AMRUTVAHINI COLLEGE OF ENGINEERING, SANGAMNER

DEPARTMENT OF INFORMATION TECHNOLOGY ENGINEERING

LABORATORY MANUAL

314446 : OPERATING SYSTEM LAB

Sr. No.		Description
I.		**Institute and Department Vision, Mission, Quality Policy, Quality Objectives, PEOs, POs and PSOs**
II.		**List of Experiments**
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
2	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.
3		Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.
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.
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	A)	Inter process communication in Linux using following:- 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.

8	Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.
9	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.

Vision and Mission of the Institute

Vision

To create opportunities for rural students to become able engineers and technocrats through continual excellence in engineering education.

Mission

Our mission is to create self-disciplined, physically fit, mentally robust and morally strong engineers and technocrats with high degree of integrity and sense of purpose who are capable to meet challenges of ever advancing technology for the benefit of mankind and nature. We, the management, the faculty and staff, therefore promise to strive hard and commit ourselves to achieve this objective through a continuous process of learning and appreciation of needs of time.

Vision and Mission of the Department

Vision

To transfer the rural learners into competent I.T. engineers and technocrats in emerging areas of I.T. Engineering education through continual excellence for the benefit of society.

Mission

Mission of Information Technology Department is elaborated in following three mission statements.

1. M1: To empower the youths in rural communities to be self-disciplined, physically fit, mentally robust and morally strong I.T. professionals.

2. M2. To provide cutting-edge technical knowledge through continuous process in rapidly changing environment as per need of industry and surrounding world.

3. M3: To provide opportunities for intellectual and personal growth of individuals in rural platform using high quality Information Technology education.

AMRUTVAHINI COLLEGE OF ENGINEERING, SANGAMNER

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

To train the Information Technology students-

1. To develop competent I.T. graduate with knowledge of fundamental concepts In mathematics, science, engineering and ability to provide solution to complex engineering problem by analyzing , designing and designing using modern I.T. software and tools.

2. To prepare I.T. graduate with professional skills of better communication, teamwork to manage projects in I.T. field at global level and ability to conduct investigations of complex problems using research based knowledge and research methods.

3. To develop I.T graduates with ethical practices, societal contributions through communities understanding impact of professional engineering solutions in societal and environmental context and ability of lifelong learning.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

1. Apply principles of science, mathematics along with programming paradigms and problem solving skills using appropriate tools, techniques to expedite solution in I.T. domain.

2. Demonstrate core competencies related to I.T. in domain of Data structures & algorithms, Software Engineering & Modeling, Hardware, Distributed Computing, Networking & security, Databases, Discrete mathematics & algebra, Machine Learning, Operating System.

3. Demonstrate leadership qualities and professional skills in modern I.T. platform for creating innovative carrier paths in placements, entrepreneurship and higher studies

AMRUTVAHINI COLLEGE OF ENGINEERING, SANGAMNER

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

- **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
- **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

QUALITY POLICY

THE AMRUTVAHINI COLLEGE OF ENGINEERING IS COMMITTED TO DEVELOP IN YOUNG MINDS THE STATE – OF – THE – ART TECHNOLOGY AND HIGH ACADEMIC AMBIENCE BY SYNERGISING SPIRITUAL VALUES AND TECHNOLOGICAL COMPETENCE CONTINUALLY IN A LEARNING ENVIRONMENT.

QUALITY OBJECTIVES

- To strive hard for academic excellence and synergizing spiritual & moral values.
- To improve overall development of student.
- To enhance industry-institute interaction.
- To provide assistance for placement & entrepreneurship development.
- To promote and encourage R&D activities.

OPERATING SYSTEM LAB			
Experiment No: 1	**Study of Basic Linux Commands: echo, ls, read, cat, touch, test, loops, arithmetic comparison, conditional loops, grep, sed etc.**	**Page**	**1/5**

Aim: Study of Basic Linux Commands

Apparatus: Ubuntu Operating system, gcc

Theory:

Linux Basic Commands

1. pwd command

Use the pwd command to find out the path of the current working directory (folder) you're in. The command will return an absolute (full) path, which is basically a path of all the directories that starts with a forward slash (/). An example of an absolute path is /home/username.

2. **cd command**

To navigate through the Linux files and directories, use the cd command. It requires either the full path or the name of the directory, depending on the current working directory that you're in. Let's say you're in /home/username/Documents and you want to go to Photos, a subdirectory of Documents. To do so, simply type the following command: cd Photos. Another scenario is if you want to switch to a completely new directory, for example,/home/username/Movies. In this case, you have to type cd followed by the directory's absolute path: cd /home/username/Movies. There are some shortcuts to help you navigate quickly:

cd .. (with two dots) to move one directory up

cd to go straight to the home folder

cd- (with a hyphen) to move to your previous directory On a side note, Linux's shell is case sensitive. So, you have to type the name's directory exactly as it is.

3. **ls command**

The ls command is used to view the contents of a directory. By default, this command will display the contents of your current working directory. If you want to see the content of other directories, type ls and then the directory's path. For example, enter ls /home/username/Documents to view the content of Documents.

There are variations you can use with the ls command:

ls -R will list all the files in the sub-directories as well

ls -a will show the hidden files

ls -al will list the files and directories with detailed information like the

permissions, size, owner, etc.

4**. cat command**

cat (short for concatenate) is one of the most frequently used commands in Linux. It is used to list the contents of a file on the standard output (sdout). To run this command, type cat followed by the file's name and its extension. For instance: cat file.txt. Here are other ways to use the cat command:

cat > filename creates a new file

cat filename1 filename2>filename3 joins two files (1 and 2) and stores the output of them in a new file (3) to convert a file to upper or lower case use, cat filename | tr a-z AZ

>output.txt

5. **cp command**

Use the cp command to copy files from the current directory to a different directory.For instance, the command cp scenery.jpg /home/username/Pictures would create a copy of scenery.jpg (from your current directory) into the Pictures directory.

6. **mv command**

The primary use of the mv command is to move files, although it can also be used to rename files.The arguments in mv are similar to the cp command. You need to type mv, the file's name, and the destination's directory. For example: mv file.txt /home/username/Documents. To rename files, the Linux command is mv oldname.ext newname.ext

7. **mkdir command**

Use mkdir command to make a new directory — if you type mkdir Music it will create a directory called Music. There are extra mkdir commands as well: To generate a new directory inside another directory, use this Linux basic command mkdir Music/Newfile use the p (parents) option to create a directory in between two existing directories. For example, mkdir -p Music/2020/Newfile will create the new "2020" file.

8**. rmdir command**

If you need to delete a directory, use the rmdir command. However, rmdir only allows you to delete empty directories.

9. **rm command**

The rm command is used to delete directories and the contents within them. If you only want to delete the directory — as an alternative to rmdir — use rm -r.

Note: Be very careful with this command and double-check which directory you are in. This will delete everything and there is no undo.

10. **touch command**

The touch command allows you to create a blank new file through the Linux command line. As an example, enter touch /home/username/Documents/Web.html to create an HTML file entitled Web under the Documents directory.

11. **locate command**

You can use this command to locate a file, just like the search command in Windows. What's more, using the -i argument along with this command will make it caseinsensitive, so you can search for a file even if you don't remember its exact name. To search for a file that contains two or more words, use an asterisk (*). For example, locate -i school*note command will search for any file that contains the word "school" and "note", whether it is uppercase or lowercase.

12. **find command**

Similar to the locate command, using find also searches for files and directories. The difference is, you use the find command to locate files within a given directory. As an example, find /home/ -name notes.txt command will search for a file called notes.txt within the home directory and its subdirectories.

Other variations when using the find are:

To find files in the current directory use, find . -name notes.txt

To look for directories use, / -type d -name notes. txt

13. **grep command**

Another basic Linux command that is undoubtedly helpful for everyday use is grep. It lets you search through all the text in a given file.

To illustrate, grep blue notepad.txt will search for the word blue in the notepad file. Lines that contain the searched word will be displayed fully.

14. **sudo command**

Short for "SuperUser Do", this command enables you to perform tasks that require administrative or root permissions. However, it is not advisable to use this command for daily use because it might be easy for an error to occur if you did something wrong.

15. **df command**

Use df command to get a report on the system's disk space usage, shown in percentage and KBs. If you want to see the report in megabytes, type df -m.

16. **du command**

If you want to check how much space a file or a directory takes, the du (Disk Usage) command is the answer. However, the disk usage summary will show disk block numbers instead of the usual size format. If you want to see it in bytes, kilobytes, and megabytes, add the -h argument to the command line.

17. **head command**

The head command is used to view the first lines of any text file. By default, it will show the first ten lines, but you can change this number to your liking. For example, if you only want to show the first five lines, type head -n 5 filename.ext.

18. **tail command**

This one has a similar function to the head command, but instead of showing the first lines, the tail command will display the last ten lines of a text file. For example, tail -n filename.ext.

19. **Diff Command**

Short for difference, the diff command compares the contents of two files line by line. After analyzing the files, it will output the lines that do not match.

20. **Man command**

Confused about the function of certain Linux commands? Don't worry, you can easily learn how to use them right from Linux's shell by using the man command. For instance, entering man tail will show the manual instruction of the tail command.

Observation Table: We understand the function and working of linux commands.

Conclusion: We have studied all linux shell commands

Questions:

1. What is Linux? It is a family of open-source Unix operating systems based on the Linux kernel.
2. Difference between Linux and Unix? ...
3. What is a kernel?
4. What is an interpreter?
5. What is a compiler ?

6. What is a shell?
7. What Greb Command
8. Why cat command use

OPERATING SYSTEM LAB			
Experiment No: 1 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**	**Page**	**1/3**

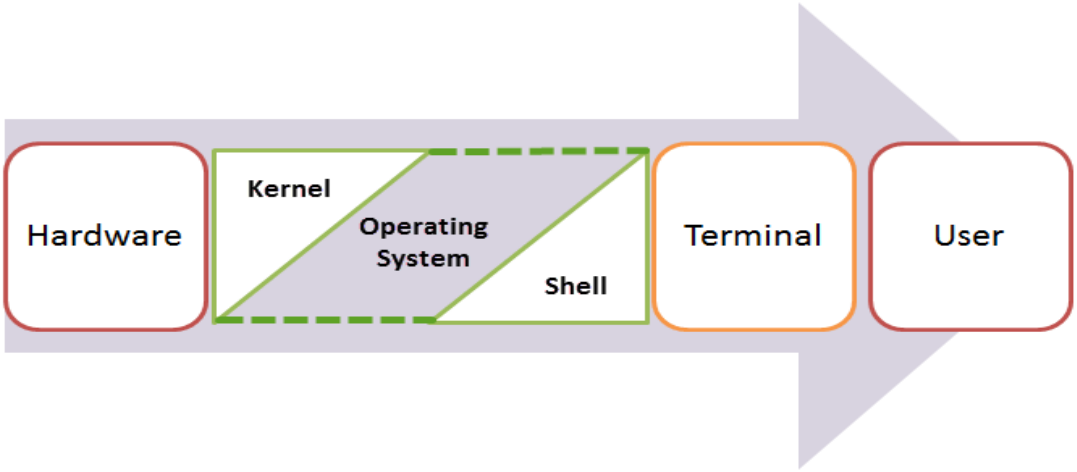
Aim: Shell program for address book

Apparatus: Ubuntu Operating system, gcc

Theory:

SHELL :-An Operating is made of many components but its two prime components are -

- Kernel
- Shell



Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one. A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands and scripts. A shell is accessed by a terminal which runs it. When you run the terminal, the Shell issues **a command prompt (usually $),** where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal. The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name **Shell**.

Types of Shell:-

There are two main shells in Linux:

1. The **Bourne Shell**: The prompt for this shell is $ and its derivatives are listed below:

- POSIX shell also known as sh

- Korn Shell also known as sh
- **B**ourne **A**gain **SH**ell also known as bash (most popular)

2. The C shell: The prompt for this shell is % and its subcategories are:

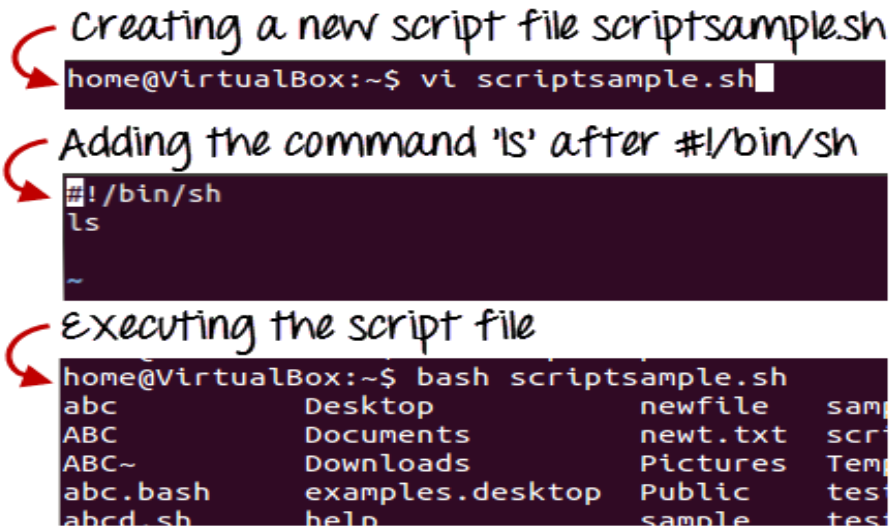
- C shell also known as csh
- Tops C shell also known as tcsh

We will discuss bash shell based shell scripting in this tutorial.

What is Shell scripting and why do We need it?
Writing a series of command for the shell to execute **is called shell scripting**.It can **combine lengthy and repetitive sequences of commands into a single and simple script**, which can be stored and executed anytime. This reduces the effort required by the end user..
Let us understand the steps in creating a Shell Script

1. **Create a fileusing** a **vi** editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**

"#!" is an operator called shebang which directs the script to the interpreter location. So, if we use"#! /bin/sh" the script gets directed to the bourne-shell.
Let create a small script -
#!/bin/sh
ls
Let's see the steps to create it –



Command 'ls' is executed when we execute the scripsample.sh file.

Adding shell comments

Commenting is important in any program. In Shell, the syntax to add a comment is
#comment

File system commands
mkdir – Creates a directory
rmdir – Deletes a directory
ls – Lists contents of given path
cat – Read from given file and output to STDOUT or given path
find – Search for a given file (find <path> -name <filename>)
chmod – Change mode/permissions
cp - Copy files (cp sourcefile destfile)
mv – Move/rename files (mv oldname newname)
scp – Secure copy (Remote file copy) (scp <filename><host>:<path>

I/O Commands
echo – To print to stdout
read – To obtain values from stdin
I/O Redirection
> - Output to given file
< - Read input from given file
>> - Append output to given file

Command Line Arguments
Command line arguments are important part of writing scripts. Command line arguments define the expected input into a shell script. For example,we may want to pass a file name or folder name or some other type of argument to a shell script.
Several special variables exist to help manage command-line arguments to a script:
• $# - represents the total number of arguments (much like argv) – except command
• $0 - represents the name of the script, as invoked
• $1, $2, $3, .., $8, $9 - The first 9 command lineargument s
• $* - all command line arguments OR
• $@ - all command line arguments
Advantages and disadvantages
Perhaps the biggest advantage of writing a shell script is that the commands and syntax are exactly the same as those directly entered at the command-line. The programmer does not have to switch to a totally different syntax, as they would if the script were written in a different language, or if a compiled language were used.

Conclusion: We have studied address book program in shell.

OPERATING SYSTEM LAB			
Experiment No: 1B	**Shell programming:-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.**	**Page**	**4/4**

Often, writing a shell script is much quicker than writing the equivalent code in other programming languages. The many advantages include easy program or file selection, quick start, and interactive debugging. A shell script can be used to provide a sequencing and decision-making linkage around
existing programs, and for moderately sized scripts the absence of a compilation step is an advantage. Interpretive running makes it easy to write debugging code into a script and re-run it to detect and fix bugs. Non-expert users can use scripting to tailor the behavior of programs, and shell scripting provides some limited scope for multiprocessing.
On the other hand, shell scripting is prone to costly errors. Inadvertent typing errors such as rm `-rf * /` (instead of the intended `rm -rf */`) are folklore in the Unix community; a single extra space converts the command from one that deletes everything in the sub-directories to one which deletes everything—and also tries to delete everything in the root directory. Similar problems can transform cp and mv into dangerous weapons, and misuse of the > redirect can delete the contents of a file. This is made more problematic by the fact that many UNIX commands differ in name by only one letter: `cp`, cd, dd, df, etc.
Another significant disadvantage is the slow execution speed and the need to launch a new process for almost every shell command executed. When a script's job can be accomplished by setting up a pipeline in which efficient filter commands perform most of the work, the slowdown is mitigated, but a complex script is typically several orders of magnitude slower than a conventional compiled program that performs an equivalent task.
There are also compatibility problems between different platforms. Larry Wall, creator of Perl, famously wrote that "It is easier to port a shell than a shell script."
Similarly, more complex scripts can run into the limitations of the shell scripting language itself; the limits make it difficult to write quality code, and extensions by various shells to ameliorate problems with the original shell language can make problems worse.[10]
Many disadvantages of using some script languages are caused by design flaws within the language syntax or implementation, and are not necessarily imposed by the use of a text-based command-line

Conclusion: We have successfully implemented program for 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.

Questions:

1. Why is a shell script needed?
2. Write some advantages of shell scripting.
3. How we can create the .text file
4. How we can insert the record.
5. Tell me by which command we can find the pattern.

OPERATING SYSTEM LAB			
Experiment No: 2 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 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.**	**Page**	**1/4**

Aim: UNIX Process Control- I
Apparatus: Ubuntu Operating system, gcc

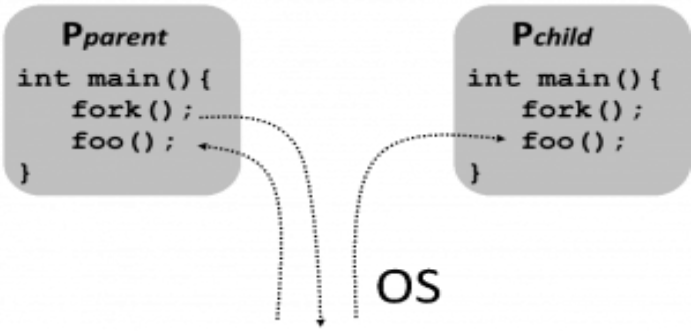
Theory:

fork() in C

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

It take no parameters and return integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.



Please note that the above programs don't compile in Windows environment.

Predict the Output of the following programs:

1. **Predict the Output of the following program**.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
```

```
 // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

```
   Hello world!
   Hello world!
```

```
int main()
      {
            printf("Before Forking");
            fork();
            printf("After Forking");
            return 0;
      }
```

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.

If we run this program, we might see the following on the screen:
Before Forking
After Forking
After Forking
Here printf() statement after fork() system call executed by parent as well as child process.
Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space.
Other address spaces created by fork() calls will not be affected even though they have identical variable names.
Consider one simpler example, which distinguishes the parent from the child.
In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i.

```
#include <stdio.h>
#include <sys/types.h>
void ChildProcess(); /* child process prototype */
```

```
void ParentProcess(); /* parent process prototype */
int main()
{
        pid_t pid;
        pid = fork();
        if (pid == 0)
        ChildProcess();
        else
        ParentProcess();
        return 0;
}

void ChildProcess()
{
}
void ParentProcess()
{
}
```

Zombie Process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes

detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX *nohup* command is one means to accomplish this.

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes

Conclusion: We have successfully implemented program for integers to be sorted using Unix Process Control -I

Questions: 1. What is fork system call.

2. What are the return types of fork system call.

3. Which header file we need to include while using fork system call.
4. How the address allocation done for child process
5. How we can distinguish the parent and child process.
6. What is process id.

SOFTWARE LABORATORY – II			
Experiment No: 2 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.**	**Page**	**1/2**

Aim: UNIX Process Control- I

Apparatus: Ubuntu Operating system, gcc

Theory: 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.

exec family:

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/ls", "ls", "-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. The function execlp() 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( ):

int execve(const char *filename, char *const argv[ ], char *const envp[ ]);

It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form:

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.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes. The execution of wait() could have two possible situations.

1 If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2 If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Conclusion: We have successfully implemented program for Binary Search using Unix Process Control -II

Questions.

1 Explain the execve system call

2. How we can pass the argument through command prompt.

3. Explain the argc and argv

4. What are the basic defence/s agc and argv

5. what is binary file

OPERATING SYSTEM LAB			
Experiment No: 3	**Implement the C program for CPU Scheduling Algorithms: Shortest Job First (Preemptive) and Round Robin with different arrival time.**	**Page**	**1/3**

Aim: Study of CPU scheduling

Apparatus: Ubuntu Operating system, gcc

Theory:

Why do we need scheduling?
A typical process involves both I/O time and CPU time. In a uni programming system like MS-OS, time spent waiting for I/O is wasted and CPU is free during this time. In multi programming systems, one process can use CPU while another is waiting for I/O. This is possible only with process scheduling.

Objectives of Process Scheduling Algorithm

Max CPU utilization [Keep CPU as busy as possible]
Fair allocation of CPU.
Max throughput [Number of processes that complete their execution per time unit]
Min turnaround time [Time taken by a process to finish execution]
Min waiting time [Time a process waits in ready queue]
Min response time [Time when a process produces first response]

Arrival Time: Time at which the process arrives in the ready queue.
Completion Time: Time at which process completes its execution.
Burst Time: Time required by a process for CPU execution.
Turn Around Time: Time Difference between completion time and arrival time.
Turn Around Time = Completion Time – Arrival Time

Different Scheduling Algorithms

Shortest Job First (Preemptive) and Round Robin, FCFS, Priority.

Shortest job first (SJF) :-

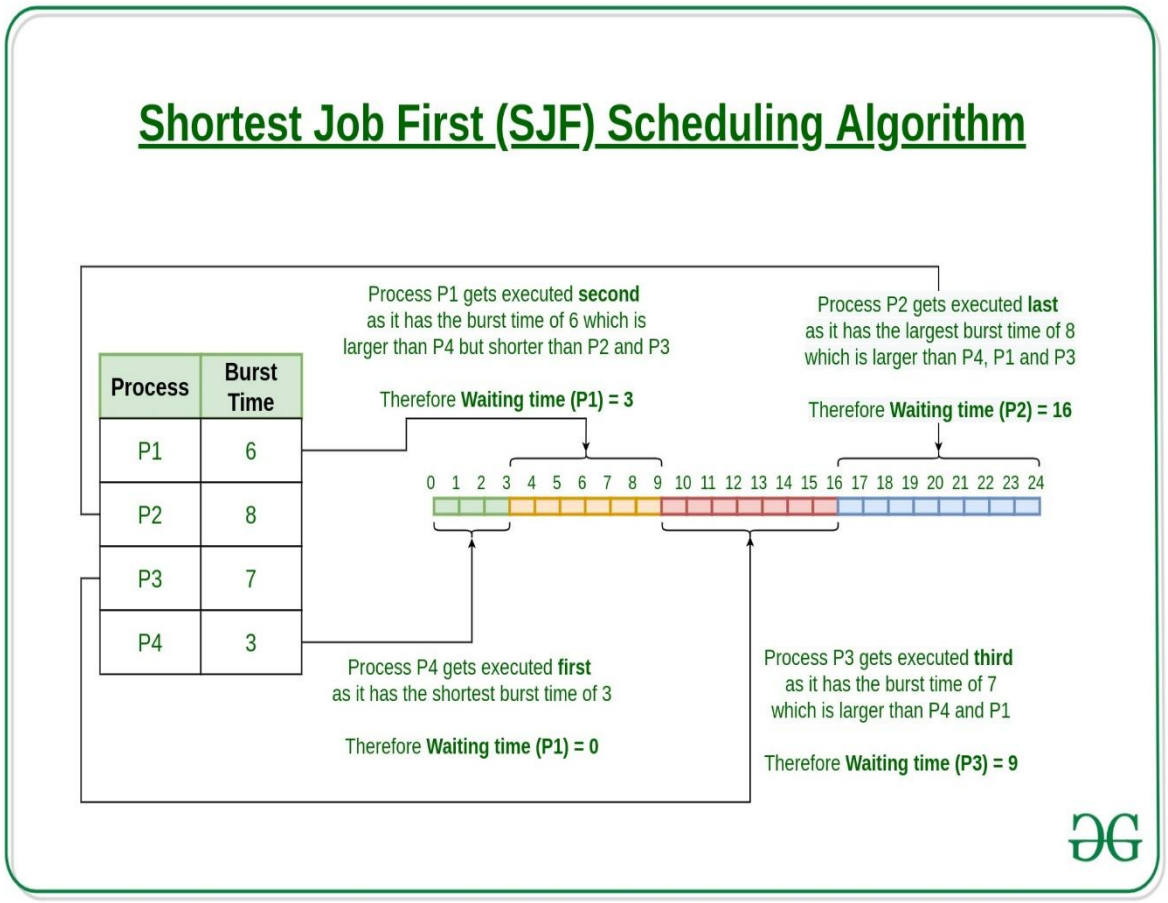
Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

- Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
- It is a Greedy Algorithm.

- It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
- It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

Algorithm:

- Sort all the process according to the arrival time.
- Then select that process which has minimum arrival time and minimum Burst time.
- After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.



Round Robin-

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

Algorithm

```
1- Create an array rem_bt[] to keep track of remaining
   burst time of processes. This array is initially a
   copy of bt[] (burst times array)
2- Create another array wt[] to store waiting times
   of processes. Initialize this array as 0.
3- Initialize time : t = 0
4- Keep traversing the all processes while all processes
   are not done. Do following for i'th process if it is
   not done yet.
    a- If rem_bt[i] > quantum
       (i)  t = t + quantum
       (ii) rem_bt[i] -= quantum;
    c- Else // Last cycle for this process
       (i)  t = t + rem_bt[i];
       (ii) wt[i] = t - bt[i]
       (ii) rem_bt[i] = 0; // This process is over
```

Conclusion: Thus we have successfully implement the CPU scheduling alorithms.

Questions: 1. What is CPU scheduling.

2. Why Scheduling is important.

3.define the term burst time

4.What is pre-emptive and non-pre-emptive scheduling

5. how we can calculate turnaround time.

OPERATING SYSTEM LAB			
Experiment No: 4A	**Thread synchronization using counting semaphores. Application to demonstrate: producer- consumer problem with counting semaphores and mutex.**	**Page**	**1/3**

Aim: Producer-Consumer Problem (Using semaphores)

Apparatus: Ubuntu Operating system, gcc

Theory:

The Producer/Consumer Problem

The general statement is this: 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 anyone time. The problem is to make sure that the producer won't try to add data in to 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:

Solution to bounded/circular buffer producer-consumer problem

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

Solution using Semaphore (Unbounded Buffer).

```
semaphore s = 1, n = 0;
void producer()
{
while (true)
{
produce();
semWait(s);
append();
semSignal(s);
semSignal(n);
}
}


void consumer()
```

```
{
while (true)
{
semWait(n);
semWait(s);
take();
semSignal(s);
consume();
}
}
void main()
{
parbegin (producer, consumer);
}
```

Semaphore

It is a special type of variable containing 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, also known as a counting semaphore or a general semaphore. A semaphore may be initialized to a nonnegative integer value. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWaitoperation,if any, is unblocked.

Generalized use of semaphore for forcing critical section

```
semaphoresv = 1;
loop forever
{
Wait(sv);
critical code section;
signal(sv);
noncritical code section;
}
```

Linux Semaphore Facilities (Binary Semaphore)

A semaphore is created with the sem_init function, which is declared as follows:

#include <semaphore.h>

intsem_init(sem_t *sem, intpshared, unsigned int value);

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared*parameter controls the type of semaphore. If the value of *pshared*is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for *pshared*will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

#include <semaphore.h>
intsem_wait(sem_t * sem);
intsem_post(sem_t * sem);

These both take a pointer to the semaphore object initialized by a call to sem_init. The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2. The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic "test and set" ability in a single function is what makes semaphores so valuable.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

#include <semaphore.h>
intsem_destroy(sem_t * sem);

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

Conclusion: - We have successfully implemented program of Thread synchronization using counting semaphores and mutual exclusion using mutex.

Questions:

1. Define the process synchronization
2. Explain the semaphore
3. Enlist the types of semaphore
4. What is buffer bounded problem
5. Explain the algorithm of producer-consumer problem.

OPERATING SYSTEM LAB			
Experiment No: 4B	**Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader- Writer problem with reader priority.**	**Page**	**1/3**

Aim: Thread synchronization

Apparatus: Ubuntu Operating system, gcc

Theory:

Readers-Writers Problem | Set 1 (Introduction and Readers Preference Solution)

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 mutex '**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.

Conclusion:- We have successfully implemented program of Reader-Writer problem with reader priority

Questions

1.What are different conflicts in reader writer problem how we can solve it.

2.Basic difference between mutex and semaphore

3. define wait and signal function

4.How reader process allows the reader processes

5. if writer process in critical region then how we can avoid the other writer process to go in to critical region.

OPERATING SYSTEM LAB			
Experiment No: 5	**Implement the C program for Deadlock Avoidance Algorithm: Bankers Algorithm.**	**Page**	**1/3**

Aim: Study of banker algorithm

Apparatus: Ubuntu Operating system, gcc

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.

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.
- 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 $\mathbf{R_j}$
- 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
$Allocation_i$ = $Allocation_i$ + $Request_i$

$Need_i = Need_i – Request_i$

Conclusion: Hence We have studied the banker algorithm.

Questions:

1.How we cn calculate the need matrix.

2. What is deadlock

3.How we can avoid the dealock.

4. How many matrix used in bankers algorithms.

5. Explain the bankers algorithm.

OPERATING SYSTEM LAB			
Experiment No: 6	**Implement the C program for Page Replacement Algorithms: FCFS, LRU, and Optimal for frame size as minimum three.**	**Page**	**1/3**

Aim: Study of page replacement

Apparatus: Ubuntu Operating system, gcc

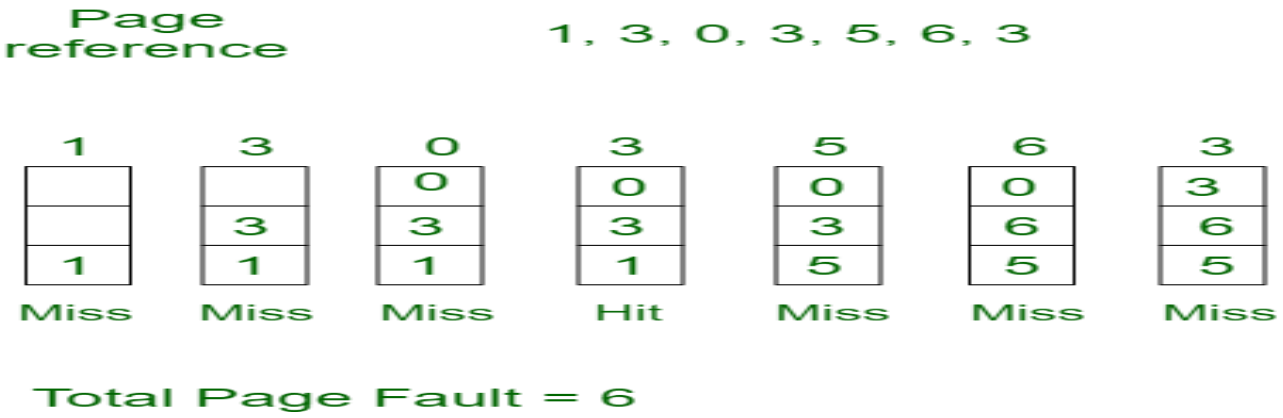
Theory:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

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.

1. First In First Out (FIFO) –

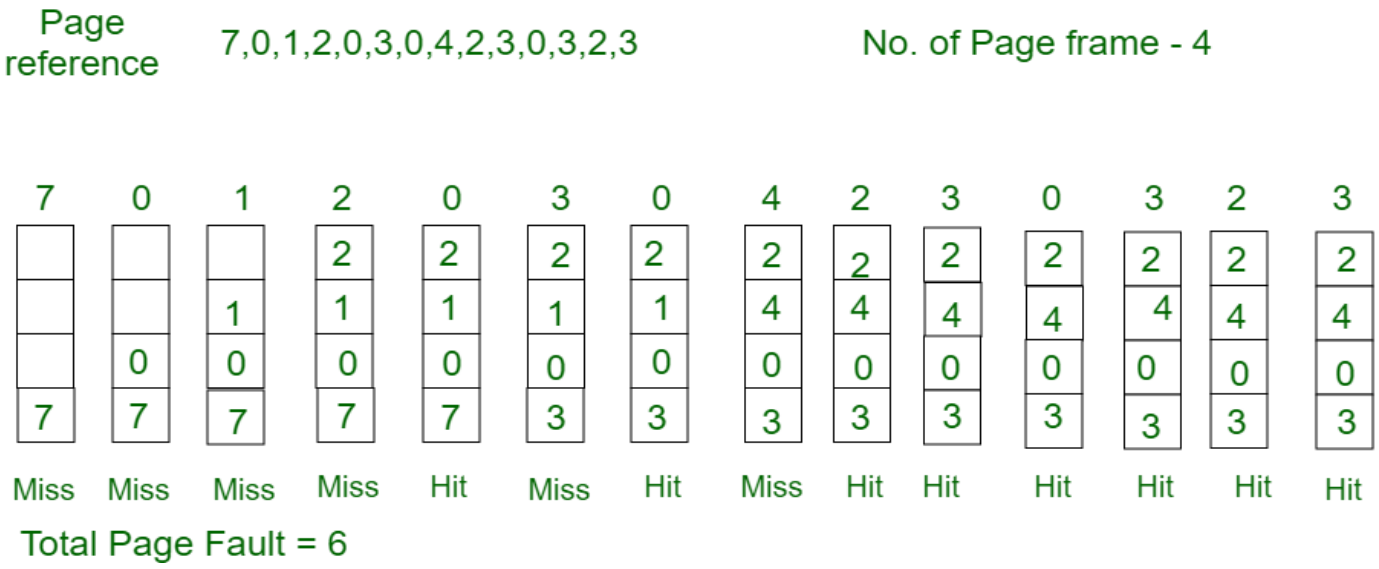
This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal. **Example-1**Consider page 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.

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.



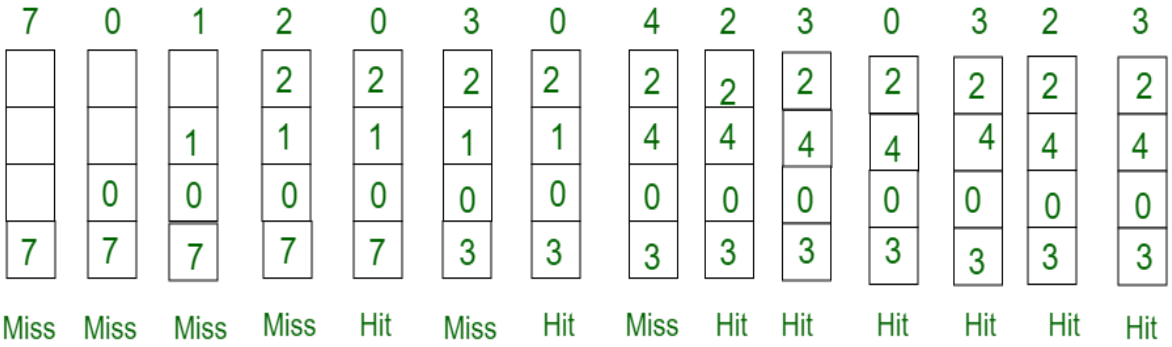
nitially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**
0 is already there so —> **0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



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.

Conclusion: Thus, We have studied and implement the page replacement algorithms successfully.

Questions:

1.Define the MMU

2. what is page fault

3. when page hit will occur

4. tell the importance of page replacement

5. which page algorithm gives better result in term of pafe fault.

OPERATING SYSTEM LAB			
Experiment No: 7 A	**Pipes: Full duplex communication between parent and child processes. Parent process writes a pathname of a file (the contents of the file are desired) on one pipe to be read by child process and child process writes the contents of the file on second pipe to be read by parent process and displays on standard output.**	**Page**	**1/3**

Aim: Inter process communication in Linux using following

Apparatus: Ubuntu Operating system, gcc

Theory:

Pipes behave **FIFO**(First in First out), Pipe behave like a **queue** data structure. Size of read and write don't have to match here. We can write **512** bytes at a time but we can read only 1 byte at a time in a pipe.

Syntax in C language:

int pipe(int fds[2]);
Parameters :
fd[0] will be the fd(file descriptor) for the read end of pipe.
fd[1] will be the fd for the write end of pipe.
Returns : 0 on Success.
-1 on error.

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem. Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem. The kernel maintains exactly one pipe object for each FIFO special file that is opened by at least one process. The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

FIFOs are pipes with a name and are also commonly referred to as named pipes. Pipes are common on Linux command lines but do not have a system-wide name. So, any two processes that wish to communicate using a pipe need to be related, either parent and child or, sharing a common parent, who sets up the pipe and passes its file descriptors to individual processes. A FIFO is different because it has a name like a file. Also, since it exists in the filesystem, it is not necessary for the processes to be related to communicate using the FIFO.

We can create a FIFO from the shell using the mknod command. We can remove a FIFO just like a file using the rm command.

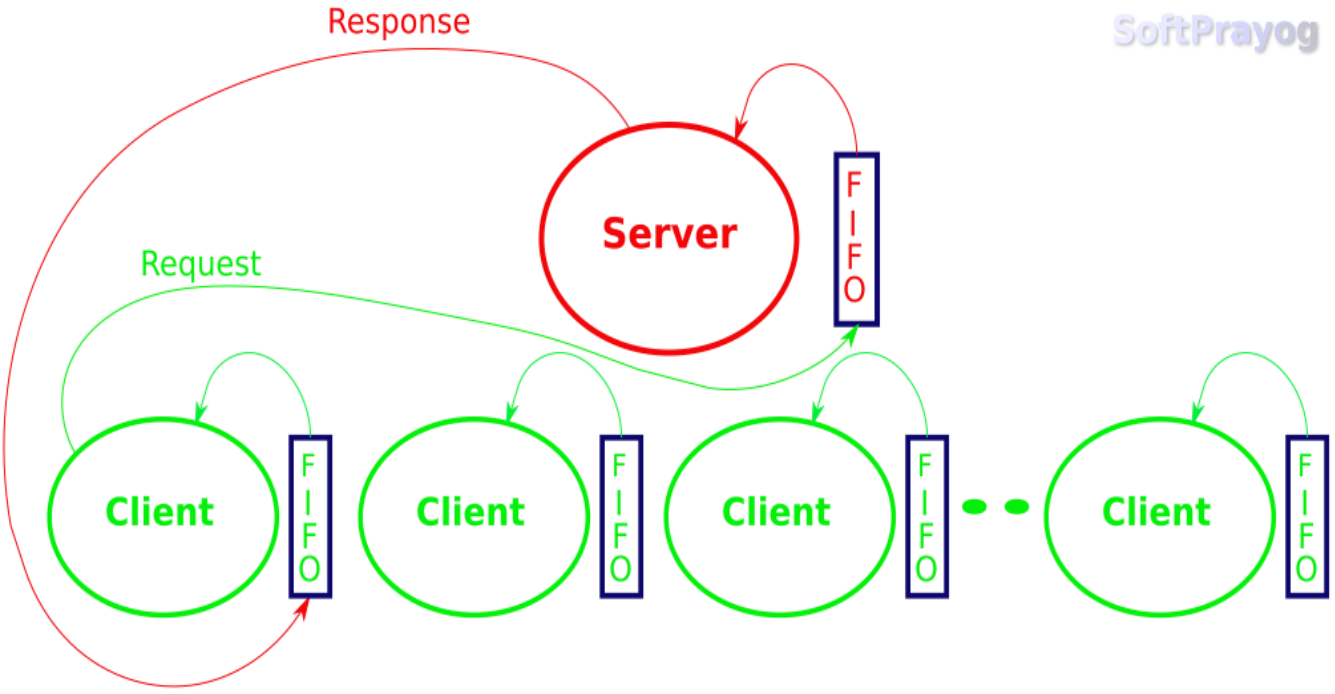
```
$ mknod myfifo p
```

```
$ ls -ls
total 0
0 prw-rw-r-- 1 user1 user1 0 May 30 15:08 myfifo
$ rm myfifo
$ ls -ls
total 0
```

There is a specialized mkfifo command which could have been used in place of the mknod command, above.

```
$ mkfifo mynewfifo
$ file mynewfifo
mynewfifo: fifo (named pipe)
```

The client-server paradigm comprises of a single server process, which works all the time, receives requests from clients and gives them responses. A client is the process that manages the inputs and outputs for a live user. Clients come and go but the server works all the time. The clients communicate with the server using an interprocess communication mechanism. Each process in the paradigm has a system-wide mechanism for receiving messages. In the example in a later section, we will use the FIFO as the mechanism for receiving messages. That is, the server will have a FIFO, where clients can put messages for the server. Similarly, each client will have a FIFO, where the server can put in messages for that client.



Interprocess Communication between client and server using FIFOs

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

int mkfifo (const char *path, mode_t perms);
// Returns -1 on error.
```

Conclusion: We have successfully implemented program of inter process communication(IPC) in Linux using FIFO.

Questions:

1.What do you mean by FIFO

2. Explain the interposes communication

3. Tell the syntax how FIFO create

4. Brief FIFO

5. Difference between FIFO and PIPE

OPERATING SYSTEM LAB			
Experiment No: 7B	**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.**	**Page**	**1/5**

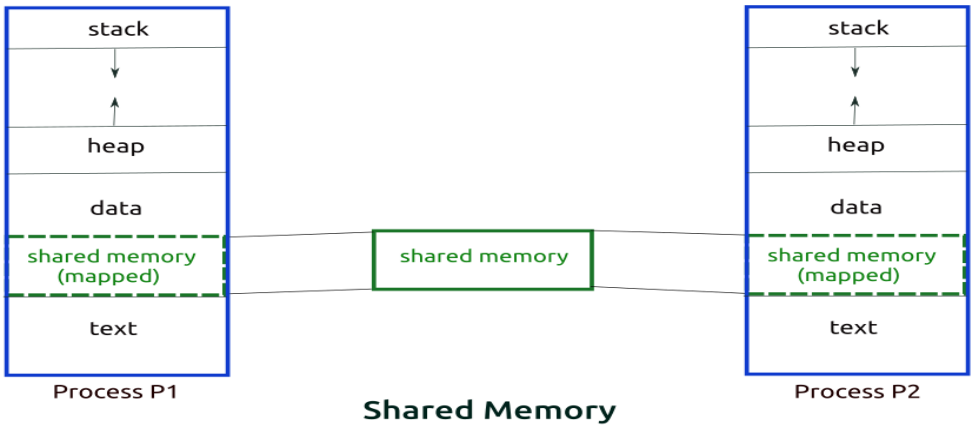
Aim: Study Inter-process Communication

Apparatus: Ubuntu Operating system, gcc

Theory:

1.Shared Memory

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.

2.System V and POSIX Shared Memory

Like message queues and semaphores, shared memory also comes in two flavors, the traditional System V shared memory and the newer POSIX shared memory. In this post, we will look at the

System V shared memory calls. Example programs for a server and client processes communicating via the System V shared memory are given near the end of the post.

3. System V IPC key

To use a System V IPC mechanism, we need a System V IPC key. The ftok function, which does the job, is

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (const char *pathname, int proj_id);
```

The *pathname* is an existing file in the filesystem. The last eight bits of *proj_id* are used; these must not be zero. A System V IPC key is returned.

4.System V Shared Memory Calls

4.1 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.

4.2 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.

On success, shmat returns pointer to the attached shared memory segment. On error, (void *) -1 is returned, with errno set to the cause of the error.

4.3 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.

4.4 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,

```
struct shmid_ds {
struct ipc_perm shm_perm; /* Ownership and permissions */
size_t  shm_segsz; /* Size of segment (bytes) */
time_t shm_atime; /* Last attach time */
time_t shm_dtime; /* Last detach time */
time_t shm_ctime; /* Last change time */
pid_t shm_cpid; /* PID of creator */
pid_t  shm_lpid; /* PID of last shmat(2)/shmdt(2) */
shmatt_t shm_nattch; /* No. of current attaches */
...
};
```

Where. the struct ipc_perm is,

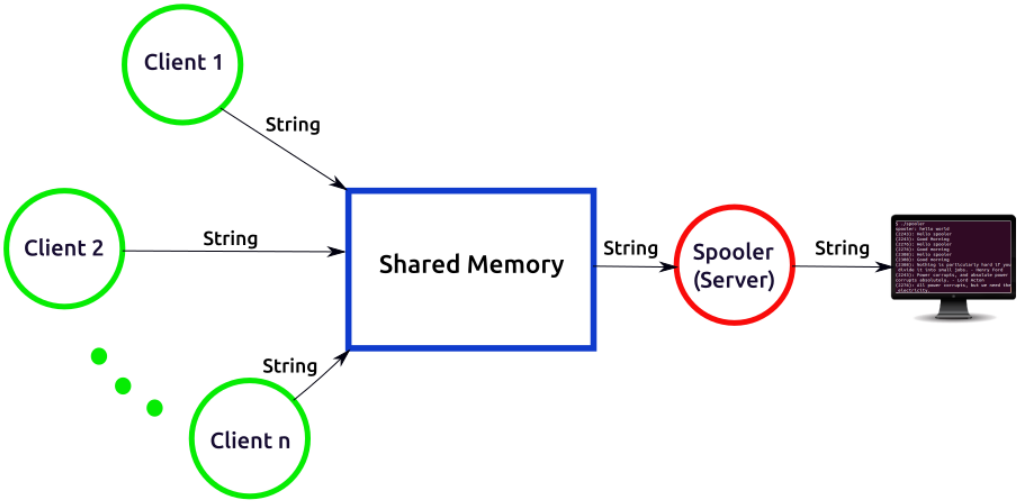
```
struct ipc_perm {
key_t __key; /* Key supplied to shmget(2) */
uid_t uid; /* Effective UID of owner */
gid_t gid; /* Effective GID of owner */
uid_t cuid; /* Effective UID of creator */
gid_t cgid; /* Effective GID of creator */
unsigned short mode; /* Permissions + SHM_DEST and SHM_LOCKED flags
*/
```

```
        unsigned short __seq; /* Sequence number */
};
```

The *cmd* parameter in shmctl specifies the command. The important commands are IPC_STAT, IPC_SET and IPC_RMID. The IPC_STAT command copies the data in the kernel data structure shmid_ds for the shared memory into the location pointed by the parameter *buf*. With the IPC_SET command, we can set some of the fields in the shmid_ds structure in the kernel for the shared memory segment. The fields that can be modified are shm_perm.uid, shm_perm.gid and the least significant 9 bits of shm_perm.mode. The command, IPC_RMID, marks a shared memory segment for removal from the system. The shared memory segment is actually removed after the last process detaches it from its address space.

5.0 An example: Client-Server communication using System V Shared Memory

The example has a server process called spooler which prints strings received from clients. The spooler is a kind of consumer process which consumes strings. The spooler creates a shared memory segment and attaches it to its address space. The client processes attach the shared memory segment and write strings in it. The client processes are producers; they produce strings. The spooler takes strings from the shared memory segment and writes the strings on the terminal. The spooler prints strings in the order they are written in the shared memory segment by the clients. So, effectively, there are *n* concurrent processes producing strings at random times and the spooler prints them nicely in the chronological order. The software architecture is like this.



Software Architecture for Clients and Server using Shared Memory

Conclusion: We have successfully implemented program of Interprocess communication using System V Shared Memory in Linux.

Questions:

1.Explain the V system call

2. Explain the Shget function

3. Explaain the Shmat

4. Tell the syntax of shmdt

5. Explain IPC

OPERATING SYSTEM LAB			
Experiment No: 8	**Implement the C program for Disk Scheduling Algorithms: SSTF, SCAN, C-Look considering the initial head position moving away from the spindle.**	**Page**	**1/4**

Aim: Study Disk scheduling algorithms

Apparatus: Ubuntu Operating system, gcc

Theory:

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

TYPES OF DISK SCHEDULING ALGORITHMS

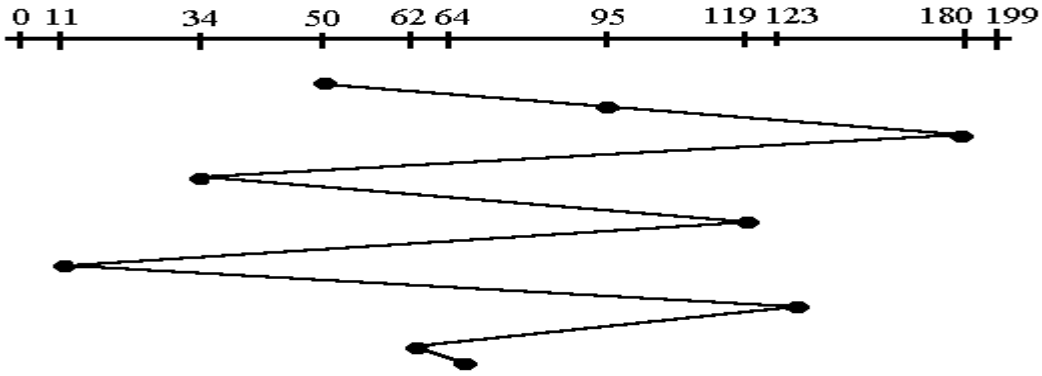
Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:
First Come-First Serve (FCFS)
Shortest Seek Time First (SSTF)
Elevator (SCAN)
Circular SCAN (C-SCAN)
LOOK
C-LOOK
These algorithms are not hard to understand, but they can confuse someone because they are so similar. What we are striving for by using these algorithms is keeping Head Movements (# tracks) to the least amount as possible. The less the head has to move the faster the seek time will be. I will show you and explain to you why C-LOOK is the best algorithm to use in trying to establish less seek time.

Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

1.First Come -First Serve (FCFS)

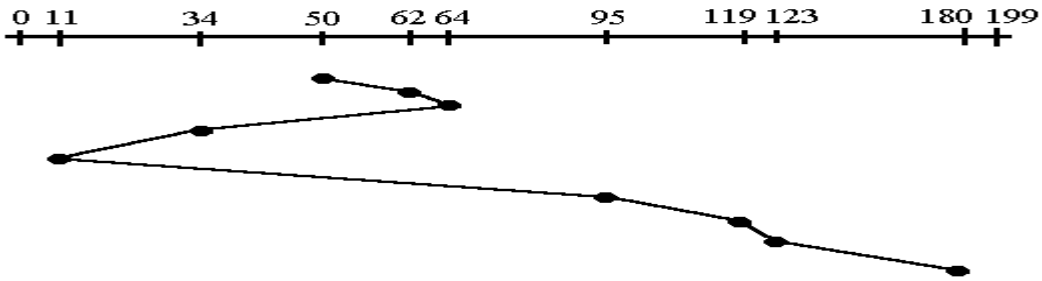
All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from

one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



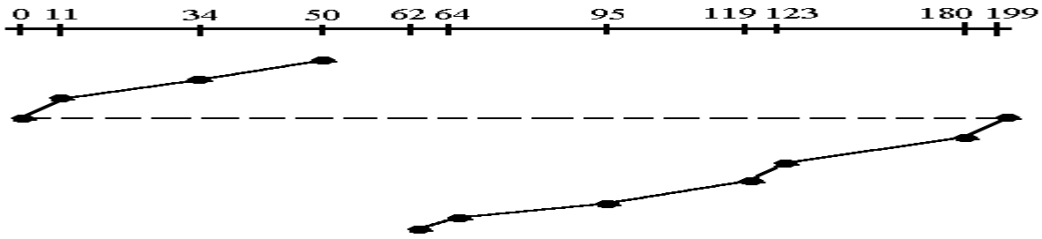
2.Shortest Seek Time First (SSTF)

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



3.Elevator (SCAN)

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

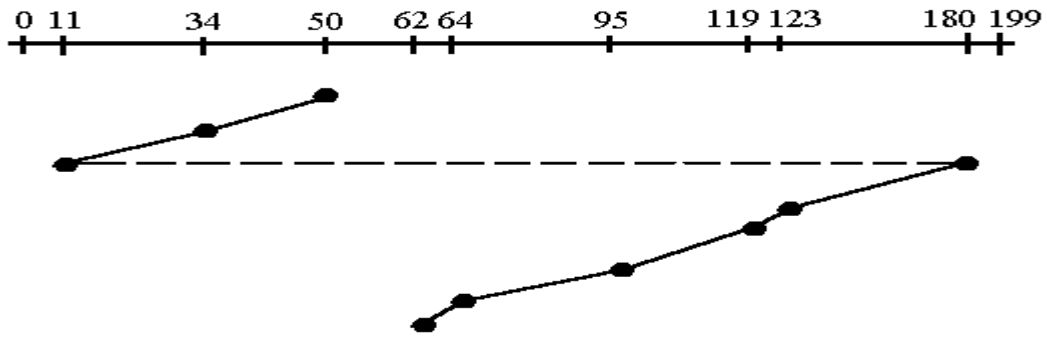


4.C-LOOK

his is just an enhanced version of C-SCAN. In this the scanning doesn't go past the last request in the direction that it is moving. It too jumps to the other end but not all the way to the end. Just to the furthest request. C-SCAN had a total movement of 187 but this scan (C-LOOK) reduced it down to 157 tracks.

From this you were able to see a scan change from 644 total head movements to just 157. You

should now have an understanding as to why your operating system truly relies on the type of algorithm it needs when it is dealing with multiple processes.



Conclusion: Thus, we have studied the disk scheduling algorithms

Experiment No: 09	**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 kernel source, any architecture and any Linux kernel distribution) and demonstrate the use of this embedded system call using C program in user space.**	**Page**	**1/3**

Aim: add new system call

Apparatus: Ubuntu Operating system, gcc

Theory:

For adding new system call to kernel, we need to implement our system call code, modify some kernel source files, and then compile and install the kernel. Once we have rebooted into the new kernel, we'll make use of a helper program that will invoke the newly added system call.
System calls in Linux are stored in syscall.tbl in arch/syscalls

- 32-bit system call table – syscall_32.tbl
- 64-bit system call table – syscall_64.tbl

Two ways to write a system call to kernel

- Adding a kernel module
- Change the existing kernel code

Setting-up our system call directory

mkdir /usr/src/linux-3.17.7/hello && cd hello

- cd /usr/src/linux-3.17.7
- Write the code for the system call in the source file(e.g. hello.c)

sudo vi hello
Inside the vi editor, we'll add the following code:

```
/* hello.c systemcall code */
#include <linux/kernel.h>
/* asmlinkage indicates we use the kernel stack to pass parameters */
asmlinkage long sys_addwm(int i, int j)
{
printk(KERN_INFO "Addwm is working! Now adding %d and %d", i, j);
return i+j;
}
```

- Create the makefile for our system call in the same directory (e.g. Makefile).

  sudo vi Makefile
  and add the following content to it:

obj-y := hello.o

- Make changes to kernel Makefile so as to accommodate our directory of a system call. (e.g. search for "kernel/ mm/" in /usr/src/linux-3.17.7/Makefile)

sudo vi /usr/src/linux-3.17.1/Makefile
We can then go to the line number & change the following line of code:
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
to
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/
Adding our system call to the 32-bit or 64-bit system call table:

- Make changes to /usr/src/linux-3.17.7/arch/x86/syscalls/syscall_64.tbl by adding system call No., function name & start routine.

321 common addnum sys_addnum

- Make changes to /usr/src/linux-3.17.7/include/linux/syscalls.h to add a prototype of our system call. Add following line in that file.

asmlinkage long sys_addnum(int i, int j);

- Compile, link and install the kernel
  - sudo make menuconfig
  - sudo make –j2 (2 cores)
  - sudo make modules_install
  - sudo make install
  - sudo reboot
- Now implement the helper program that will call our system call.
  - Write a helper program (e.g. helper.c in any directory)
  - Use syscall function known as indirect system call to invoke our system call.
  - syscall is needed because we have no wrapper function to call our system call through that wrapper function.

We can now use gedit to implement our helper program, and add the following code to the file:

```
#include <stdio.h>
#include <ctype.h>
#include <syscall.h>
int main(int argc, char *argv[]){
int var1, var2 = 0;
long res = 0;
```

```
char term;
printf("Please enter 2 space separated numbers and hit the enter key..\n:");
/*1*/
if(scanf("%d%d%c", &var1, &var2, &term) != 3 || term != '\n'){
printf("Invalid input, please try again.\n");
}
else {
/* remember, our syscall number is 321,
so we invoke it using its number.  */
res = syscall(321, var1, var2);
printf("%s: You entered %d and %d. Result is: %ld\n", argv[0], var1, var2, res);
}
return 0;
}
```

Save above file with hello.c & compile it as follow.

gcc hello.c

execute the helper program as follows:

./a.out

Please enter 2 space separated numbers and hit the enter key.

:4 8

./addwm-main: You entered 4 and 8. Result is: 12

Conclusion: A new system call is added to Linux kernel and tested successfully.