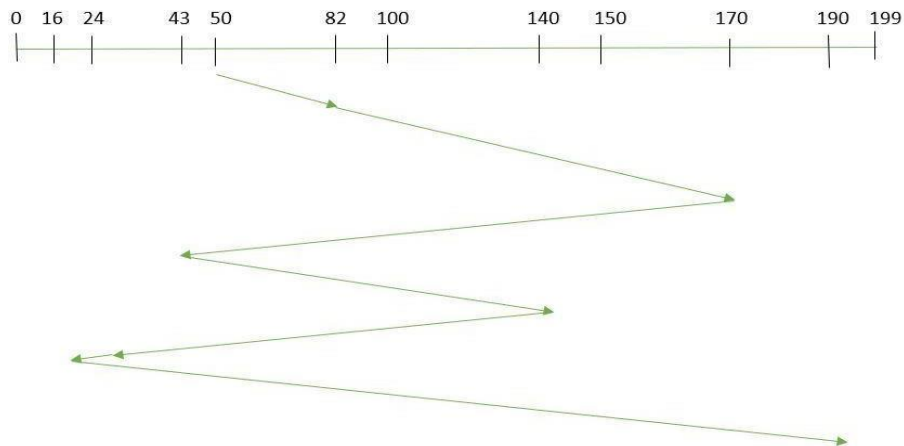
Disk Scheduling Algorithms

FCFS:

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190) And current position of Read/Write head is : 50



Advantages:

- Every request gets a fair chance
- No indefinite postponement

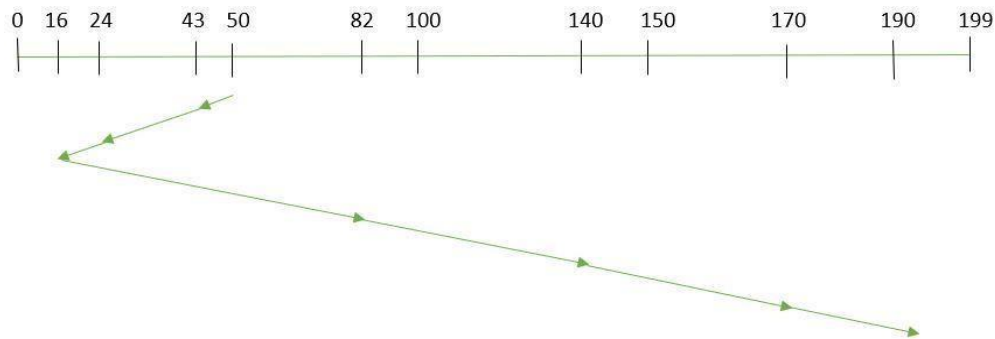
Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service

SSTF: In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system. Let us understand this with the help of an example.

Example:

Suppose the order of request is- (82,170,43,140,24,16,190) And current position of Read/Write head is : 50



So, total seek time:

$$=(50-43)+(43-24)+(24-16)+(82-16)+(140-82)+(170-140)+(190-170) =208$$

Advantages:

- Average Response Time decreases
- Throughput increases

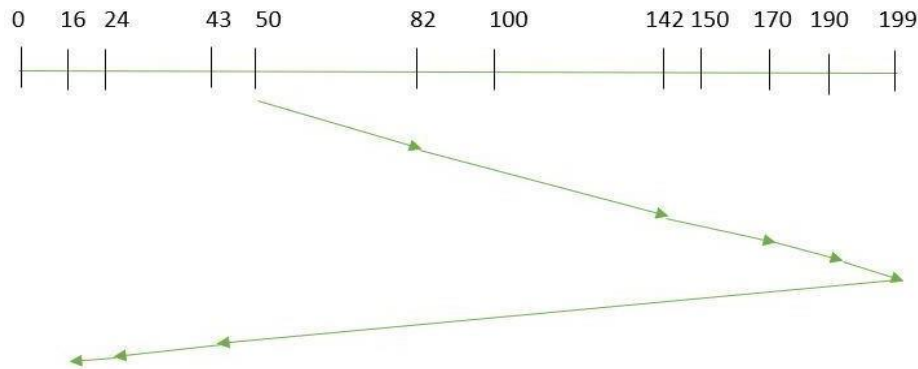
Disadvantages:

- Overhead to calculate seek time in advance
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favors only some requests

SCAN: In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example:

Suppose the requests to be addressed are-82, 170,43,140,24,16,190. In addition, the Read/Write arm is at 50, and it is given that the disk arm should move “towards the larger value”.



Therefore, the seek time is calculated as: $= (199-50) + (199-16) = 332$ Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

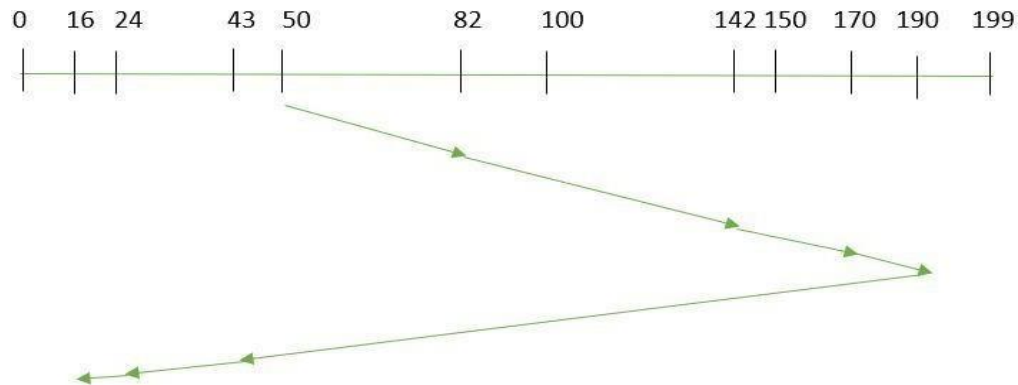
- Long waiting time for requests for locations just visited by disk arm

LOOK: It is similar to the SCAN disk-scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus, it prevents the extra delay that occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82, 170,43,140,24,16,190. In addition, the Read/Write arm is at 50, and it is given that the disk arm should move “towards the larger value”.

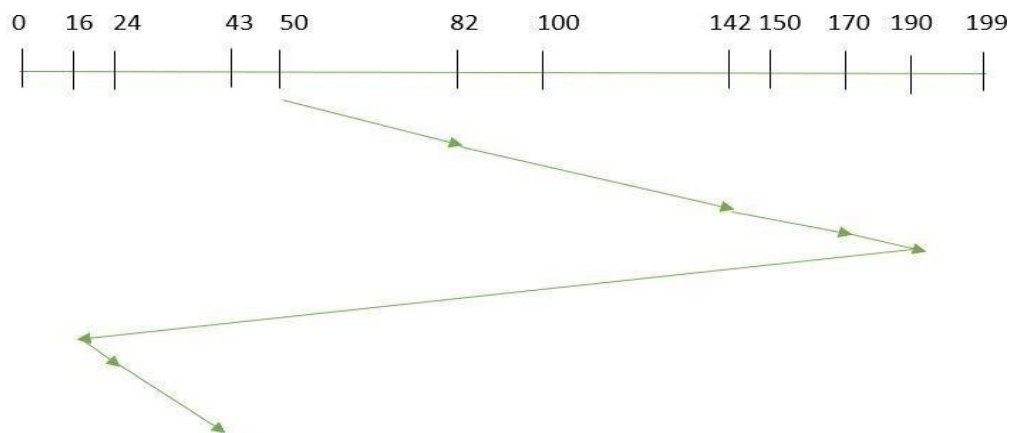
So, the seek time is calculated as: $= (190-50) + (190-16) = 314$



CLOOK: As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay, which occurred due to unnecessary traversal to the end of the disk.

Example:

Suppose the requests to be addressed are-82, 170,43,140,24,16,190. Moreover, the Read/Write arm is at 50, and it is given that the disk arm should move “**towards the larger value**”



So, the seek time is calculated as: $= (190-50) + (190-16) + (43-16) = 34$

CONCLUSION:

Thus, we have implemented Disk scheduling Algorithm using 'C' in Linux.

A	P	C/W	TOTAL	SIGN
(3)	(4)	(3)	(10)	

A – Attendance, P – Performance , C/W – Completion & Writing

FAQ:

1. What is dead lock?
2. What is disk scheduling?
3. Why disk scheduling required?
4. What is storage management?
5. What is partitioning?