Marathwada Mitra Mandal's

COLLEGE OF ENGINEERING

Karvenagar, PUNE – 411 052

Accredited with "A++" Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)
Recipient of "Best College Award 2019" by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

LABORATORY MANUAL

TE (INFORMATION TECHNOLOGY)

(SEMESTER – I)

314446: OPERATING SYSTEMS LABORATORY

2019 Course

'येथे बहुतांचे हित ।'

Marathwada Mitra Mandal's
COLLEGE OF ENGINEERING
Karvenagar, PUNE – 411 052

Accredited with "A++" Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)
Recipient of "Best College Award 2019" by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

Course Code: 314446
Course Name: OPERATING SYSTEMS LABORATORY

Teaching Scheme: **Credits:** 02 **Examination Scheme:**

Practical: 4 hrs/week

TW: 25 Marks

Practical: 25 Marks

Course Outcomes:

Course Outcome	Statement
	At the end of the course, a student will be able to:
CO1	Apply the basics of Linux commands.
CO2	Build shell scripts for various applications.
CO3	Implement basic building blocks like processes, threads under the Linux.
CO4	Develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling in Linux.
CO5	Develop various system programs for the functioning of OS concepts in user space like Memory Management and Disk Scheduling in Linux.
CO6	Develop system programs for Inter Process Communication in Linux.

'येथे बहुतांचे हित ।'

Marathwada Mitra Mandal's

COLLEGE OF ENGINEERING

Karvenagar, PUNE – 411 052

Accredited with "A++" Grade by NAAC // Accredited by NBA (Mechanical Engg. & Electrical Engg.)

Recipient of "Best College Award 2019" by SPPU // Recognized under section 2(f) and 12B of UGC Act 1956

DEPARTMENT OF INFORMATION TECHNOLOGY

Sr. No.	Title of Assignment	PO Mapping	PSO Mapping
1	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	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
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. 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	PO1, PO2, PO3, PO4, PO5, PO12	PSO1

	uses EXECVE system call to load new program which display array in reverse order.		
3	Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2
4	Thread synchronization using counting semaphores. A. 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.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1, PSO2
5	Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
6	Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
7	Inter process communication 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.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1
8	Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.	PO1, PO2, PO3, PO4, PO5, PO12	PSO1

List of Laboratory Assignments
Group A
Assignment No. 1 : **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 **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. **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 which display array in reverse order.

Assignment No. 3:

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

Assignment No. 4:

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.

Assignment No. 5:

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

Assignment No. 6:

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

Assignment No. 7:

Inter process communication 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 onstandard 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.

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.

Study Assignment: Implement a new system call in the kernel space, add this new system call in theLinux kernel by the compilation of this kernel (any kernel source, any architecture and any Linux kernel distribution) and demonstrate the use of this embedded system call using C program in user space.

Reference Books:
1. Das, Sumitabha, UNIX Concepts and Applications, TMH, ISBN-10: 0070635463, ISBN-13: 978-0070635463, 4th Edition. 2. Kay Robbins and Steve Robbins, UNIX Systems Programming, Prentice Hall, ISBN-13: 978-0134424071, ISBN-10: 0134424077, 2nd Edition. 3. Mendel Cooper, Advanced Shell Scripting Guide, Linux Documentation Project, Public domain. 4. Yashwant Kanetkar, UNIX Shell Programming, BPB Publication.

Program Outcomes		
Students are expected to know and be able to–		
PO1	Engineering knowledge	An ability to apply knowledge of mathematics, computing, science, engineering and technology.
PO2	Problem analysis	An ability to define a problem and provide a systematic solutionwith the help of conducting experiments, analyzing the problem and interpreting the data.
PO3	Design / Development ofSolutions	An ability to design, implement, and evaluate software or asoftware /hardware system ,component ,or process to meet desired need switch in realistic constraints.
PO4	Conduct Investigation of Complex Problems	An ability to identify, formulate, and provide essay schematicsolutions to complex engineering /Technology problems.
PO5	Modern Tool Usage	An ability to use the techniques, skills, and modern engineering technology tools, standard processes necessary for practice as a IT professional.
PO6	The Engineer and Society	An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems with necessary constraints andassumptions.
PO7	Environment and Sustainability	An ability to analyze and provide solution for the local and global impact of information technology on individuals, organizations and society.
PO8	Ethics	An ability to understand professional, ethical, legal, security andsocial issues and responsibilities.
PO9	Individual and Team Work	An ability to function effectively as an individual or a sate anmember to accomplish a desired goal(s).
PO10	Communication Skills	An ability to engage in life-long learning and continuing professional development to cope up with fast changes in the technologies /tools with the help of electives, profession along animations and extra-curricular activities.
PO11	Project Management and Finance	An ability to communicate effectively in engineering community at large by means of effective presentations, report writing, paper publications, demonstrations.
PO12	Life-long Learning	An ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessary for professional practice.

COURSE OBJECTIVES:

1. To introduce and learn Linux commands required for administration.
2. To learn shell programming concepts and applications.
3. To demonstrate the functioning of OS basic building blocks like processes, threads under the LINUX.
4. To demonstrate the functioning of OS concepts in user space like concurrency control (process synchronization, mutual exclusion), CPU Scheduling, Memory Management and Disk Scheduling in LINUX.
5. To demonstrate the functioning of Inter Process Communication under LINUX.
6. To study the functioning of OS concepts in kernel space like embedding the system call in any LINUX kernel.

COURSE OUTCOMES:

On completion of the course, students will be able to–

CO1: Apply the basics of Linux commands.

CO2: Build shell scripts for various applications.

CO3: Implement basic building blocks like processes, threads under the Linux.

CO4: Develop various system programs for the functioning of OS concepts in user space like concurrency control, CPU Scheduling in Linux.

CO5: Develop various system programs for the functioning of OS concepts in user space like Memory Management and Disk Scheduling in Linux.

CO6: Develop system programs for Inter Process Communication in Linux.

Assignment No. 1A

Aim: Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

Theory:

echo is one of the most commonly and widely used built-in command for Linux bash and C shells, that typically used in scripting language and batch files to display a line of text/string on standard output or a file.

The syntax for echo is:

echo [option(s)] [string(s)]

1. Input a line of text and display on standard output

$ echo Tecmint is a community of Linux Nerds

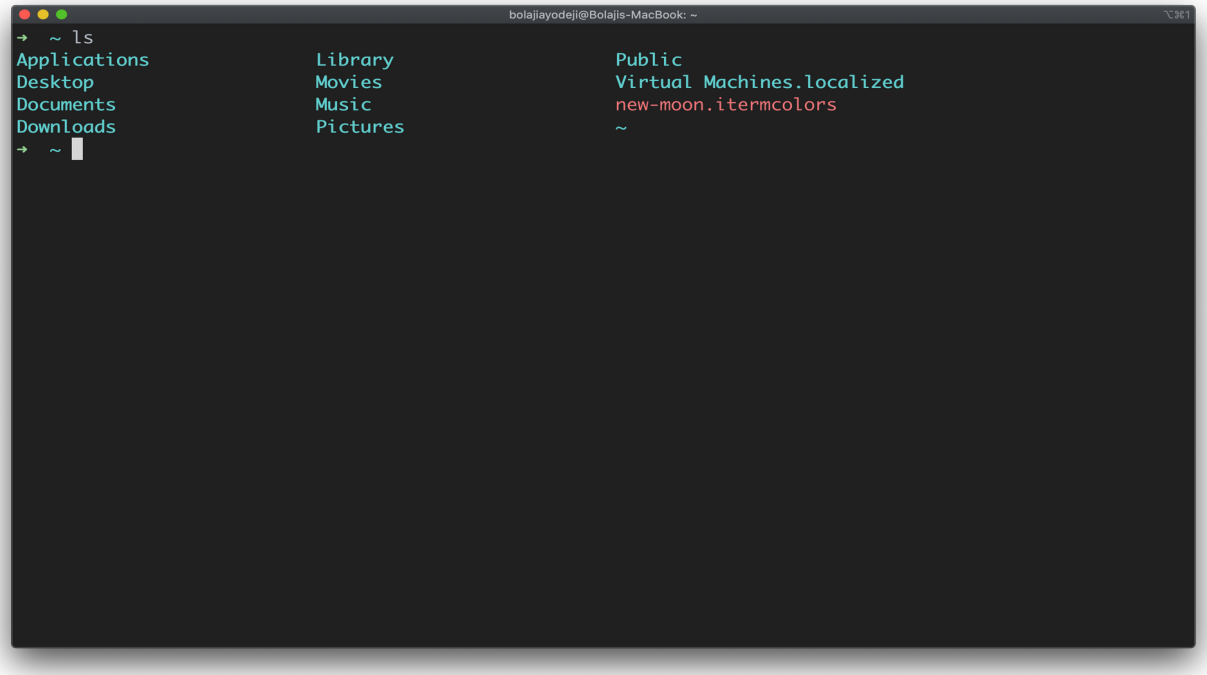
Outputs the following text:

Tecmint is a community of Linux Nerds

The ls command is used to list files or directories in Linux and other Unix-based operating systems.

Just like you navigate in your *File explorer* or *Finder* with a GUI, the ls command allows you to list all files or directories in the current directory by default, and further interact with them via the command line.

Launch your terminal and type ls to see this in action:



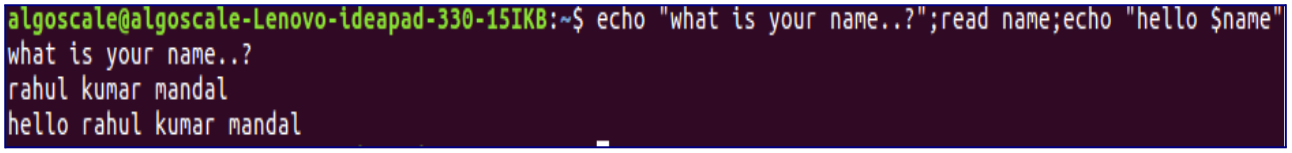
read command in Linux system is used to read from a file descriptor. Basically, this command read up the total number of bytes from the specified file descriptor into the buffer. If the number or count is zero then this command may detect the errors. But on success, it returns the number of bytes read.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

Syntax:

read

In the following example we are acquiring the user's name and then showing the user's name with a greeting.

echo "what is your name..?";read name;echo "hello $name"



The cat (short for "concatenate") command is one of the most frequently used command in Linux/Unix like operating systems. cat command allows us to create single or multiple files, view contain of file, concatenate files and redirect output in terminal or files. In this article, we are going to find out handy use of cat commands with their examples in Linux.

General Syntax

cat [OPTION] [FILE]...

1. Display Contents of File

In the below example, it will show contents of **/etc/passwd** file.

 # cat /etc/passwd

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
narad:x:500:500::/home/narad:/bin/bash

2. View Contents of Multiple Files in terminal

In below example, it will display contents of **test** and **test1** file in terminal.

 # cat test test1

Hello everybody
Hi world,

3. Create a File with Cat Command

We will create a file called **test2** file with below command.

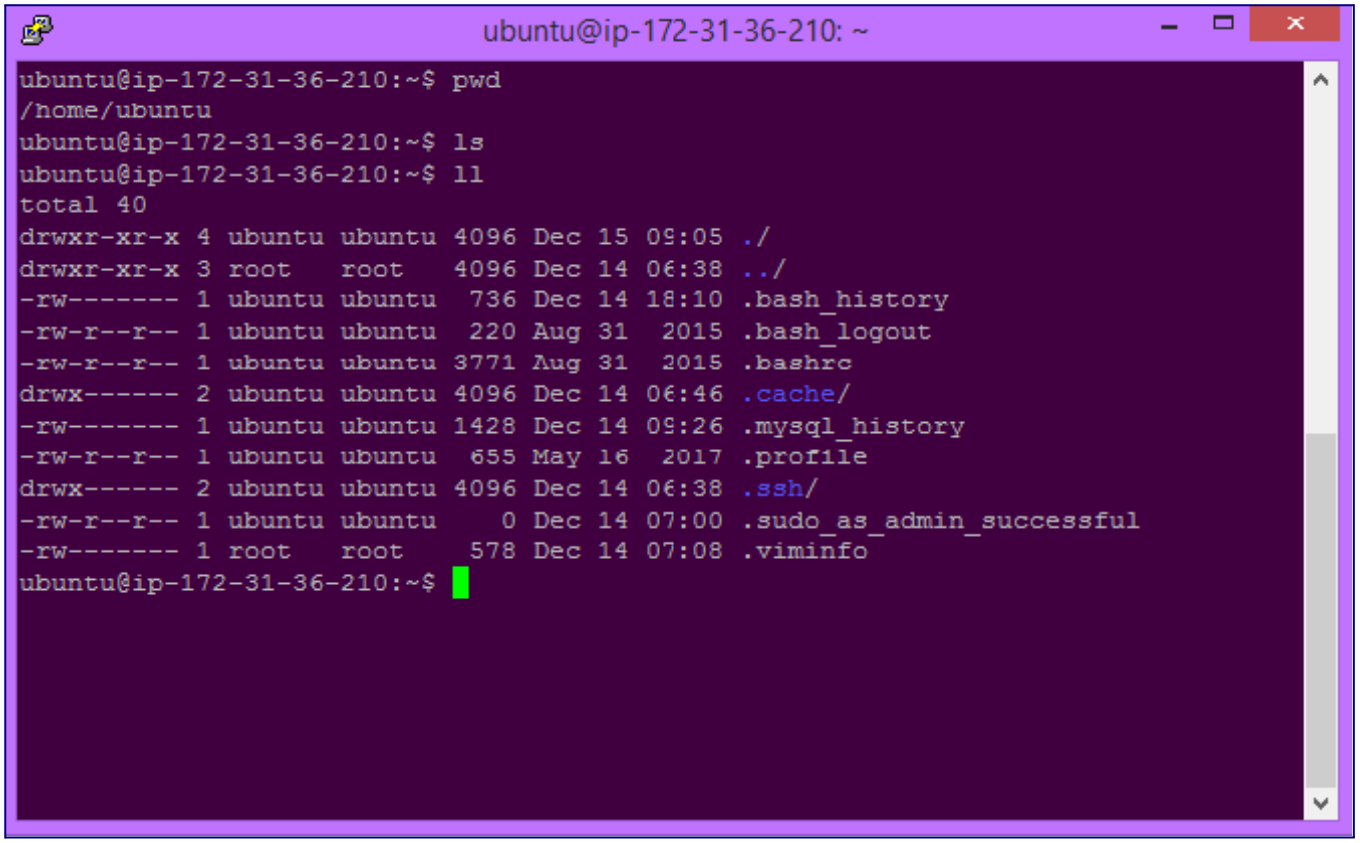
 # cat >test2

Awaits input from user, type desired text and press **CTRL+D** (hold down **Ctrl Key** and type '**d**') to exit. The text will be written in **test2** file. You can see content of file with following **cat** command.

touch command: It is used to create a file without any content. The file created using touch command is empty. This command can be used when the user doesn't have data to store at the time of file creation.

Using touch Command

Initially, we are in home directory and this can be checked using the *pwd* command. Checking the existing files using command **ls** and then long listing command(ll) is used to gather more details about existing files. As you can see in the below figure there is no existing files.



Touch command to create multiple files: Touch command can be used to create the multiple numbers of files at the same time. These files would be empty while creation.

Syntax:

touch File1_name File2_name File3_name

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*]

Expressions

Expressions take the following forms:

(*EXPRESSION* **)**	*EXPRESSION* is true
! *EXPRESSION*	*EXPRESSION* is false

EXPRESSION1 **-a** *EXPRESSION2*	both *EXPRESSION1* and *EXPRESSION2* are true
EXPRESSION1 **-o** *EXPRESSION2*	either *EXPRESSION1* or *EXPRESSION2* is true
-n *STRING*	the length of *STRING* is nonzero
STRING	equivalent to **-n** *STRING*

Output: We have studied basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.

Assignment No. 1B

Aim: 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

Theory:

View contents of the phone book. Display the contents of the phone book preceeding each entry with a line number for easy reference.

1. *Add new entries to the phone book.* Alway add the entries to the end of the book.
2. *Find specific entries in the book.* Ask the user for a name or number to search for, then find that entry.
3. *Delete entries from the book.* Ask he user which line to delete, then remove that line.

```
#!/bin/sh
    # File: /home/john/work/bayer/training/add.sh
    # Date: Wed Feb 27 18:06:30 EST 2002
    # Last Revised: Time-stamp: <2002-02-27 18:45:45 john> maintained by emacs
    # Description: Add entries to the address book
    #
================================================================================

    # Name of address book
    BOOK="address-book.txt"

    # Ask the user for a name and assign to a variable
    echo -n "Name of person: "
    read name

    # Ask the user for a phone number and assign to a variable
    echo -n "Phone number: "
    read phone

    # Echo the answers and ask for confirmation
    echo "Should I enter the values:"
    echo -e "$name ; $phone \n"
    echo -n "y/n: "
    read answer

    if [ "$answer" == "y" ]
    then
       # Write the values to the address book
       echo "$name ; $phone" >>$BOOK
    else
       # Give the user a message
       echo "$name ; $phone NOT written to $BOOK"
    fi
```

```
exit 0
```

eg.

```
opt=1
while [ "$opt" -lt 6 ]
do

        echo -e "Choose one of the Following\n1. Create a New Address Book\n2. View Records\n3. Insert new Record\n4. Delete a Record\n5. Modify a Record\n6. Exit"
        # echo -e, enables special features of echo to use \n \t \b etc.
        read opt
        case $opt in

        1)
                echo "Enter filename"
                read fileName
                if [ -e $fileName ] ; then        # -e to check if file exists, if exits remove the file
                        rm $fileName
                fi
                cont=1
                echo "NAME\tNUMBER\t\tADDRESS\n==============================\n" | cat >> $fileName
                while [ "$cont" -gt 0 ]
                do
                        echo "\nEnter Name"
                        read name
                        echo "Enter Phone Number of $name"
                        read number
                        echo "Enter Address of $name"
                        read address
```

```
            echo "$name\t$number\t\t$address" | cat >> $fileName
            echo "Enter 0 to Stop, 1 to Enter next"
            read cont
        done
        ;;

2)
        cat $fileName
        ;;
3)
        echo "\nEnter Name"
        read name
        echo "Enter Phone Number of $name"
        read number
        echo "Enter Address of $name"
        read address
        echo "$name\t$number\t\t$address" | cat >> $fileName
        ;;
4)
        echo "Delete record\nEnter Name/Phone Number"
        read pattern
        temp="temp"
        grep -v $pattern $fileName | cat >> $temp
        rm $fileName
        cat $temp | cat >> $fileName
        rm $temp
        ;;
5)
        echo "Modify record\nEnter Name/Phone Number"
        read pattern
        temp="temp"
```

```
                grep -v $pattern $fileName | cat >> $temp
                rm $fileName
                cat $temp | cat >> $fileName
                rm $temp
                echo "Enter Name"
                read name
                echo "Enter Phone Number of $name"
                read number
                echo "Enter Address of $name"
                read address
                echo -e "$name\t$number\t$address" | cat >> $fileName
                ;;
        esac
done
```

Output:

```
[fedora@localhost OSL]$ sh assignment01.sh
Creation of Database :

        *******MENU********
        1.Create a File
        2.Add a Record
        3.delete a Record
        4.Search a Record
        5.display database
Enter your choice :
1
Enter file name
database.txt

    File is already exits !!
Do u want to overwrite ? (y/n) :
```

FAQ:

1. Explain Chmod, Grep, Cat & Sort Command with example.
2. Explain sed command with example.
3. Write shell script for sorting a given list of numbers.

Assignment No. 2A

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

Theory:

Implementation Details:

PART A:

Algorithm:

1: START
2: Accept array of integers from the user
3: Call Merge_sort() to sort the array of integers
4: Call fork() to create child
5: Using wait system call, wait for child to finish execution
6: if (pid == 0) i.e. if child process, then call quick_sort() to sort array using quick sort.
7: END

Input:

1: Array of Integers

Steps to execute:

Step 1: Compile file using ―gcc program_name.c –o program_name.out‖
Step 2: Execute the program using ./program_name.out
Step 3: Open 2 new terminals and using command ps –e –o pid, ppid, stat, command show various states of the processes that are created & terminated from the program (use sleep and wait functions as per the requirement to show zombie and orphan states).

PART B:

Algorithm:

File1.c

1: START
2: Accept array of integers from the user
3: Sort the array of integers using any sorting technique
4: Convert the integer numbers into string using sprintf(char * str, const char * format, ...)
5: Call the fork() function to create child process.
6: Using wait system call, wait for child to finish execution.
7: Call the execve() system call, and pass the second program name and the string converted array as parameters to the execve system call.
8: END

File2.c

1: START
2: Convert the received string into integer array using atoi() function.
3: Ask the user which number to find
4: Call binary search function to search the particular number.
5: Return the location of the number if found, else print error.
6: END

Input:

1: Array of Integers
2: Number to find

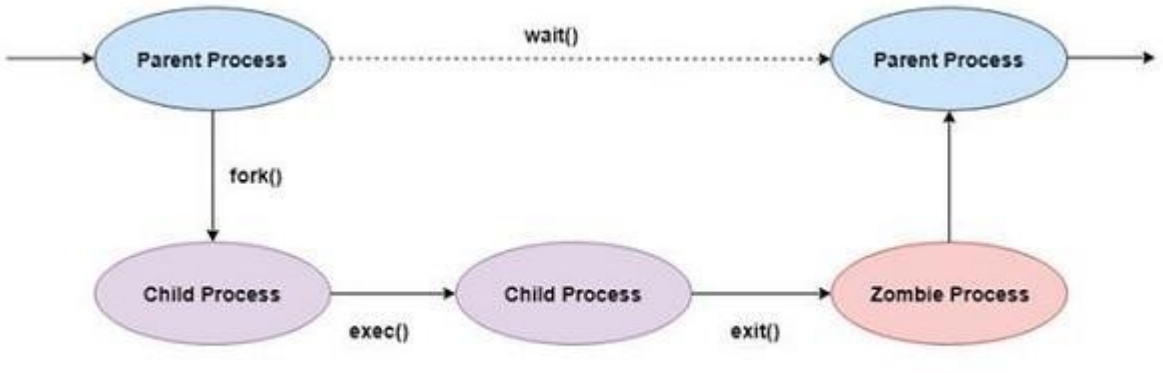
Steps to execute:

Step 1: Compile file1 & file2 using —gcc program_name.c –o program_name.out‖
Step 2: Execute the program using —./file1.out file2.out

Zombie Processes

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process.

A diagram that demonstrates the creation and termination of a zombie process is given as follows



Zombie Process in Linux

Zombie processes don't use any system resources but they do retain their process ID. If there are a lot of zombie processes, then all the available process ID's are monopolized by them. This prevents other processes from running as there are no process ID's available.

Orphan Processes

Orphan processes are those processes that are still running even though their parent process has terminated or finished. A process can be orphaned intentionally or unintentionally.

An intentionally orphaned process runs in the background without any manual support. This is usually done to start an indefinitely running service or to complete a long-running job without user attention.

An unintentionally orphaned process is created when its parent process crashes or terminates. Unintentional orphan processes can be avoided using the process group mechanism.

Fork() System Call

When a parent process uses fork(), it creates a duplicate copy of itself and this duplicates becomes the child of the process. The fork() is implemented using clone() system call in linux which returns twice from kernel.

A non-zero value(Process ID of child) is returned to the parent.
A value of zero is returned to the child.

In case the child is not created successfully due to any issues like low memory, -1 is returned to the fork().

Let's understand this with some code

```
pid);

// Both child and parent will now start execution from here.
if(pid < 0) {
   //child was not created successfully
   return 1;
}
else if(pid == 0) {
   // This is the child process
   // Child process code goes here
}
else {
   // Parent process code goes here
}
printf("This is code common to parent and child");
```

How Parent and Child Process differs?

A parent and child differs in some of the PCB(process control block) attributes. These are:

1. PID - Both child and parent have a different Process ID.
2. Pending Signals - The child doesn't inherit Parent's pending signals. It will be empty for the child process when created.
3. Memory Locks - The child doesn't inherit its parent's memory locks. Memory locks are locks which can be used to lock a memory area and then this memory area cannot be swapped to disk.
4. Record Locks - The child doesn't inherit its parent's record locks. Record locks are associated with a file block or an entire file.
5. Process resource utilisation and CPU time consumed is set to zero for the child.
6. The child also doesn't inherit timers from the parent.

Understanding process creation in operating system with fork, exec and wait system calls

A process begins its life when it is created. A process goes through different states before it gets terminated. The first state that any process goes through is the creation of itself. Process creation happens through the use of fork() system call, which creates a new process(child process) by duplicating an existing one(parent process). The process that calls fork() is the parent, whereas the new process is the child.

In most cases, we may want to execute a different program in child process than the parent. The exec() family of function calls creates a new address space and loads a new program into it.

Finally, a process exits or terminates using the exit() system call. This function frees all the resources held by the process(except for pcb). A parent process can enquire about the status of a terminated child using wait() system call. When the parent process uses wait() system call, the parent process is blocked till the child on which it is waiting terminates.

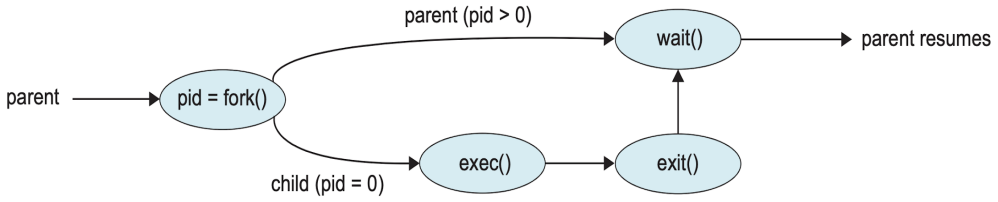


Figure 3.10 Process creation using the `fork()` system call.

If you find reading boring, you can watch process creation videos to understand the below concepts.

Exec System Call

The exec family of functions replaces the current running program(executable) with a new executable. This is very useful when you want the child process to run a different program than the parent. Take example of the shell program. A shell takes commands as input from the user, runs it and displays the output on screen. Each command corresponds to a new program. The number of shell commands in any operating system is huge. It would be very bad code design to append the program or executable associated with all the available commands in a single program. In this case, the shell(parent process) can create a child process when a user types a command and let the child process execute the command.

Wait System Call

A process may wait on another process to complete its execution. The parent process may issue a wait system call, which suspends the execution of the parent process while the child executes. When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.
When the child terminates, the kernel notifies this to the waiting parent using signal(SIGCHLD) and allowing it to retrieve the child process's exit status.
In the above code example used for exec() system call, the shell(parent process) was waiting for its child to complete. Once the child terminates, it can return the output and exit code of the command to the user.

Exit System Call

A computer process terminates its execution by making an exit system call. When the child process terminates ("dies"), either normally by calling exit, or abnormally due to a fatal exception or signal (e.g., SIGTERM, SIGINT, SIGKILL), an exit status is returned to the operating system and a SIGCHLD signal is sent to the parent process. The exit status can then be retrieved by the parent process via the wait system call.

FAQ's

1. What is a system call? Explain briefly execve system call.
2. List and explain different states of a process.
3. What is a process? Explain how Process is represented internally.
4. What is orphan process?

Assignment No. 2B

Aim: 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 which display array in reverse order.

Theory:

execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized
stack, heap, and (initialized and uninitialized) data segments.
pathname must be either a binary executable, or a script starting with a line of the form:

#!*interpreter* [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., *argv[0]*) should contain the filename associated with the file being executed. The *argv* array must be terminated by a NULL pointer. (Thus, in the new program, *argv[argc]* will be NULL.)

envp is an array of pointers to strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The *envp* array must be terminated by a NULL pointer.

The argument vector and environment can be accessed by the new program's main function, when it is defined as:

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

Note, however, that the use of a third argument to the main function is not specified in POSIX.1; according to POSIX.1, the environment should be accessed via the external variable environ(7).

execve() does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

If the current program is being ptraced, a **SIGTRAP** signal is sent to it after a successful **execve**().

If the set-user-ID bit is set on the program file referred to by *pathname*, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, if the set-group-ID bit is set on the program file, then the effective group ID of the calling process is set to the group of the program file.

The aforementioned transformations of the effective IDs are *not* performed (i.e., the set-user-ID and set-group-ID bits are ignored) if any of the following is true:

* the *no_new_privs* attribute is set for the calling thread (see prctl(2));

* the underlying filesystem is mounted *nosuid* (the **MS_NOSUID** flag for mount(2)); or

* the calling process is being ptraced.

How Parent and Child Process differs?

A parent and child differs in some of the PCB(process control block) attributes. These are:

1. PID - Both child and parent have a different Process ID.
2. Pending Signals - The child doesn't inherit Parent's pending signals. It will be empty for the child process when created.
3. Memory Locks - The child doesn't inherit its parent's memory locks. Memory locks are locks which can be used to lock a memory area and then this memory area cannot be swapped to disk.
4. Record Locks - The child doesn't inherit its parent's record locks. Record locks are associated with a file block or an entire file.
5. Process resource utilisation and CPU time consumed is set to zero for the child.
6. The child also doesn't inherit timers from the parent.

Fork() System Call

When a parent process uses fork(), it creates a duplicate copy of itself and this duplicates becomes the child of the process. The fork() is implemented using clone() system call in linux which returns twice from kernel.

A non-zero value(Process ID of child) is returned to the parent.
A value of zero is returned to the child.
In case the child is not created successfully due to any issues like low memory, -1 is returned to the fork().

Let's understand this with some code

```
pid = fork();

// Both child and parent will now start execution from here.

if(pid < 0) {

   //child was not created successfully

   return 1;

}

else if(pid == 0) {

   // This is the child process
```

```
    // Child process code goes here
}
else {
    // Parent process code goes here
}
printf("This is code common to parent and child");
```

Output

```
admin@ll17:~$ ~/Desktop/os/1$ gcc os1b1.c -o p
admin@ll17:~$ ~/Desktop/os/1$ gcc os1b2.c -o p1
admin@ll17:~$ ~/Desktop/os/1$ ./p ./p1

Enter number of Elements:6
2
4
1
2
6
7

Sorted Array :122467
Eneter the element to be serached:4
Elemets  found

FAQ:
1. What is EXECVE system call?
2. Explain WAIT system call.
3. Describe FORK system call
```

Assignment No. 3

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

Theory:

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

Characteristics of SJF Scheduling

It is associated with each job as a unit of time to complete.

This algorithm method is helpful for batch-type processing, where waiting for jobs to complete is not critical.

It can improve process throughput by making sure that shorter jobs are executed first, hence possibly have a short turnaround time.

It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time.

Preemptive SJF

In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Example of Preemptive SJF Scheduling: In the following example, we have 4 processes with process ID P1, P2, P3, and P4. The arrival time and burst time of the processes are given in the following table.

Process	Burst time	Arrival time	Completion time	Turnaround time	Waiting time
P1	18	0	31	31	13
P2	4	1	5	4	0
P3	7	2	14	12	5
P4	2	3	7	4	2

The waiting time and turnaround time are calculated with the help of the following formula.

Waiting Time = Turnaround time – Burst Time

Turnaround Time = Completion time – Arrival time

Process waiting time:

P1=31-18=13

P2=4-4=0

P3=12-7=5

P4=4-2=2

Average waiting time= 13+0+5+2/4

=20

Process Turnaround Time:

P1=31-0=31

P2=5-1=4

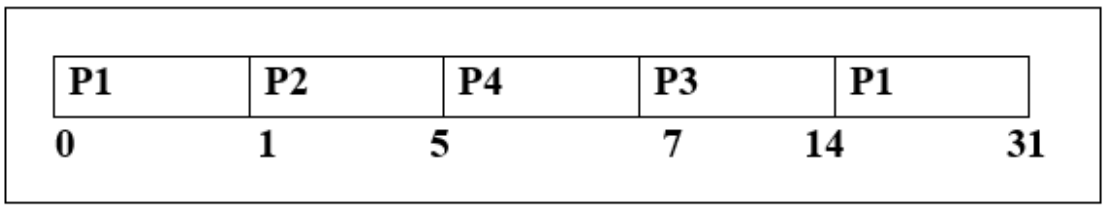
P3=14-2=12

P4=7-3=4

Average turnaround time= 31+4+12+4/4

=12.75

Gantt Chart:



Implementation:

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]= Completion time - arrival_time-burst_time
   f)Increment time lap by one.
2- Find turnaround time (waiting_time+burst_time).

Output:

```
Processes  Burst time  Waiting time  Turn around time
 1         6          3          9
 2         8          16          24
 3         7          8          15
 4         3          0          3
Average waiting time = 6.75
Average turn around time = 12.75
```

Round-robin scheduling algorithm is used to schedule process fairly each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which is arrived in the quantum time that makes these scheduling fairly.

Advantages:

Each process is served by CPU for a fixed time, so priority is the same for each one
Starvation does not occur because of its cyclic nature.

Round Robin CPU Scheduling Algorithm

Step 1: Organize all processes according to their arrival time in the ready queue. The queue structure of the ready queue is based on the FIFO structure to execute all CPU processes.

Step 2: Now, we push the first process from the ready queue to execute its task for a fixed time, allocated by each process that arrives in the queue.

Step 3: If the process cannot complete their task within defined time interval or slots because it is stopped by another process that pushes from the ready queue to execute their task due to arrival time of the next process is reached. Therefore, CPU saved the previous state of the process, which helps to resume from the point where it is interrupted. (If the burst time of the process is left, push the process end of the ready queue).

Step 4: Similarly, the scheduler selects another process from the ready queue to execute its tasks. When a process finishes its task within time slots, the process will not go for further execution because the process's burst time is finished.

Step 5: Similarly, we repeat all the steps to execute the process until the work has finished.

Characteristics of Round Robin

1. It is a pre-emptive algorithm.
2. It shares an equal time interval between all processes to complete their task.
3. It is a starvation free CPU scheduling algorithm. Hence it is known as the fairest and simple algorithm.

How to implement in a programming language

1. Declare arrival[], burst[], wait[], turn[] arrays and initialize them. Also declare a timer variable and initialize it to zero. To sustain the original burst array create another array (temp_burst[]) and copy all the values of burst array in it.

2. To keep a check we create another array of bool type which keeps the record of whether a process is completed or not. we also need to maintain a queue array which contains the process indicies (initially the array is filled with 0).

3. Now we increment the timer variable until the first process arrives and when it does, we add the process index to the queue array

4. Now we execute the first process until the time quanta and during that time quanta, we check whether any other process has arrived or not and if it has then we add the index in the queue (by calling the fxn. queueUpdation()).

5. Now, after doing the above steps if a process has finished, we store its exit time and execute the next process in the queue array. Else, we move the currently executed process at the end of the queue (by calling another fxn. queueMaintainence()) when the time slice expires.

6. The above steps are then repeated until all the processes have been completely executed. If a scenario arises where there are some processes left but they have not arrived yet, then we shall wait and the CPU will remain idle during this interval.

Output:

Enter the time quanta : 2

Enter the number of processess : 4

Enter the arrival time of the processess : 0 1 2 3

Enter the burst time of the processess : 5 4 2 1

Program No.	Arrival Time	Burst Time	Wait Time	TurnAround Time
1	0	5	7	12
2	1	4	6	10
3	2	2	2	4
4	3	1	5	6

Average wait time : 5
Average Turn Around Time : 8

FAQ:

1. Whar are advantages and disadvantages of SJF?

2. What is time quantum in RR?

3. Which scheduling algorithm is best?

Assignment No. 4A

Aim: Thread synchronization using counting semaphores. Application to demonstrate: producer-consumer problem with counting semaphores and mutex.

Theory:

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.

A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal().

The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

```
wait(S)
{
while(S<=0);   // busy waiting
S--;
}

signal(S){
S++;
}
```

Semaphores are of two types:

1. Binary Semaphore – This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. “Full” keeps track of number of items in the buffer at any given time and “Empty” keeps track of number of unoccupied slots.

Initialization of semaphores –
mutex = 1
Full = 0 // Initially, all slots are empty. Thus full slots are 0
Empty = n // All slots are empty initially

Solution for Producer –
do{

//produce an item

wait(empty);

wait(mutex);

//place in buffer

signal(mutex);

signal(full);

}while(true)

When producer produces an item then the value of “empty” is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of “full” is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –
do{

wait(full);

wait(mutex);

// remove item from buffer

signal(mutex);

signal(empty);

// consumes item

}while(true)

As the consumer is removing an item from buffer, therefore the value of "full" is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of "empty" by 1. The value of mutex is also increased so that producer can access the buffer now.

Solution in C using Semaphore and Mutex

We will be converting the above Pseudocode to actual code in C language. Let's first have a look at some important data structures we will be using in the code.

One can use include

Some important methods that can be used with semaphore in c

1. sem_init -> Initialise the semaphore to some initial value
2. sem_wait -> Same as wait() operation
3. sem_post -> Same as Signal() operation
4. sem_destroy -> Destroy the semaphore to avoid memory leak

One can use include

Some important methods that can be used with semaphore in c

1. pthread_mutex_init -> Initialise the mutex
2. pthread_mutex_lock() -> Same as wait() operation
3. pthread_mutex_unlock() -> Same as Signal() operation
4. pthread_mutex_destroy() -> Destroy the mutex to avoid memory leak

The following is the pseudo-code for the producer:

```
void producer() {
   while(T) {
      produce()
      wait(E)
      wait(S)
      append()
      signal(S)
      signal(F)
   }
}
```

FAQ

1. What is semaphore?

2. What is mutex?

3. Which are different types of semaphore?

Assignment No. 4B

Aim: Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.

Theory:

What is readers writers problem?
Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers-writers problem.

Consider a situation where we have a file shared between many people.

If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.

However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**
Problem parameters:

One set of data is shared among a number of processes

Once a writer is ready, it performs its write. Only one writer may write at a time

If a process is writing, no other process can read it

If at least one reader is reading, no other process can write

Readers may not write and only read

Solution when Reader has the Priority over Writer
Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:

   it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.

   It then, signals mutex as any other reader is allowed to enter while others are already reading.

   After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.
3. If not allowed, it keeps on waiting.

```
do {

  // Reader wants to enter the critical section
  wait(mutex);

  // The number of readers has now increased by 1
  readcnt++;

  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);
```

```
  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);         // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the semaphore '**wrt**' is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

FAQ:

1. What is Process Synchronization?
2. What is mutual exclusion?
3. What is wait()?
4. What is signal() ?

Assignment No. 5

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

Theory:

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.

Available :

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

Max :

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

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.
Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

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

for its execution.

Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.
Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:
1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both
a) Finish[i] = false
b) $Need_i$ <= 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 $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$ <= 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 Pi by modifying the state as
follows:
Available = Available – Requesti
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$

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:

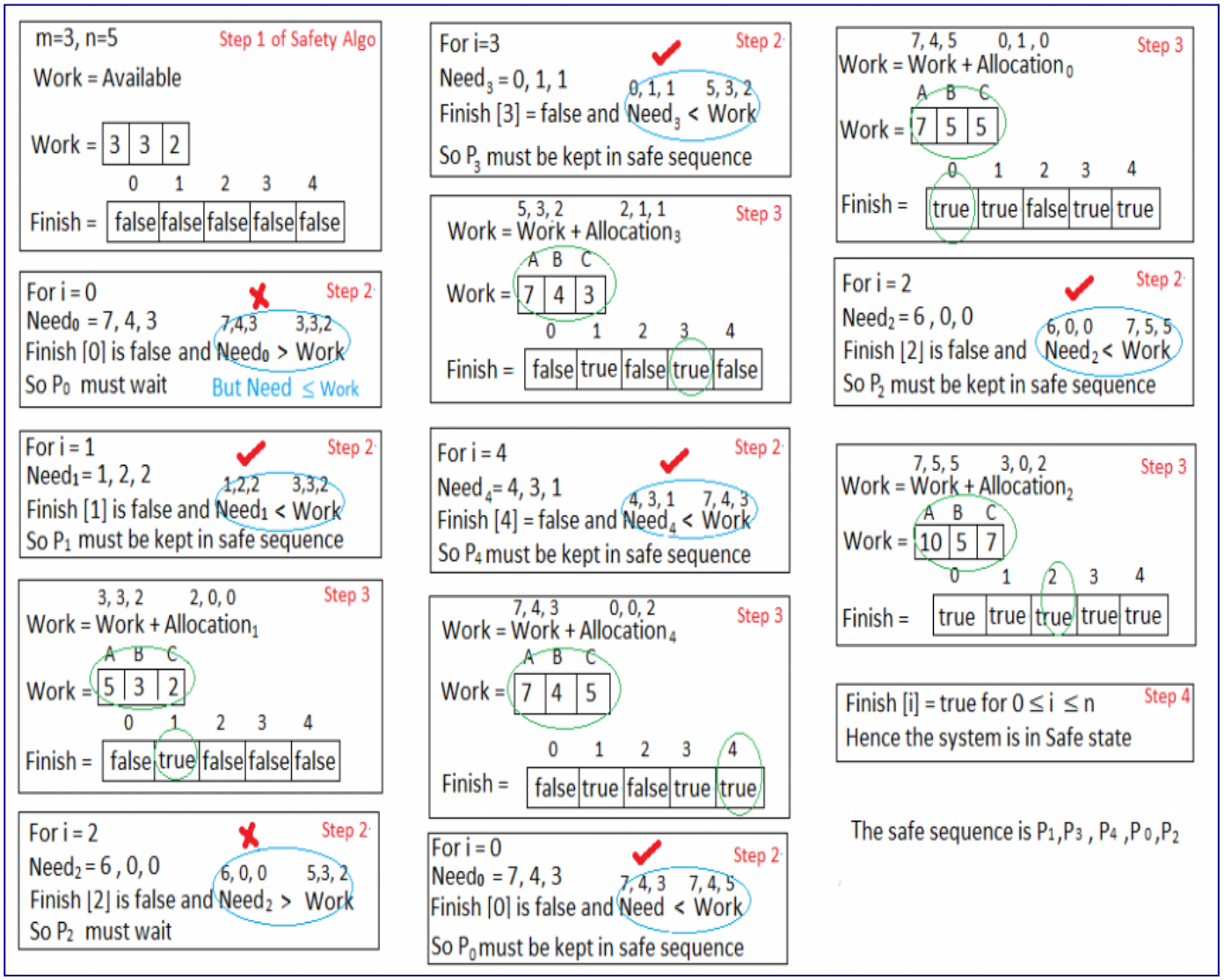
Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

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

Is the system in a safe state? If Yes, then what is the safe sequence?
Applying the Safety algorithm on the given system,



ALGORITHM:

1. Start .
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.
9. or not if we allow the request.
10. stop .

OUTPUT:

Enter the no of processes 5
Enter the no of resources instances 3
Enter the max matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the allocation matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter available resources 3 2 2
P1->p3->p4->p2->p0
The system is in safe state.

FAQ:

1. What is deadlock?
2. Which are Conditions for Deadlock?
3. What is circular wait?

Assignment No. 6

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

Theory:

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

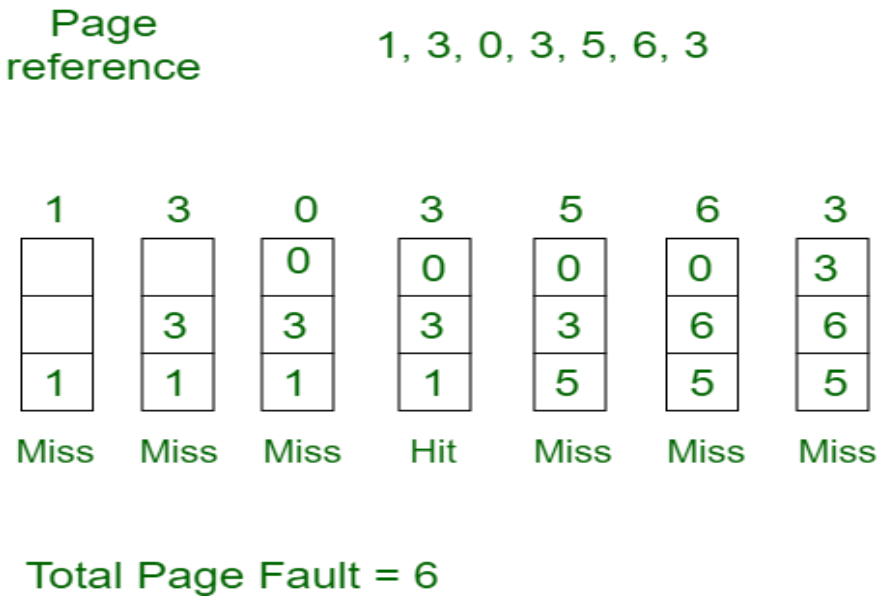
Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to 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.

Page Replacement Algorithms :

1. First In First Out (FIFO) –

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example-1 Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames.Find number of page faults.



Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults.
when 3 comes, it is already in memory so —> 0 Page Faults.

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault.
6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.
Finally when 3 come it is not available so it replaces 0 1 page fault

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

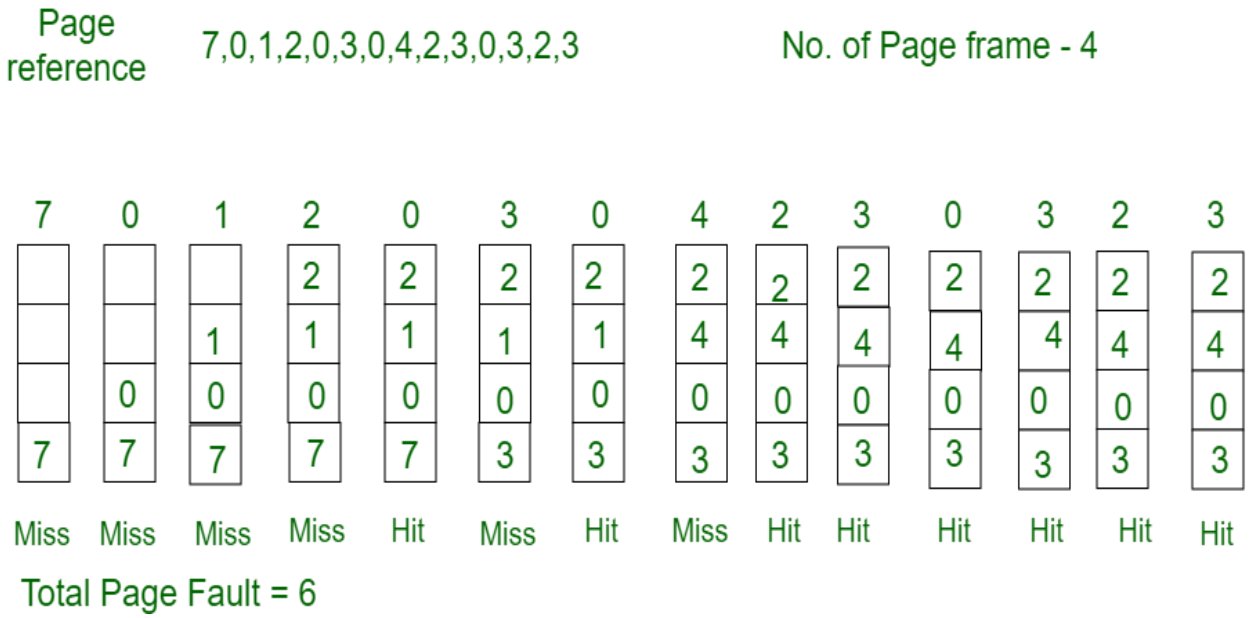
Implementation – Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.
i) If set holds less pages than capacity.
a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
b) Simultaneously maintain the pages in the queue to perform FIFO.
c) Increment page fault
ii) Else
If current page is present in set, do nothing.
Else
a) Remove the first page from the queue as it was the first to be entered in the memory
b) Replace the first page in the queue with the current page in the string.
c) Store current page in the queue.
d) Increment page faults.

2. Return page faults.

2. Optimal Page replacement –

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.
Example-2:Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

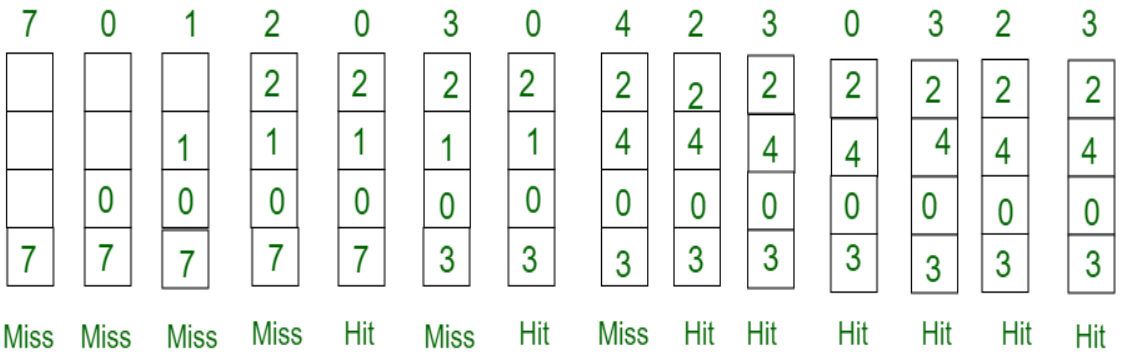


Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**
0 is already there so —> **0 Page fault.**.
4 will takes place of 1 —> **1 Page Fault.**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

3. Least Recently Used –
In this algorithm page will be replaced which is least recently used.
Example-3 Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames.Find number of page faults.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already their so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Let capacity be the number of pages that memory can hold. Let **set** be the current set of pages in memory.

1- Start traversing the pages.
i) **If set holds less pages than capacity.**
a) Insert page into the set one by one until the size of **set** reaches **capacity** or all page requests are processed.
b) Simultaneously maintain the recent occurred index of each page in a map called **indexes**.
c) Increment page fault
ii) **Else**
If current page is present in **set**, do nothing.
Else
a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.

2. Return page faults.

Output

Enter number of frames: 3

Enter number of pages: 10

Enter page reference string: 2 3 4 2 1 3 7 5 4 3

Number of Page fault:

FAQ :

1. What is page fault?

2. Which page replacement algorithm is best?

3. What is **Belady's anomaly?**

Assignment No. 7A

Aim: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 onstandard output.

Theory:

n computing, a named pipe (also known as a FIFO) is one of the methods for inter-process communication.

It is an extension to the traditional pipe concept on Unix. A traditional pipe is "unnamed" and lasts only as long as the process.

A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Creating a FIFO file: In order to create a FIFO file, a function calls i.e. mkfifo is used.

mkfifo() makes a FIFO special file with name *pathname*. Here *mode* specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).
Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. *open*, *read*, *write*, *close*.
Example Programs to illustrate the named pipe: There are two programs that use the same FIFO.

FIFO Special File:
FIFO special file is similar to a pipe, except that it is created in a different way. FIFO special file is entered into the file system by calling mkfifo function.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file.

However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa

How To Create FIFO

Fifo are created using the function mkfifo() which takes as arguments

•€€€€€€€ The name of the fifo that has to be created

•€€€€€€€ The permissions for the file.

Once the file is created, it needs to be opened using the system call open() and the data can be read and written from the file using read() and write system calls.

The mkfifo function is declared in the header file sys/stat.h.

Function:

int mkfifo (const char *filename, mode_t mode)

The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the file's permissions;

The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned

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);
```

Crating My First FIFO

```
#include<unistd.h>

#include<stdlib.h>

#include<stdio.h>
```

```
#include<sys/types.h>
#include<sys/stat.h>
intmain()
{
   int res = mkfifo("my_fifo", 0777);
   if (res == 0)
      printf("FIFO created\n");
   exit(0);
}
```

Eg.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
      int fd,fd1;
      char * myfifo = "myfifo";
      char * myfifo1 = "myfifo1";
      char buf[1024];

      mkfifo(myfifo, 0666);
      mkfifo(myfifo1, 0777);

      fd = open(myfifo, O_WRONLY);
      write(fd, "Hello friends... \nWelcome...", sizeof("Hello friends... \nWelcome..."));
      close(fd);

     unlink(myfifo);

       fd1 = open(myfifo1, O_RDONLY);
      read(fd1, buf, sizeof(buf));
   printf("%s",buf);
   unlink(myfifo1);
      close(fd1);
      return 0;
}
```

Execution steps:
Save the above program as create_fifo.c and compile is using gcc as follows
cc create_fifo.c -o create_fifo

If the compilation is successfull, run the program as follows

./create_fifo

Output:
Enter the String:hello my name is Huzaif(Professional Cipher) the string ends with (hash sign) #

The contents of file are hello my name is Huzaif(Professional Cipher) the string ends with (hash sign)

No of Words: 13
No of Characters: 66
No of Lines: 1

FAQ

1. How To Create FIFO?
2. What is the use of function mkfifo()?
3. What is named pipe?

Assignment No. 7B

Aim: 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.

Theory:

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

Server reads from the input file.

The server writes this data in a message using either a pipe, fifo or message queue.

The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.

Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

ftok(): is use to generate a unique key.

shmget(): int shmget(key_t,size_tsize,intshmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself to it using shmat(). void *shmat(int shmid ,void *shmaddr ,int shmflg);
shmid is shared memory id. shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

shmdt(): When you're done with the shared memory segment, your program should detach itself from it using shmdt(). int shmdt(void *shmaddr);

shmctl(): when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used. shmctl(int shmid,IPC_RMID,NULL);

SHARED MEMORY FOR WRITER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;
```

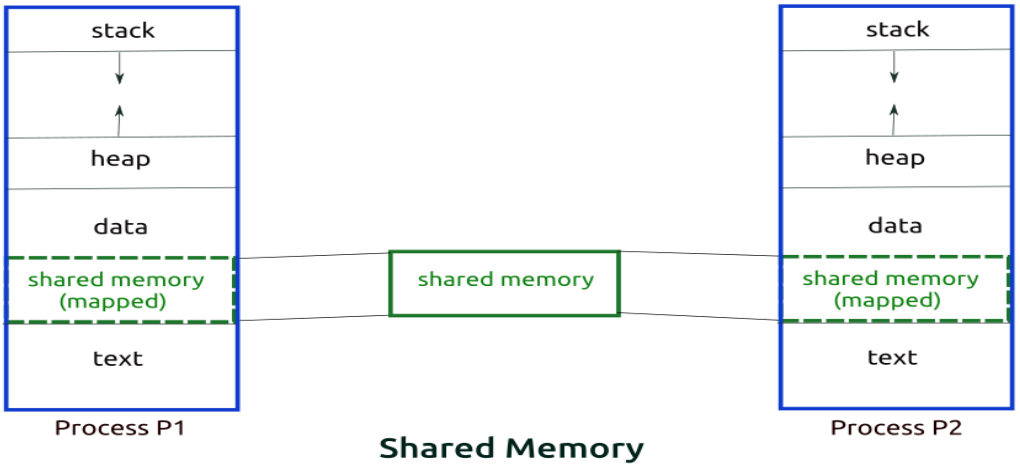
```
int main()
{
   // ftok to generate unique key
   key_t key = ftok("shmfile",65);

   // shmget returns an identifier in shmid
   int shmid = shmget(key,1024,0666|IPC_CREAT);

   // shmat to attach to shared memory
   char *str = (char*) shmat(shmid,(void*)0,0);

   cout<<"Write Data : ";
   gets(str);

   printf("Data written in memory: %s\n",str);

   //detach from shared memory
   shmdt(str);

   return 0;
}
```

SHARED MEMORY FOR READER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
   // ftok to generate unique key
   key_t key = ftok("shmfile",65);

   // shmget returns an identifier in shmid
   int shmid = shmget(key,1024,0666|IPC_CREAT);

   // shmat to attach to shared memory
   char *str = (char*) shmat(shmid,(void*)0,0);

   printf("Data read from memory: %s\n",str);

   //detach from shared memory
   shmdt(str);

   // destroy the shared memory
   shmctl(shmid,IPC_RMID,NULL);

   return 0;
}
```

Shared memory is one of the three interprocess 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 interprocess 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 interprocess communication.

shmget

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

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

As the name suggests, shmget gets you a shared memory segment associated with the given *key*. The key is obtained earlier using the ftok function. If there is no existing shared memory segment corresponding to the given *key* and IPC_CREAT flag is specified in *shmflg*, a new shared memory segment is created. Also, the *key* value could be IPC_PRIVATE, in which case a new shared memory segment is created. *size* specifies the size of the shared memory segment to be created; it is rounded up to a multiple of PAGE_SIZE. If *shmflg* has IPC_CREAT | IPC_EXCL specified and a shared memory segment for the given key exists, shmget fails and returns -1, with errno set to EEXIST. The last nine bits of *shmflg* specify the permissions granted to owner, group and others. The execute permissions are not used. If shmget succeeds, a shared memory identifier is returned.

On error, -1 is returned and errno is set to the relevant error.

shmat

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

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

With shmat, the calling process can attach the shared memory segment identified by *shmid*. The process can specify the address at which the shared memory segment should be attached with *shmaddr*. However, in most cases, we do not care at what address system attaches the shared memory segment and *shmaddr* can conveniently be specified as NULL. *shmflg* specifies the flags for attaching the shared memory segment. If *shmaddr* is not null and SHM_RND is specified in *shmflg*, the shared memory segment is attached at address rounded down to the nearest multiple of SHMLBA, where SHMLBA stands for Segment low boundary address. The idea is to attach at an address which is a multiple of SHMLBA. On most Linux systems, SHMLBA is the same as PAGE_SIZE. Another flag is SHM_RDONLY, which means the shared memory segment should be attached with read only access.

shmdt

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

int shmdt (const void *shmaddr);

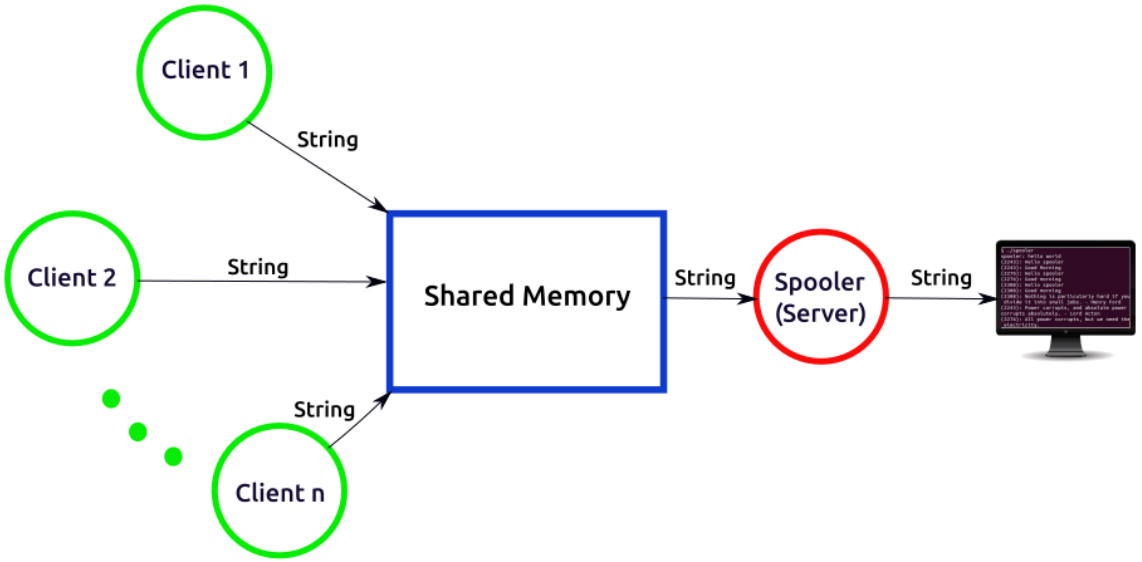
shmdt detaches a shared memory segment from the address space of the calling process. *shmaddr* is the address at which the shared memory segment was attached, being the value returned by an earlier shmat call. On success, shmdt returns 0. On error, shmdt returns -1 and errno is set to indicate the reason of error.

shmctl

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

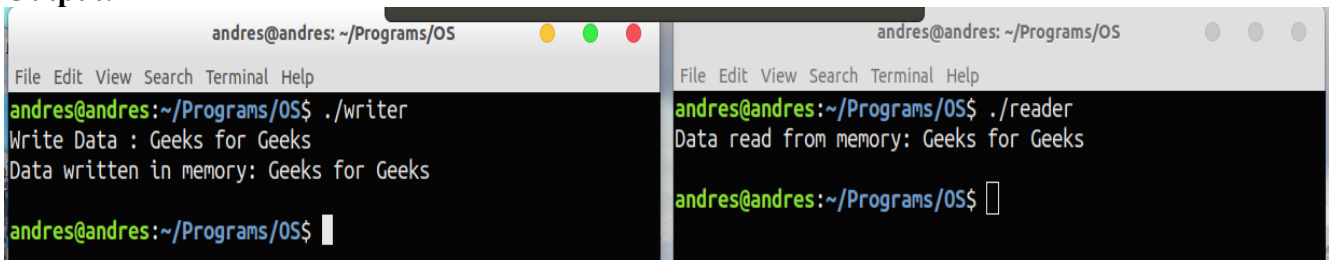
int shmctl (int shmid, int cmd, struct shmid_ds *buf);

The shmctl call is for control operations of a System V shared memory segment identified by the *shmid* identifier, returned by an earlier shmget call. The data structure for shared memory segments in the kernel is,



Software Architecture for Clients and Server using Shared Memory

Output:



FAQ :

1. What is IPC?
2. Explain shared memory.
3. Descrite shmat.

Assignment No 8

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

Theory:

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:

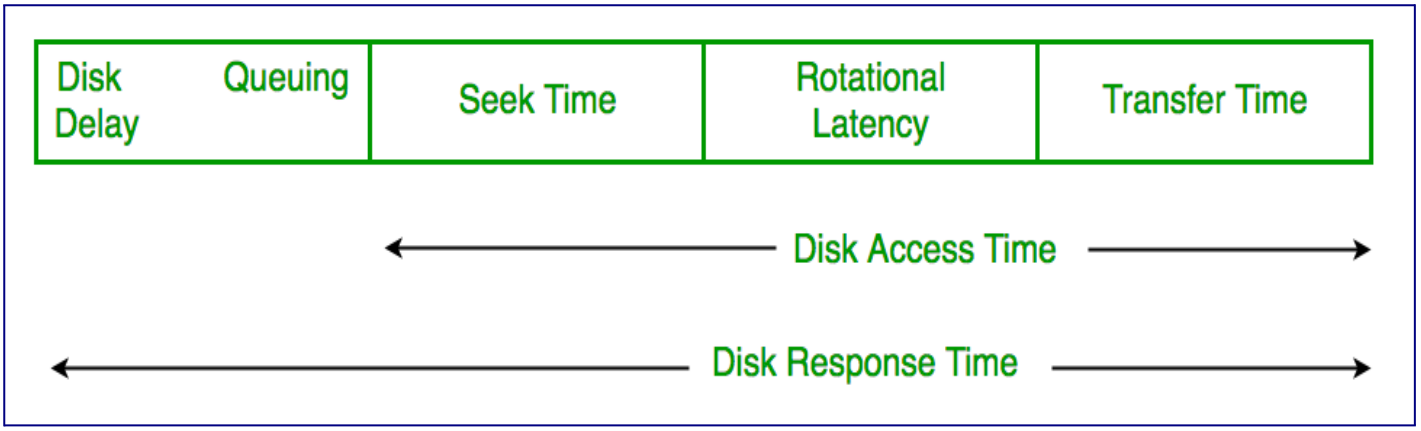
Seek Time: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.

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. So the disk scheduling algorithm that gives minimum variance response time is better.

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested tracks if Shortest Seek Time First (SSTF) is a disk scheduling algorithm is used.

Shortest Seek Time First (SSTF) –

Basic idea is the tracks which are closer to current disk head position should be serviced first in order to *minimise the seek operations*.

Advantages of Shortest Seek Time First (SSTF) –

1. Better performance than FCFS scheduling algorithm.
2. It provides better throughput.
3. This algorithm is used in Batch Processing system where throughput is more important.
4. It has less average response and waiting time.

Disadvantages of Shortest Seek Time First (SSTF) –

1. Starvation is possible for some requests as it favours easy to reach request and ignores the far away processes.
2. Their is lack of predictability because of high variance of response time.
3. Switching direction slows things down.

Algorithm –

1. Let Request array represents an array storing indexes of tracks that have been requested. 'head' is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.

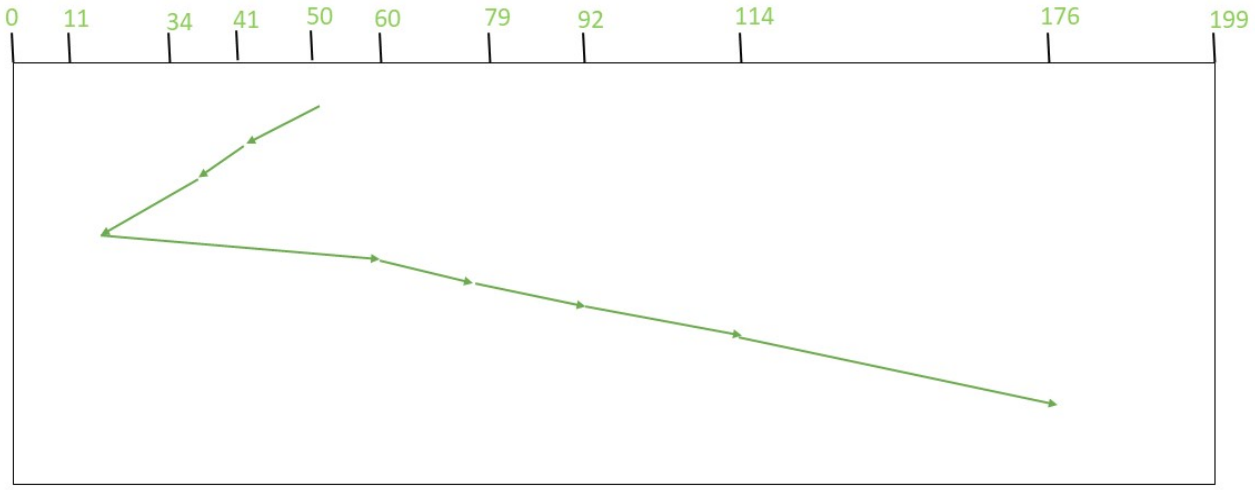
6. Go to step 2 until all tracks in request array have not been serviced.

Example –

Request sequence = {176, 79, 34, 60, 92, 11, 41, 114}

Initial head position = 50

The following chart shows the sequence in which requested tracks are serviced using SSTF.



Therefore, total seek count is calculates as:

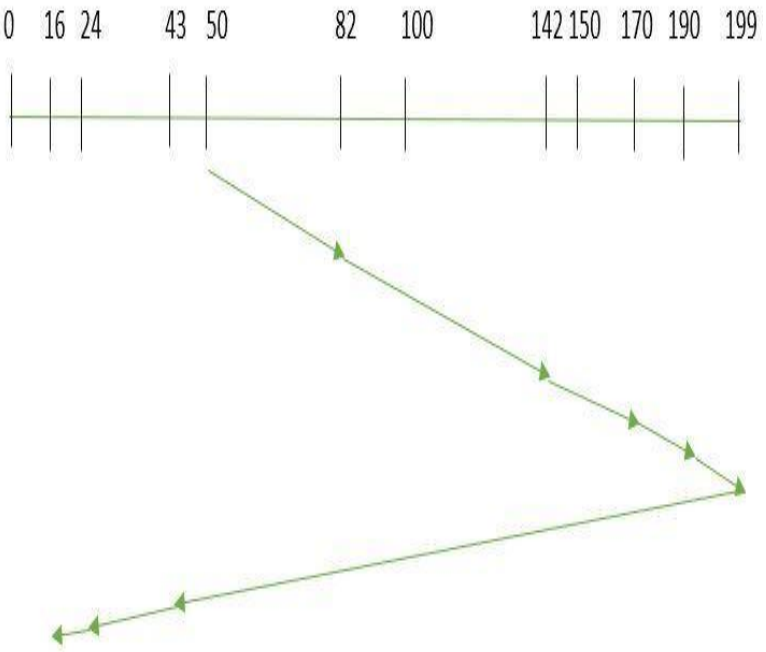
= (50-41)+(41-34)+(34-11)+(60-11)+(79-60)+(92-79)+(114-92)+(176-114)

= 204

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. And the Read/Write arm is at 50, and it is also 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

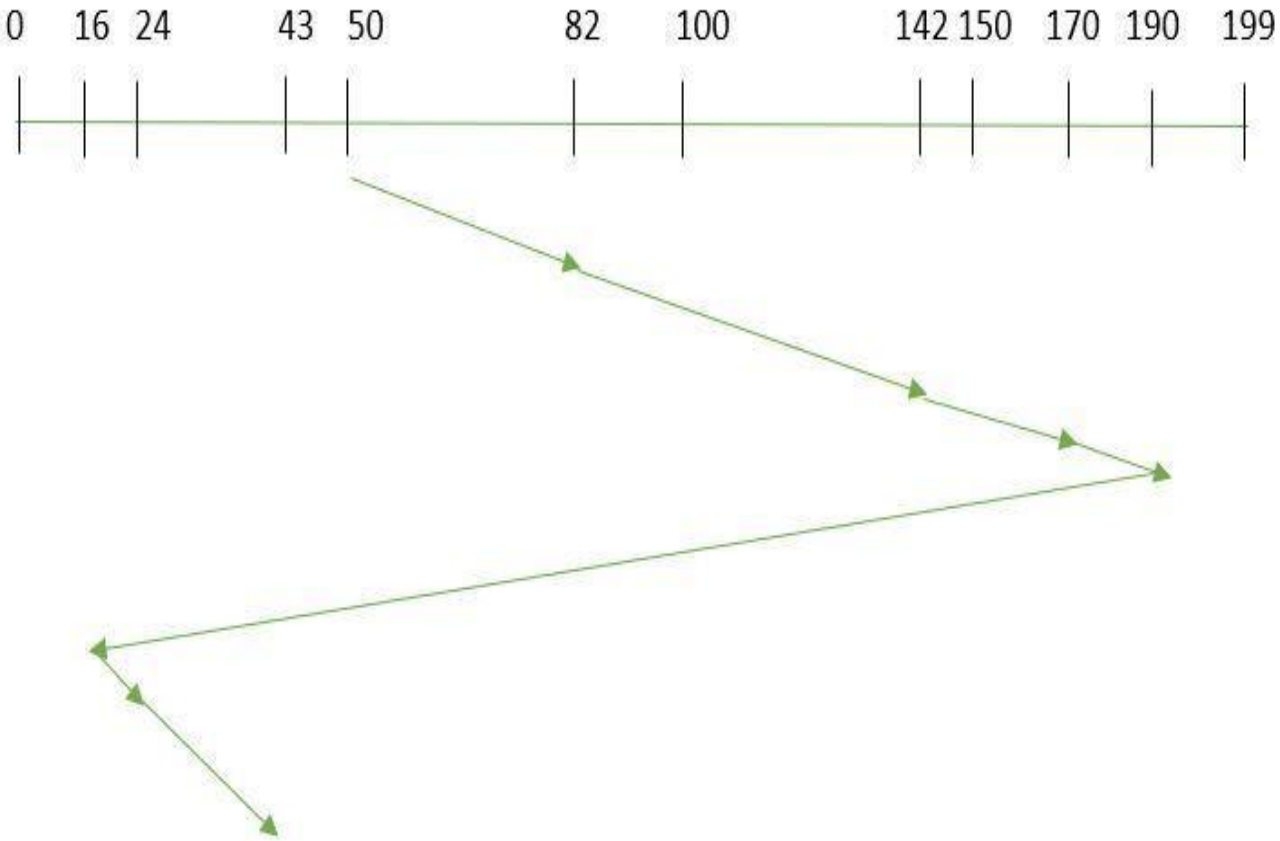
Disadvantages:

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

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. And the Read/Write arm is at 50, and it is also given that the disk arm should move "**towards the larger value"**



So, the seek time is calculated as:

=(190-50)+(190-16)+(43-16)
=341

Output:
Total number of seek operations = 204
Seek Sequence is
50
41
34
11
60
79
92
114
176

FAQ:

1. What is seek time?

2. What is difference between LOOK and SCAN Scheduling?

3. What is rotational latency?

Assignment No. 8B

Study Assignment: Implement a new system call in the kernel space, add this new system call in the Linux

kernel by the compilation of this kernel (any kernelsource, any architecture and any Linux kernel

distribution) and demonstrate the use of this embedded system call using C program in user space.

Theory:

1. Setup environment

I prefer to get root and apply the steps as root to prevent getting permission failures on the way. If you don't want to apply the steps as a root, you can just use sudo for root required commands.

su -

will switch to root user after entering the password of the root. Then, first you can check the active kernel version of your OS with:

uname -r

This prints out 4.19.0-6-amd64 in my case. Then, we need to get the source of a kernel. I will use slightly newer version (4.20.1) from my version with following wget command.

wget https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/linux-4.20.1.tar.xz

This will download the source of 4.20.1. It may take a while, like 2-3 minutes depending on your internet speed.

Extract the compressed kernel code with

tar -xvf linux-4.20.1.tar.xz

It will create a folder named linux-4.20.1.tar.xz and extract the compressed code into that folder.

Now we will change our directory to new kernel code.

cd linux-4.20.1

2. Add "Hello world" syscall to kernel

I prefer creating new folder for my own stuff while adding a new syscall.

mkdir hello && cd hello

After that, I will create a C file for my syscall implementation. I prefer vim hello.c to create and edit

the file and insert following C code.

```
#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
        //printk prints to the kernel's log file.
        printk("Hello world\n");
        return 0;
}
```

We need to create a Makefile in the hello directory.

vim Makefile

and then insert this:

obj-y := hello.o

Then, go to the parent directory (kernel source main directory):

cd ..

We need to add our new syscall directory to Makefile, in this way it will compile our syscall, too. To achieve this, search for core-y in the Makefile then, find the

In vim you can do search with /core-y *after pressing ESC.*

core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/

line and add hello/ to the end of this line.

As a result, the line should be looking like this:

core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/

Next step is adding the new system call to the system call table.
If you are on a 32-bit system you'll need to change syscall_32.tbl file.

vim arch/x86/entry/syscalls/syscall_32.tbl

For 64-bit, change syscall_64.tbl.

vim arch/x86/entry/syscalls/syscall_64.tbl

I will continue with my 64-bit OS and my steps will be accordingly.

We need to keep table's structure while adding our syscall. So that we will add our line to the end of the line. My last syscall has number of 547, so that I will use 548, you also should use N+1.

My syscall data:

548 64 hello sys_hello

Example file:

```
546     ...     ...     ...
547     ...     ...     ...
548     64      hello   sys_hello
```

Now, we need to add our syscall method signature to syscalls header file which is syscalls.h.

vim include/linux/syscalls.h

Then, add the following line to the end of the document before the #endif statement:

asmlinkage long sys_hello(void);

3. Compiling our kernel

Before starting to compile you need to install a few packages. Type the following commands in your terminal to install required packages:

```
apt-get install gcc &&
apt-get install libncurses5-dev &&
apt-get install bison &&
apt-get install flex &&
apt-get install libssl-dev &&
apt-get install libelf-dev &&
apt-get update &&
apt-get upgrade &&
apt-get make
```

Now, you can configure your kernel by using the config menu by executing:

make menuconfig

IMPORTANT NOTE: While entering this command, you might want to maximize your terminal screen. Otherwise, you might get and error and a pop-up screen will not appear.

Once the above command is used to configure the Linux kernel, you will get a pop up window with the list of
menus and you can select the items for the new configuration. If your unfamiliar with the configuration just check for the file systems menu and check whether “ext4” is chosen or not, if not select it and save the configuration.

OR: You can simply execute the following command and use the default configuration.

make defconfig

Now, finally we can compile the kernel with the following command:

sudo make

The compilation took 40 mins to 1 hour on my uni-core VM. As an alternative solution you can increase the core count of your VM in the VM settings and use them to compilation with the -jn

parameter. n is the number of cores dedicated to your VM.

In my case, I've used the following ocmmand for 8-core VM which reduced the compilation time to 20 seconds to 1 minute:

make -j8

4. Installing our kernel

After the successful compilation, to install/update kernel use the following command:

make modules_install install

The command will create some files under /boot/ directory and it will automatically make a entry in your grub.cfg.
To check whether it made correct entry, check the files under /boot/ directory . If you have followed the steps without any error you will find the following files in it in addition to others.

5. Testing our syscall in new kernel

Now all we need to do is restart the system:

shutdown -r now

After computer restarts, in grub's advanced options, you can see there are 2 options (without counting the recovery mode options).

Once your computer is up again, you can run the following command the check your kernel version:

uname -r

which prints 4.20.1 in my OS after installing the kernel.

After checking the version of the kernel, we will test our syscall with tiny C program.

vim hello_test.c

Insert the following C code into the hello_test.c file:

```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

int main()
{
	long int helloCheck = syscall(548);
	printf("System call sys_hello returned %ld\n", helloCheck);
	return 0;
}
```

Then compile and run it:

gcc hello_test.c -o hello.o && ./hello.o

Output: We have implemented a new system call in the kernel space, add this new system call in the Linux kernel by the compilation of this kernel (any kernelsource, any architecture and any Linux kernel distribution) and demonstrate the use of this embedded system call using C program in user space.