Lab Manual
Operating Systems Lab

TE – Information Technology

Pune Institute of Computer Technology, Pune - 411043

ASSIGNMENT No: 1-A

Title : Basic Linux Commands

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

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 A-Z >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 case-insensitive, 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. Programmers often use this command when they need to make program alterations instead of rewriting the entire source code.

The simplest form of this command is **diff file1.ext file2.ext**

20. tar command

The **tar** command is the most used command to archive multiple files into a **tarball** — a common Linux file format that is similar to zip format, with compression being optional.

This command is quite complex with a long list of functions such as adding new files into an existing archive, listing the content of an archive, extracting the content from an archive, and many more. Check out some **practical examples** to know more about other functions.

21. chmod command

chmod is another Linux command, used to change the read, write, and execute permissions of files and directories. As this command is rather complicated, you can read **the full tutorial** in order to execute it properly.

22. chown command

In Linux, all files are owned by a specific user. The **chown** command enables you to change or transfer the ownership of a file to the specified username. For instance, **chown linuxuser2 file.ext** will make **linuxuser2** as the owner of the **file.ext**.

23. kill command

If you have an unresponsive program, you can terminate it manually by using the **kill** command. It will send a certain signal to the misbehaving app and instructs the app to terminate itself.

There is a total of **sixty-four signals** that you can use, but people usually only use two signals:

> **SIGTERM (15)** — requests a program to stop running and gives it some time to save all of its progress. If you don't specify the signal when entering the kill command, this signal will be used.
>
> **SIGKILL (9)** — forces programs to stop immediately. Unsaved progress will be lost.

Besides knowing the signals, you also need to know the process identification number (PID) of the program you want to **kill**. If you don't know the PID, simply run the command **ps ux**.

After knowing what signal you want to use and the PID of the program, enter the following syntax:

kill [signal option] PID.

24. 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.

ASSIGNMENT No: 1-B

Title : Shell Programming

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.

OBJECTIVE:

This assignment helps the students understand the basic commands in Unix/Linux and how write the shell scripts.

THEORY:

What Is Shell Scripting?
Being a Linux user means you play around with the command-line. Like it or not, there are just some things that are done much more easily via this interface than by pointing and clicking. The more you use and learn the command-line, the more you see its potential. Well, the command-line itself is a program: the shell. Most Linux distros today use Bash, and this is what you’re really entering commands into.
Now, some of you who used Windows before using Linux may remember batch files. These were little text files that you could fill with commands to execute and Windows would run them in turn. It was a clever and neat way to get some things done, like run games in your high school computer lab when you couldn’t open system folders or create shortcuts. Batch files in Windows, while useful, are a cheap imitation of shell scripts.
Shell scripts allow us to program commands in chains and have the system execute them as a scripted event, just like batch files. They also allow for far more useful functions, such as command substitution. You can invoke a command, like date, and use its output as part of a file-naming scheme. You can automate backups and each copied file can have the current date appended to the end of its name. Scripts aren’t just invocations of commands, either. They’re programs in their own right. Scripting allows you to use programming functions – such as ‘for’ loops, if/then/else statements, and so forth – directly within your operating system’s interface. And, you don’t have to learn another language because you’re using what you already know: the command-line.

Shell scripts are executed in a separate child shell process. This is done by providing special interpreter line at the beginning (starting with #!).

To run the script we make it executable and then invoke the script name.

```
$ chmod +x script.sh      or      $ chmod 755 script.sh
$ script.sh
```

User can explicitly spawn a child shell of his choice with the script name as argument. In this case it is not mandatory to include the interpreter line.

read: Making Scripts Interactive-

The read statement is the shell's internal tool for taking inputs from the user, i.e., making scripts interactive. It is used with one or more variable. When we use a statement like

read name

The script pauses at that point to take i/p from the keyboard. Since this is the form of assignment, no $ is used before name.

Using Command-Line Arguments

The shell script accepts arguments from the command line. The first argument is read by shell into parameter $1, second into $2, and so on.

Special Parameter used by Shell:

$*	-	It stores complete set of positional parameters as a single string.
$#	-	It is set to number of arguments specified. Used to check whether right number of argument have been entered.
$0	-	Holds the command name itself.
"$@"	-	Each Quoted string treated as a separate argument.
$?	-	Exit Status of Last Command.

exit and EXIT STATUS OF COMMAND

exit - Command to terminate a program. This command is generally run with numeric arguments.

exit0	-	When everything went fine
exit1	-	When something went wrong
exit2	-	Failure in opening a file.

Example:

```
$ grep "director" emp.lst >/home/vishal; echo$?
```

"All command returns an exit Status"

Logical Operators && and || - Conditional Execution

Cmd1 && Cmd2: The Cmd2 will execute only when Cmd1 is succeeds.
Cmd1 || Cmd2: The Cmd2 will execute only when Cmd1 is fails.

Example:

```
$ grep "director" emp.lst >/home/vishal && echo "Pattern found in file"

$ grep "manager" emp.lst >/home/vishal || echo "Pattern not found in file"
```

THE if CONDITIONAL

Form 1	Form 2	Form 3

if command is successful
then
    execute command
else
    execute command
fi
	if command is successful
	then
	    execute command
	fi
		if command is successful
		then
		    execute command
		elif command is successful
		then ….
		else …..
		fi

Using test and [] to evaluate expressions

When we use if to evaluate expressions, we require the test statement because the true or false values returned by expressions cant be directly handled by if.

test use certain operator to evaluate the condition on its right and returns either true or false exit status, which is then used by if for making decisions.

test works in 3 ways:

- Compare two numbers
- Compare two strings or a single string for a null value
- Check a file attribute

Numeric Comparison

The numeric comparison operators always begin with a – (hyphen), followed by two character word, and enclosed either side by a whitespace.

Example:

```
$ x=5; y=7; z=7.2
$test $x –eq $y; echo $?
1
$test $z –eq $y; echo $?
0
```

Numeric comparison is restricted to integers only.
Operators: -eq, -ne, -gt, -ge, -lt, -le.

String Comparison

Another set of operator is used for string comparison.
String tests used by test

Test	True if
S1 = S2	String S1 is equal to String S2
S1 != S2	String S1 is not equal to String S2
-n stg	String stg is not a null String
-z stg	String stg is a null String
stg	String stg is assigned and not a null String

test also permits the checking of more than one condition in the same line using the –a (AND) and –o(OR) operators.

Example:

```
if  [ -n “$pname” –a -n “$flname” ] ; then
demo.sh “$pname” “$flname”
else
echo “ At least one input was null string ” ; exit 1
fi
```

File Tests

test can be used to test various file attributes like its type or permissions.

Example:

```
#!/bin/sh
if [ ! -e $1 ] ; then
     echo “ File does not exists ”
elif [ ! -r $1 ] ; then
      echo “ File is not readable ”
elif [ ! -w $1 ] ; then
      echo “ File is not writable ”
else
      echo “ File is both readable and writable ”
fi
```

File – Related Tests with test

Test	true if
-f *file*	*file* exists and is a regular file
-r *file*	*file* exists and is a readable
-w *file*	*file* exists and is a writable
-x *file*	*file* exists and is a executable
-d *file*	*file* exists and is a directory
-s *file*	*file* exists and has a size greater than zero

THE case CONDITIONAL

case statement matches an expression or string for more than one alternative, in more efficient manner than if.

Form:

```
case expr in
        pattern 1) cmd1 ;;
        pattern 2) cmd2 ;;
        pattern 3) cmd3 ;;
esac
```

expr : Computation and String Handling

The shell relies on external **expr** command for computing features.
Functions of expr:

- Perform arithmetic operations on integers
- Manipulate strings

Computation
expr can perform four basic arithmetic operation as well as modulus function

String Handling

expr can perform 3 important string functions

- Determine the length of the string.
- Extract a sub-string.
- Locate the position of a character in a string

Assignments

1. Use a script to take two numbers as arguments and output their sum using
   i) bc ii) expr.
   Include error checking to test if two arguments were entered.
2. Write a shell script that uses find to look for a file and echo a suitable msg if the file is not found. You must not store the find output in a file.

ASSIGNMENT No: 2

Title : Process Control System Calls

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

1. 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.
2. Implement the C program in which main program accepts an integer array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an integer 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.

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

THEORY:

Process in UNIX:

A process is the basic active entity in most operating-system models.

Process IDs

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

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

Creating Processes

Two common techniques are used for creating a new process.

1. using system() function.

2. using fork() system calls.

1. **Using system**

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

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

2. **Using fork**

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

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

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

The wait Function

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

SYNOPSIS

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

pid_t wait(int *status);
```

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

Example:

```
pid_t childpid;

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

Status values

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

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

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

SYNOPSIS

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

WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)
```

New program execution within the existing process (The exec Function)

The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent. The exec family of functions provides a facility for overlaying the process image of the calling process with a new image. The traditional way to use the fork–exec combination is for the child to execute (with an exec function) the new program while the parent continues to execute the original code.

SYNOPSIS

```
#include <unistd.h>

extern char **environ;
```

1. int execl(const char *path, const char *arg0, ... /*, char *(0) */);
2. int execle (const char *path, const char *arg0, ... /*, char *(0),

char *const envp[] */);

3. int execlp (const char *file, const char *arg0, ... /*, char *(0) */);
4. int execv(const char *path, char *const argv[]);
5. int execve (const char *path, char *const argv[], char *const envp[]);
6. int execvp (const char *file, char *const argv[]);

exec() system call:

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.

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

Process Termination

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

Zombie Processes

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

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

When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

Orphan Process:

An Orphan Process is nearly the same thing which we see in real world. Orphan means someone whose parents are dead. The same way this is a process, whose parents are dead, that means parents are either terminated, killed or exited but the child process is still alive.

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

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

Finding a Orphan Process:

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

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

This will show you all the orphan processes running in your system. The output from this command confirms that they are Orphan processes but does not mean that they are all useless, so confirm from some other source also before killing them.

Killing a Orphan Process:

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

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

If that does not work then simply use

```
# kill -9 <PID>
```

Daemon Process:

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.

vfork: alternative of fork

create a new process when exec a new program.

Compare with fork:

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

INPUT:

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

OUTPUT:

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

FAQs:

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

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

Example 1

Printing the Process ID

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

int main()
{
printf(“The process ID is %d\n”, (int) getpid());
printf(“The parent process ID is %d\n”, (int) getppid());
return 0;
}
```

Example 2

Using the system call

```
#include <stdlib.h>

int main()
{
int return_value;
return_value=system(“ls –l /”);
return return_value;
}
```

Example 3

Using fork to duplicate a program’s process

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

int main()
{
pid_t child_pid;
printf(“The main program process ID is %d\n”, (int) getpid());
child_pid=fork();
if(child_pid!=0)  {
printf(“This is the parent process ID, with id %d\n”, (int) getpid());
printf(“The child process ID is %d\n”, (int) chi;d_pid);
}
else
printf("This is the child process ID, with id %d\n", (int) getpid());
return 0;
}
```

Example 4

Determining the exit status of a child.

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

void show_return_status(void)
{
     pid_t    childpid;    int
     status;

     childpid  =  wait(&status);  if  (childpid
     == -1)
          perror("Failed to wait for child");

     else if (WIFEXITED(status))
          printf("Child  %ld  terminated  with  return  status  %d\n",  (long)childpid,
                    WEXITSTATUS(status));

}
```

Example 5

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

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

int main(void)
{
     pid_t childpid;

     childpid = fork();
     if  (childpid  ==  -1)  {  perror("Failed  to
          fork"); return 1;
     }
     if (childpid == 0) {
                                        /*     child     code     */
          execl("/bin/ls",  "ls",  "-l",  NULL);  perror("Child
          failed to exec ls"); return 1;
     }

     if (childpid != wait(NULL)) {
                                   /* parent code */
          perror("Parent failed to wait due to signal or error"); return 1;
     }
```

```
    return 0;
}
```

Example 6
Making a zombie process

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

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

Example 7
Demonstration of fork system call

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

int main()
{
pid_t pid;
char *msg;
int n;
printf("Program starts\n");
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(-1);
case 0:
msg="This is the child process";
n=5;
break;
default:
msg="This is the parent process";
n=3;
break;
}
while(n>0)
{
puts(msg);
sleep(1);
n--;
}
return 0;
}
```

Example 8
Demo of multiprocess application using fork()system call

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

#define SIZE 1024

void do_child_proc(int pfd[2]);
void do_parent_proc(int pfd[2]);

int main()
{
int pfd[2];
int ret_val,nread;
pid_t pid;
ret_val=pipe(pfd);
```

```
if(ret_val==-1)
{
perror("pipe error\n");
exit(ret_val);
}
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(pid);
case 0:
do_child_proc(pfd);
exit(0);
default:
do_parent_proc(pfd);
exit(pid);
}
wait(NULL);

return 0;
}

void do_child_proc(int pfd[2])
{
int nread;
char *buf=NULL;
printf("5\n");
close(pfd[1]);
while(nread=(read(pfd[0],buf,size))!=0)
printf("Child Read=%s\n",buf);
close(pfd[0]);
exit(0);
}
void do_parent_proc(int pfd[2])
 {
char ch;
char *buf=NULL;
close(pfd[0]);
while(ch=getchar()!='\n')  {
printf("7\n");
*buf=ch;
buff+;
}
*buf='\0';
```

```
write(pfd[1],buf,strlen(buf)+1);
close(pfd[1]);
}
```

ASSIGNMENT No: 3

Title : SJF & Round Robin CPU Scheduling

AIM: Implement C program for CPU scheduling algorithms: Shortest Job First (SJF) and Round Robin with different arrival time.

OBJECTIVE: Understand SJF and RR algorithms and implement it in C

THEORY:

Shortest Job First scheduling works on the process with the shortest burst time or duration first.

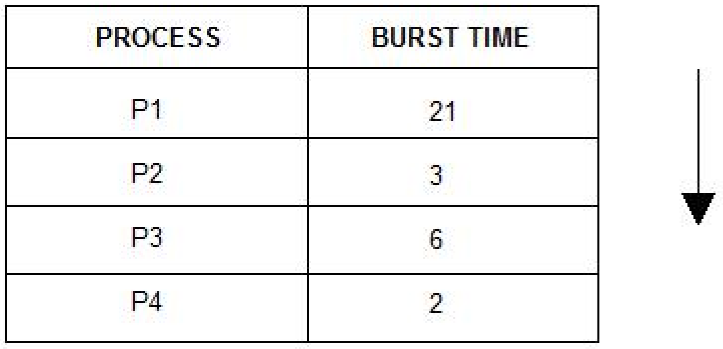
This is the best approach to minimize waiting time.
This is used in Batch Systems.
It is of two types:

1. Non Pre-emptive
2. Pre-emptive

To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

Non Pre-emptive Shortest Job First :

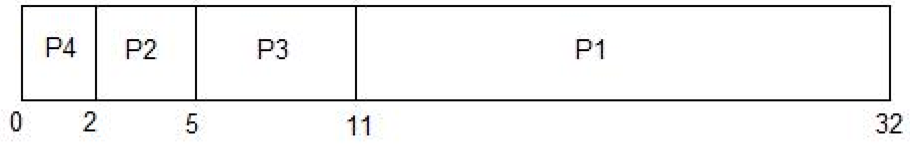
Consider the below processes available in the ready queue for execution, with **arrival time** as 0 for all and given **burst times**.



PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

As you can see in the **GANTT chart** above, the process **P4** will be picked up first as it has the shortest burst time, then **P2**, followed by **P3** and at last **P1**.

We scheduled the same set of processes using the First come first serve algorithm in the previous tutorial, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

Problem with Non Pre-emptive SJF:

If the **arrival time** for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time have to wait for the current process's execution to finish, because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

This leads to the problem of Starvation, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of aging.

Pre-emptive Shortest Job First

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with **short burst time** arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.

The average waiting time will be,((5-3)+(6-2)+(12-1))/4=8.75

The average waiting time for preemptive shortest job first scheduling is less than both,non preemptive SJF scheduling and FCFS scheduling

As you can see in the **GANTT chart** above, as **P1** arrives first, hence it's execution starts immediately, but just after 1 ms, process **P2** arrives with a **burst time** of 3 ms which is less than the burst time of **P1**, hence the process **P1**(1 ms done, 20 ms left) is preemptied and process **P2** is executed.

As **P2** is getting executed, after 1 ms, **P3** arrives, but it has a burst time greater than that of **P2**, hence execution of **P2** continues. But after another millisecond, **P4** arrives with a burst time of 2 ms, as a result **P2**(2 ms done, 1 ms left) is preemptied and **P4** is executed.

After the completion of **P4**, process **P2** is picked up and finishes, then **P2** will get executed and at last **P1**.

The Pre-emptive SJF is also known as **Shortest Remaining Time First**, because at any given point of time, the job with the shortest remaining time is executed first.

Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes.

A fixed time is allotted to each process, called a **quantum**, for execution.

Once a process is executed for the given time period that process is preempted and another process executes for the given time period.

Context switching is used to save states of preempted processes.

This algorithm is simple and easy to implement and the most important is thing is this algorithm is starvation-free as all processes get a fair share of CPU.

It is important to note here that the length of time quantum is generally from 10 to 100 milliseconds in length.

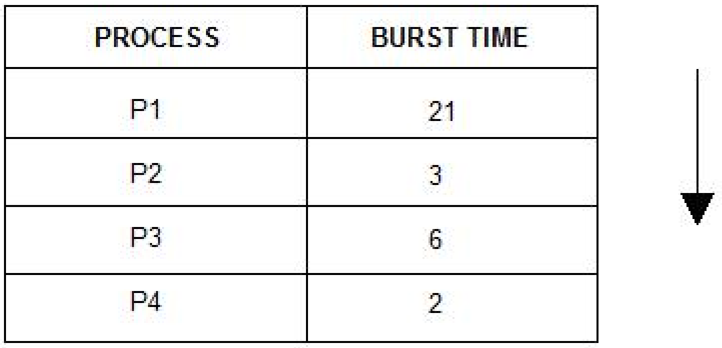
Some important characteristics of the Round Robin(RR) Algorithm are as follows:

1. Round Robin Scheduling algorithm resides under the category of Preemptive Algorithms.
2. This algorithm is one of the oldest, easiest, and fairest algorithm.
3. This Algorithm is a real-time algorithm because it responds to the event within a specific time limit.
4. In this algorithm, the time slice should be the minimum that is assigned to a specific task that needs to be processed. Though it may vary for different operating systems.
5. This is a hybrid model and is clock-driven in nature.
6. This is a widely used scheduling method in the traditional operating system.

Important terms:

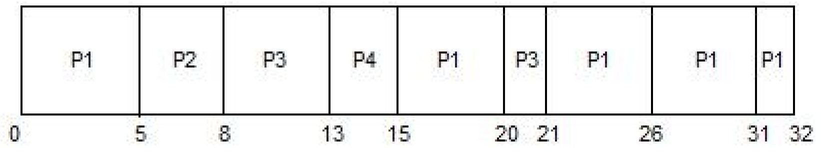
1. **Completion Time** It is the time at which any process completes its execution.
2. **Turn Around Time** This mainly indicates the time Difference between completion time and arrival time. The Formula to calculate the same is: **Turn Around Time = Completion Time – Arrival Time**
3. **Waiting Time(W.T):** It Indicates the time Difference between turn around time and burst time. And is calculated as **Waiting Time = Turn Around Time – Burst Time**

Let us now cover an example for the same:



PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

In the above diagram, arrival time is not mentioned so it is taken as 0 for all processes.

Note: If arrival time is not given for any problem statement then it is taken as 0 for all processes; if it is given then the problem can be solved accordingly.

Explanation:

The value of time quantum in the above example is 5.Let us now calculate the Turn around time and waiting time for the above example :

Processes	Burst Time	Turn Around Time Turn Around Time = Completion Time – Arrival Time	Waiting Time Waiting Time = Turn Around Time – Burst Time
P1	21	32-0=32	32-21=11
P2	3	8-0=8	8-3=5

Processes	**Burst Time**	**Turn Around Time** **Turn Around Time = Completion Time – Arrival Time**	**Waiting Time** **Waiting Time = Turn Around Time – Burst Time**
P3	6	21-0=21	21-6=15
P4	2	15-0=15	15-2=13

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no.of processes.

average waiting time = waiting time of all processes/ no.of processes

average waiting time=11+5+15+13/4 = 44/4= **11ms**

Advantages of Round Robin Scheduling Algorithm:

Some advantages of the Round Robin scheduling algorithm are as follows:

While performing this scheduling algorithm, a particular time quantum is allocated to different jobs.

In terms of average response time, this algorithm gives the best performance.

With the help of this algorithm, all the jobs get a fair allocation of CPU.

In this algorithm, there are no issues of starvation or convoy effect.

This algorithm deals with all processes without any priority.

This algorithm is cyclic in nature.

In this, the newly created process is added to the end of the ready queue.

Also, in this, a round-robin scheduler generally employs time-sharing which means providing each job a time slot or quantum.

In this scheduling algorithm, each process gets a chance to reschedule after a particular quantum time.

Disadvantages of Round Robin Scheduling Algorithm:

This algorithm spends more time on context switches.

For small quantum, it is time-consuming scheduling.

This algorithm offers a larger waiting time and response time.

In this, there is low throughput.

<u>If time quantum is less for scheduling, then its Gantt chart seems to be too big.</u>

ASSIGNMENT NO: 4-A

TITLE: Thread synchronization using counting semaphores.

AIM: Application to demonstrate producer-consumer problem with counting semaphores and mutex.

OBJECTIVE: Implement C program to demonstrate producer-consumer problem with counting semaphores and mutex.

THEORY:

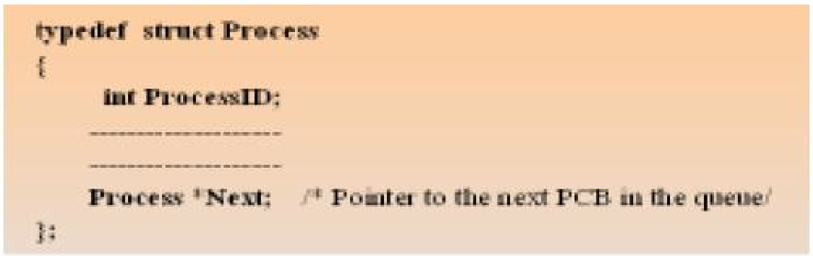
Semaphores:

An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.

Semaphores are the **OS tools** for **synchronization**. Two types:
1. **Binary Semaphore**.
2. **Counting Semaphore**.

Counting semaphore
The counting semaphores are free of the limitations of the binary semaphores. A counting semaphore comprises:
An integer variable, initialized to a value K (K>=0). During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

A counting semaphore can be implemented as follows:



```
typedef  struct Process
{
      int ProcessID;
      ----------------------
      ----------------------
      Process *Next;    /* Pointer to the next PCB in the queue/
};
```

```
typedef  struct Semaphore
{
      int count;
     Process *head  /* Pointer to the head of the queue */
     Process *tail;   /* Pointer to the tail of the queue/
};
Semaphore S;
```

Operation of a counting semaphore:

1. Let the initial value of the semaphore count be 1.
2. When semaphore count = 1, it implies that no process is executing in its critical section and no process is waiting in the semaphore queue.
3. When semaphore count = 0, it implies that one process is executing in its critical section but no process is waiting in the semaphore queue.
4. When semaphore count = N, it implies that one process is executing in its critical section and N processes are waiting in the semaphore queue.
5. When a process is waiting in semaphore queue, it is not performing any busy waiting. It is rather in a “waiting” or “blocked” state.
6. When a waiting process is selected for entry into its critical section, it is transferred from “Blocked” state to “ready” state.

The Producer/Consumer Problem

We now examine one of the most common problems faced in concurrent processing: 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 any one time. The problem is to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer. We will look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores. 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:

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

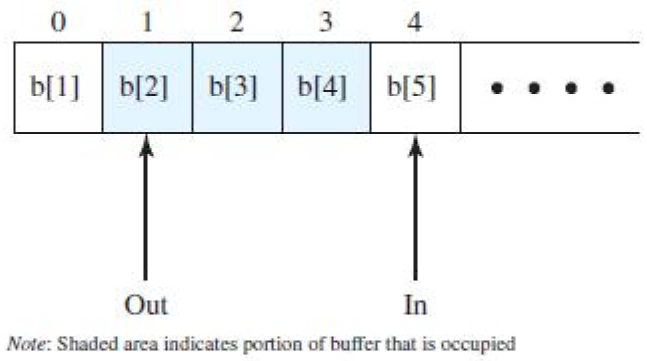

0
1
2
3
4
b[1]
b[2]
b[3]
b[4]
b[5]
• • • •
Out
In
Note: Shaded area indicates portion of buffer that is occupied

Figure: Infinite buffer for producer/consumer problem

CONCLUSION:

Thus, we have implemented producer-consumer problem using ‘C’ in Linux.

FAQ

1. Explain the concept of semaphore?
2. Explain wait and signal functions associated with semaphores.
3. What is meant by binary and counting semaphores?

ASSIGNMENT NO: 4-B

TITLE: Thread synchronization and mutual exclusion using mutex.

AIM: Application to demonstrate Reader-Writer problem with reader priority.

OBJECTIVE: Implement C program to demonstrate Reader-Writer problem with readers having priority using counting semaphores and mutex.

THEORY:

Semaphores:

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

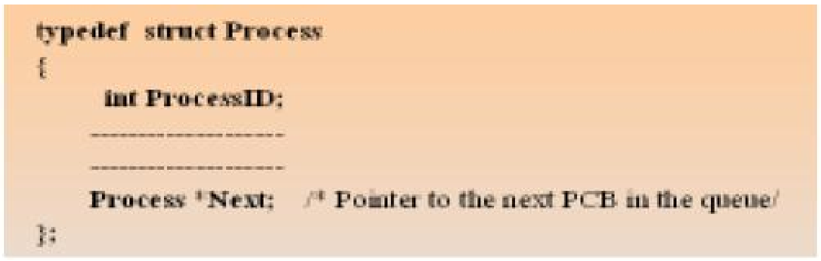
The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **countingsemaphore** or a **general semaphore**.

Semaphores are the **OS tools** for **synchronization**. Two types:
1. **Binary Semaphore**.
2. **Counting Semaphore**.

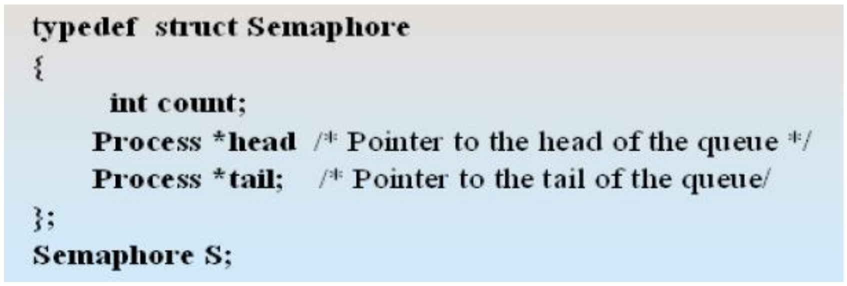
Counting semaphore
The counting semaphores are free of the limitations of the binary semaphores. A counting semaphore comprises:

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

A counting semaphore can be implemented as follows:



```
typedef  struct Process
{
      int ProcessID;
    --------------------
    --------------------
    Process *Next;   /* Pointer to the next PCB in the queue/
};
```



```
typedef  struct Semaphore
{
        int count;
       Process *head  /* Pointer to the head of the queue */
       Process *tail;   /* Pointer to the tail of the queue/
};
Semaphore S;
```

Operation of a counting semaphore:

1. Let the initial value of the semaphore count be 1.
2. When semaphore count = 1, it implies that no process is executing in its critical section and no process is waiting in the semaphore queue.
3. When semaphore count = 0, it implies that one process is executing in its critical section but no process is waiting in the semaphore queue.
4. When semaphore count = N, it implies that one process is executing in its critical section and N processes are waiting in the semaphore queue.
5. When a process is waiting in semaphore queue, it is not performing any busy waiting. It is rather in a "waiting" or "blocked" state.
6. When a waiting process is selected for entry into its critical section, it is trans ferred from "Blocked" state to "ready" state.

Reader-Writer problem with readers priority

The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

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

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike. Before proceeding, let us distinguish this problem from two others: the general mutual exclusion problem and the producer/consumer problem. In the readers/writers problem readers do not also write to the data area, nor do writers read the data area while writing.

A more general case, which includes this case, is to allow any of the processes to read or write the data area. In that case, we can declare any portion of a process that accesses the data area to be a critical section and impose the general mutual exclusion solution. The reason for being concerned with the more restricted case is that more efficient solutions are possible for this case and that the less efficient solutions to the general problem are unacceptably slow.

For example, suppose that the shared area is a library catalog. Ordinary users of the library read the catalog to locate a book. One or more librarians are able to update the catalog.

In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time, it is important to prevent writers from interfering with each other and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

This is not a special case of producer-consumer. The producer is not just a writer. It must read queue pointers to determine where to write the next item, and it must determine if the buffer is full. Similarly, the consumer is not just a reader, because it must adjust the queue pointers to show that it has removed a unit from the buffer.

Solution using semaphore:

```
int readcount;
semaphore x = 1,wsem = 1;

void reader()
{
while (true)
{
semWait (x);
readcount++;
if(readcount == 1)
semWait (wsem);
semSignal (x);
READUNIT();
semWait (x);
readcount;
if(readcount == 0)
semSignal (wsem);
semSignal (x);
}
}

void writer()
{
while (true)
{
semWait (wsem);
WRITEUNIT();
semSignal (wsem);
}
}

void main()
{
readcount = 0;
parbegin (reader,writer);
}
```

Threads

Multiple strands of execution in a single program are called threads. A more precise definition is that a thread is a sequence of control within a process. Like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution.

POSIX Thread in Unix

Including the file pthread.h provides us with other definitions and prototypes that we will need in our code, much like stdio.h for standard input and output routines.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t*attr,
                   void*(*start_routine)(void *), void *arg);
```

This function is used to create the thread. The first argument is a pointer to pthread_t.
When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread. The next argument sets the thread attributes. We do not usually need any special attributes, and we can simply pass NULL as this argument. The final two arguments tell the thread the function that it is to start executing and the arguments that are to be passed to this function.

void *(*start_routine)(void *)

We must pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. Thus, we can pass any type of single argument and return a pointer to any type. Using fork causes execution to continue in the same location with a different return code, whereas using a new thread explicitly provides a pointer to a function where the new thread should start executing. The return value is 0 for success or an error number if anything goes wrong.

When a thread terminates, it calls the pthread_exit function, much as a process calls exit when it terminates. This function terminates the calling thread, returning a pointer to an object. Never use it to return a pointer to a local variable, because the variable will cease to exist when the thread does so, causing a serious bug.

pthread_exit is declared as follows:
#include <pthread.h>

void pthread_exit(void *retval);

pthread_join is the thread equivalent of wait that processes use to collect child processes. This function is declared as follows:
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);

The first parameter is the thread for which to wait, the identifier that pthread_create filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.

Linux Semaphore Facilities (Binary Semaphore)
A semaphore is created with the sem_init function, which is declared as follows:
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, 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>
int sem_wait(sem_t * sem);
int sem_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>

int sem_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.

References :

1. "Beginning Linux Programming" by Neil Mathew and Richard Stones, Wrox Publications.

2. "Operating System Internals and Design Implementation" by William Stallings, Pearson Education.

CONCLUSION:

Thus, we have implemented Reader-Writer problem with readers priority using 'C' in Linux.

FAQ

1. Explain the concept of semaphore?
2. Explain wait and signal functions associated with semaphores.
3. What is meant by binary and counting semaphores?

ASSIGNMENT NO: 5

Title : Bankers Algorithm

AIM: Implement C program for Deadlock Avoidance: Banker's Algorithm

OBJECTIVE: Understand deadlock avoidance and implement it in C

THEORY :

Banker's algorithm is a **deadlock avoidance algorithm**. It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.

Consider there are n account holders in a bank and the sum of the money in all of their accounts is S. Every time a loan has to be granted by the bank, it subtracts the **loan amount** from the **total money** the bank has. Then it checks if that difference is greater than S. It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.

Characteristics of Banker's Algorithm :

The characteristics of Banker's algorithm are as follows:

If any process requests for a resource, then it has to wait.

This algorithm consists of advanced features for maximum resource allocation.

There are limited resources in the system we have.

In this algorithm, if any process gets all the needed resources, then it is that it should return the resources in a restricted period.

Various resources are maintained in this algorithm that can fulfill the needs of at least one client.

Let us assume that there are n processes and m resource types.

Following data structures are required to implement banker's algorithm for deadlock avoidance.

Some data structures that are used to implement the banker's algorithm are:

1. Available
It is an **array** of length m. It represents the number of available resources of each type. If Available[j] = k, then there are k instances available, of resource type Rj.

2. Max
It is an n x m matrix which represents the maximum number of instances of each resource that a process can request. If Max[i][j] = k, then the process Pi can request atmost k instances of resource type Rj.

3. Allocation
It is an n x m matrix which represents the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process Pi is currently allocated k instances of resource type Rj.

4. Need
It is a two-dimensional array. It is an n x m matrix which indicates the remaining resource needs of each process. If Need[i][j] = k, then process Pi may need k more instances of resource type Rj to complete its task.

Need[i][j] = Max[i][j] - Allocation [i][j]

Banker's algorithm comprises of two algorithms:

A) Safety algorithm

B) Resource request algorithm

A) Safety Algorithm

A safety algorithm is an algorithm used to find whether or not a system is in its safe state. The algorithm is as follows:

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initially,

Work = Available
Finish[i] =false for i = 0, 1, ... , n - 1.

This means, initially, no process has finished and the number of available resources is represented by the **Available** array.

2. Find an index **i** such that both

Finish[i] ==false
Needi <= Work

If there is no such i present, then proceed to step 4.

It means, we need to find an unfinished process whose needs can be satisfied by the available resources. If no such process exists, just go to step 4.

3. Perform the following:

Work = Work + Allocationi
Finish[i] = true

Go to step 2.

When an unfinished process is found, then the resources are allocated and the process is marked finished. And then, the loop is repeated to check the same for all other processes.

4. **If Finish[i] == true** for all i, then the system is in a safe state.

That means if all processes are finished, then the system is in safe state.

This algorithm may require an order of **mxn^2 operations** in order to determine whether a state is safe or not.

B. Resource Request Algorithm

Now the next algorithm is a resource-request algorithm and it is mainly used to determine whether requests can be safely granted or not.

Let Requesti be the request vector for the process Pi. If **Requesti[j]==k**, then process Pi wants k instance of Resource type Rj.When a request for resources is made by the process Pi, the following are the actions that will be taken:

1. If **Requesti <= Needi**, then go to step 2;else raise an error condition, since the process has exceeded its maximum claim.

2.If **Requesti <= Availablei** then go to step 3; else **Pi** must have to wait as resources are not available.

3.Now we will assume that resources are assigned to process **Pi** and thus perform the following steps:

Available= Available-Requesti ;

Allocationi=Allocationi +Requesti;

Needi =Needi - Requesti;

If the resulting resource allocation state comes out to be safe, then the transaction is completed and, process Pi is allocated its resources. But in this case, if the new state is unsafe, then Pi waits for Requesti, and the old resource-allocation state is restored.

Disadvantages of Banker's Algorithm

Some disadvantages of this algorithm are as follows:

1. During the time of Processing, this algorithm does not permit a process to change its maximum need.
2. Another disadvantage of this algorithm is that all the processes must know in advance about the maximum resource needs.
3. This algorithm permits the requests to be provided in constrained time, but for one year which is a fixed period.

Working Example

Example:

Let us consider the following snapshot for understanding the banker's algorithm:

Processes	**Allocation A B C**	**Max A B C**	**Available A B C**
P0	**1 1 2**	**4 3 3**	**2 1 0**
P1	**2 1 2**	**3 2 2**	
P2	**4 0 1**	**9 0 2**	
P3	**0 2 0**	**7 5 3**	
P4	**1 1 2**	**1 1 2**	

Solution:

1. The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Process	**Need**		
	A	**B**	**C**
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

. Let us now check for the safe state.

Safe sequence:

1. For process P0, Need = (3, 2, 1) and

Available = (2, 1, 0)

Need <=Available = False

So, the system will move to the next process.

2. For Process P1, Need = (1, 1, 0)

Available = (2, 1, 0)

Need <= Available = True

Request of P1 is granted.

Available = Available +Allocation

= (2, 1, 0) + (2, 1, 2)

= (4, 2, 2) (New Available)

3. For Process P2, Need = (5, 0, 1)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

4. For Process P3, Need = (7, 3, 3)

Available = (4, 2, 2)

Need <=Available = False

So, the system will move to the next process.

5. For Process P4, Need = (0, 0, 0)

Available = (4, 2, 2)

Need <= Available = True

Request of P4 is granted.

Available = Available + Allocation

= (4, 2, 2) + (1, 1, 2)

= (5, 3, 4) now, (New Available)

6. Now again check for Process P2, Need = (5, 0, 1)

Available = (5, 3, 4)

Need <= Available = True

Request of P2 is granted.

Available = Available + Allocation

= (5, 3, 4) + (4, 0, 1)

= (9, 3, 5) now, (New Available)

7. Now again check for Process P3, Need = (7, 3, 3)

Available = (9, 3, 5)

Need <=Available = True

The request for P3 is granted.

Available = Available +Allocation

= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)

8. Now again check for Process P0, = Need (3, 2, 1)

= Available (9, 5, 5)

Need <= Available = True

So, the request will be granted to P0.

Safe sequence: < P1, P4, P2, P3, P0>

The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

3. The total amount of resources

The total amount of resources will be calculated by the following formula:

The total amount of resources= sum of columns of allocation + Available

<u>= [8 5 7] + [2 1 0] = [10 6 7]</u>

ASSIGNMENT NO: 6

Title : Page Replacement Algorithms

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

OBJECTIVE:
This assignment helps the students understand the Page Replacement Algorithms in Unix/Linux and how to implement it in C

THEORY:

What are PAGE REPLACMENT Algorithms?

As studied in Demand Paging, only certain pages of a process are loaded initially into the memory. This allows us to get more processes into memory at the same time. But what happens when a process requests for more pages and no free memory is available to bring them in. Following steps can be taken to deal with this problem:

1. Put the process in the wait queue, until any other process finishes its execution thereby freeing frames.
2. Remove some other process completely from the memory to free frames.
3. Find some pages that are not being used right now, move them to the disk to get free frames. This technique is called Page replacement and is most commonly used.
4.

In this case, if a process requests a new page and supposes there are no free frames, then the Operating system needs to decide which page to replace. The operating system must use any page replacement algorithm in order to select the victim frame. The Operating system must then write the victim frame to the disk then read the desired page into the frame and then update the page tables. And all these require double the disk access time.

1. Page replacement prevents the over-allocation of the memory by modifying the page-fault service routine.
2. To reduce the overhead of page replacement a modify bit (dirty bit) is used in order to indicate whether each page is modified.

3. This technique provides complete separation between logical memory and physical memory.

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

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

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

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

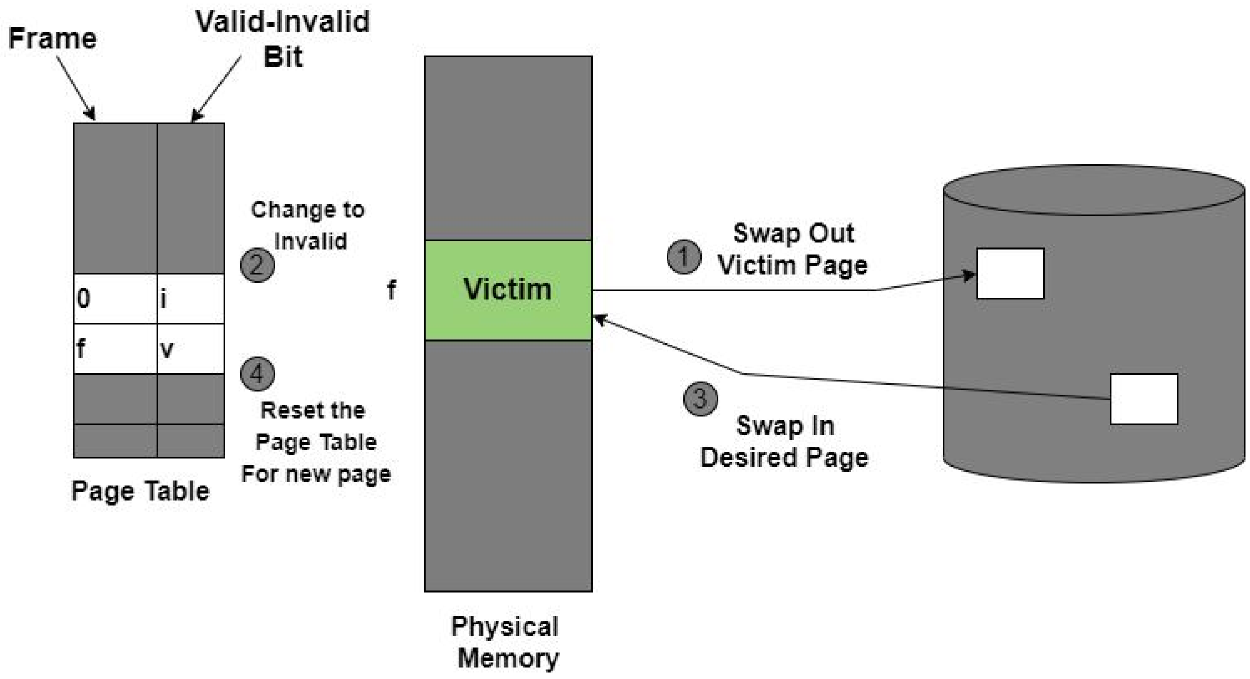
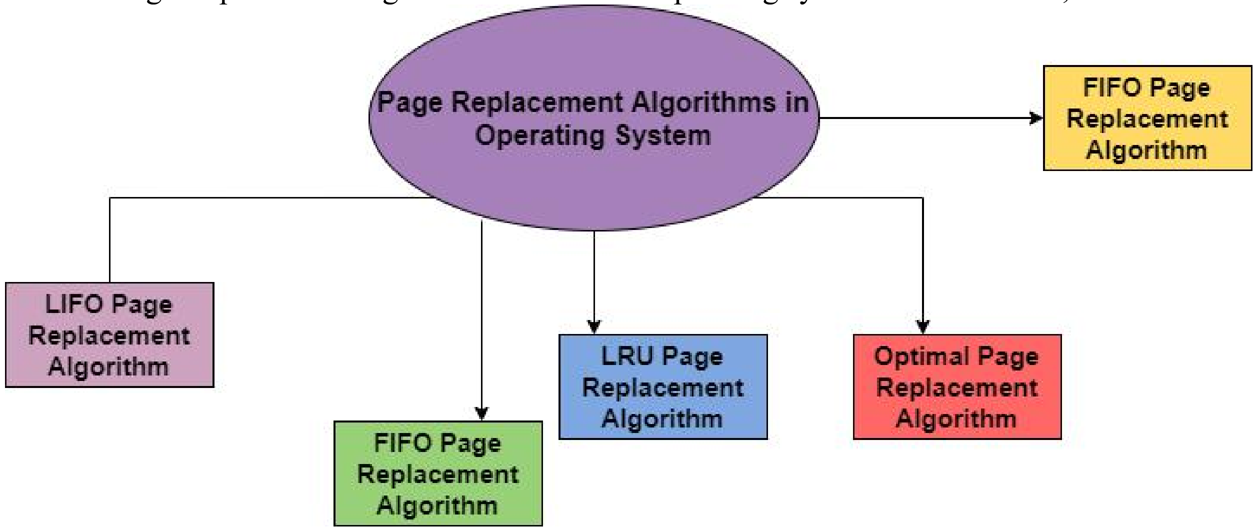


Figure: Page Replacement

Page Replacement Algorithms in OS

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

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


Page Replacement Algorithms in Operating System
FIFO Page Replacement Algorithm
LIFO Page Replacement Algorithm
FIFO Page Replacement Algorithm
LRU Page Replacement Algorithm
Optimal Page Replacement Algorithm

Let us discuss all algorithms one by one in the upcoming sections:

1. FIFO Page Replacement Algorithm

It is a very simple way of Page replacement and is referred to as First in First Out. This algorithm mainly replaces the oldest page that has been present in the main memory for the longest time.
This algorithm is implemented by keeping the track of all the pages in the queue.
As new pages are requested and are swapped in, they are added to the tail of a queue and the page which is at the head becomes the victim.
This is not an effective way of page replacement but it can be used for small systems.

Advantages

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

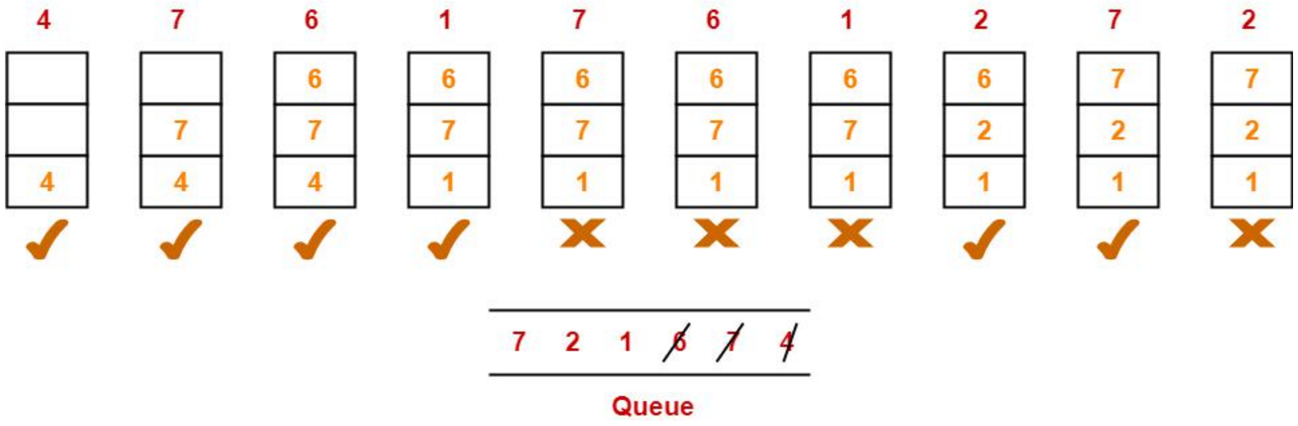
Disadvantages

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

Example:

A system uses 3 page frames for storing process pages in main memory. It uses the First in First out (FIFO) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-
4 , 7, 6, 1, 7, 6, 1, 2, 7, 2
Also calculate the hit ratio and miss ratio.

Solution-
Total number of references = 10



From here,

Total number of page faults occurred = 6

Calculating Hit ratio-
Total number of page hits
= Total number of references – Total number of page misses or page faults
= 10 – 6
= 4

Thus, Hit ratio
= Total number of page hits / Total number of references
= 4 / 10
= 0.4 or 40%

Calculating Miss ratio-
Total number of page misses or page faults = 6
Thus, Miss ratio
= Total number of page misses / Total number of references
= 6 / 10
= 0.6 or 60%

Alternatively,
Miss ratio
= 1 – Hit ratio
= 1 – 0.4
= 0.6 or 60%

2. LRU Page Replacement Algorithm in OS
This algorithm stands for "Least recent used" and this algorithm helps the Operating system to search those pages that are used over a short duration of time frame.

The page that has not been used for the longest time in the main memory will be selected for replacement.
This algorithm is easy to implement.
This algorithm makes use of the counter along with the even-page.

Advantages of LRU

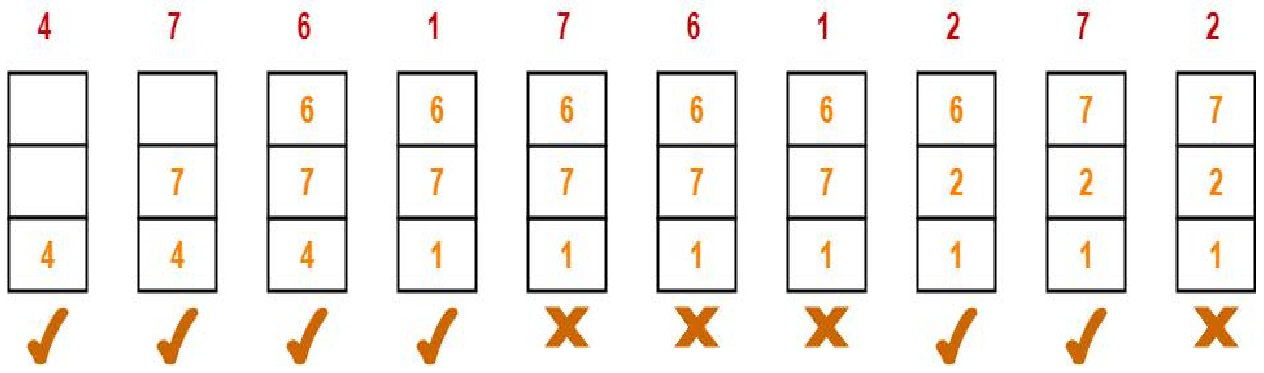
It is an efficient technique.
With this algorithm, it becomes easy to identify the faulty pages that are not needed for a long time.
It helps in Full analysis.

Disadvantages of LRU

It is expensive and has more complexity.
There is a need for an additional data structure.

Example:
A system uses 3 page frames for storing process pages in main memory. It uses the Least Recently Used (LRU) page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-
4 , 7, 6, 1, 7, 6, 1, 2, 7, 2
Also calculate the hit ratio and miss ratio.

<u>Solution-</u>
Total number of references = 10



From here,
Total number of page faults occurred = 6

In the similar manner as above-
Hit ratio = 0.4 or 40%
Miss ratio = 0.6 or 60%

3. Optimal Page Replacement Algorithm

This algorithm mainly replaces the page that will not be used for the longest time in the future. The practical implementation of this algorithm is not possible.

Practical implementation is not possible because we cannot predict in advance those pages that will not be used for the longest time in the future.

This algorithm leads to less number of page faults and thus is the best-known algorithm

Also, this algorithm can be used to measure the performance of other algorithms.

Advantages of OPR

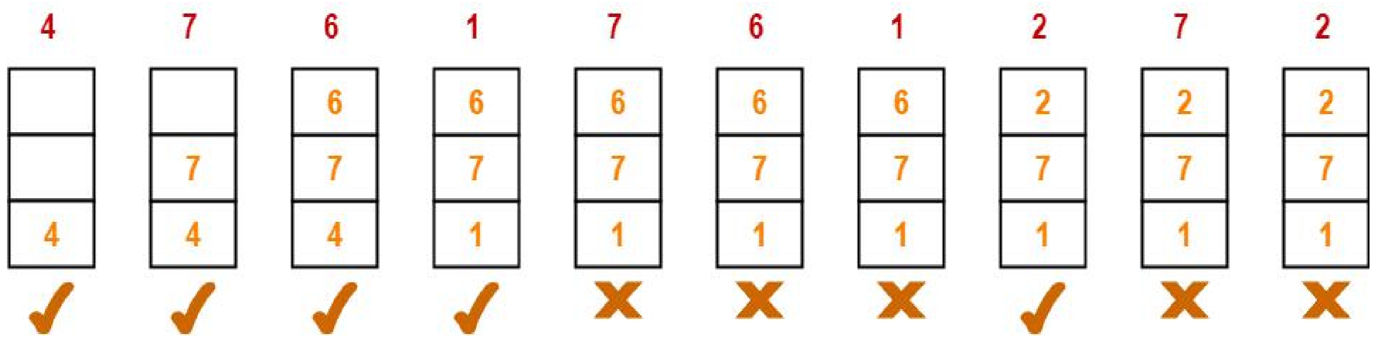
This algorithm is easy to use.
This algorithm provides excellent efficiency and is less complex.
For the best result, the implementation of data structures is very easy

Disadvantages of OPR

In this algorithm future awareness of the program is needed.
Practical Implementation is not possible because the operating system is unable to track the future request

Example:
A system uses 3 page frames for storing process pages in main memory. It uses the Optimal page replacement policy. Assume that all the page frames are initially empty. What is the total number of page faults that will occur while processing the page reference string given below-
4 , 7, 6, 1, 7, 6, 1, 2, 7, 2
Also calculate the hit ratio and miss ratio.

<u>Solution-</u>
Total number of references = 10



From here,
Total number of page faults occurred = 5
In the similar manner as above-
Hit ratio = 0.5 or 50%
Miss ratio = 0.5 or 50%

ASSIGNMENT NO: 7-A

Title : Inter process communication

AIM: Inter process communication in Linux using FIFOs.

OBJECTIVES:

Implementation of 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.

THEORY:

FIFOs

A first-in, first-out (FIFO) file is a pipe that has a name in the filesystem. Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other. FIFOs are also called named pipes

You can make a FIFO using the mkfifo command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-
1 samuel          users               0 Jan 16 14:04 /tmp/fifo
```

The first character of the output from ls is p, indicating that this file is actually a FIFO (named pipe). In one window, read from the FIFO by invoking the following:

```
% cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
% cat > /tmp/fifo
```

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window. Remove the FIFO with this line:

```
% rm /tmp/fifo
```

Creating a FIFO

Create a FIFO programmatically using the mkfifo function.The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions, and a pipe must have a reader and a writer, the permissions must include both read and write permissions. If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns –1. Include<sys/types.h> and <sys/stat.h> if you call mkfifo.

Accessing a FIFO

Access a FIFO just like an ordinary file. To communicate through a FIFO, one program must open it for writing, and another program must open it for reading. Either low-level I/O functions like open, write, read, close or C library I/O functions (fopen, fprintf, fscanf, fclose, and soon) may be used.
For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
intfd = open (fifo_path, O_WRONLY);

write (fd, data, data_length);

close (fd);
```

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

```
FILE* fifo = fopen (fifo_path, “r”);

fscanf (fifo, “%s”, buffer);

fclose (fifo);
```

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

CONCLUSION:

Thus, we studied inter process communication using FIFOs.

ASSIGNMENT No: 7-B

TITLE: Inter-process Communication using Shared Memory using System V.

AIM: Application to demonstrate: Client and Server Programs in which server process creates a shared memory segment and write the message to the shared memory segment. Client process reads the message from the shared memory segment and displays it to the screen.

THEORY:

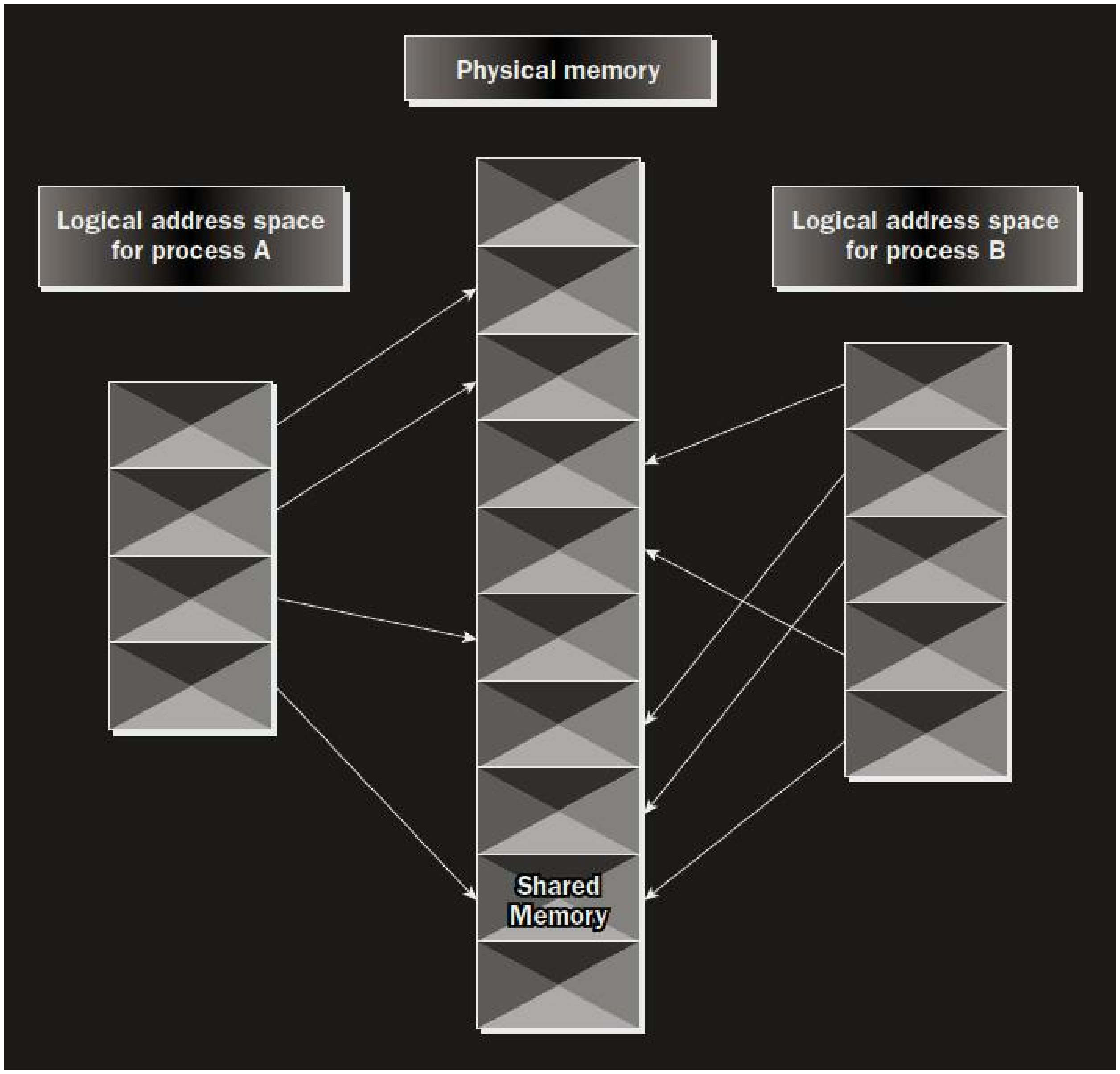
Shared Memory

Shared memory allows two unrelated processes to access the same logical memory. Shared memory is a very efficient way of transferring data between two running processes. Shared memory is a special range of addresses that is created by IPC for one process and appears in the address space of that process.

Other processes can then "attach" the same shared memory segment into their own address space. All processes can access the memory locations just as if the memory had been allocated by malloc. If one process writes to the shared memory, the changes immediately become visible to any other process that has access to the same shared memory.

Shared memory provides an efficient way of sharing and passing data between multiple processes. By itself, shared memory doesn't provide any synchronization facilities. Because it provides no synchronization facilities, we usually need to use some other mechanism to synchronize access to the shared memory.

Typically, we use shared memory to provide efficient access to large areas of memory and pass small messages to synchronize access to that memory. There are no automatic facilities to prevent a second process from starting to read the shared memory before the first process has finished writing to it. It's the responsibility of the programmer to synchronize access. Figure below shows an illustration of how shared memory works.


Physical memory
Logical address space for process A
Logical address space for process B
Shared Memory

The arrows show the mapping of the logical address space of each process to the physical memory available. In practice, the situation is more complex because the available memory actually consists of a mix of physical memory and memory pages that have been swapped out to disk. The functions for shared memory resemble those for semaphores:

#include <sys/shm.h>

void *shmat(int shm_id, const void *shm_addr, int shmflg);

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

int shmdt(const void *shm_addr);

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

As with semaphores, the include files sys/types.h and sys/ipc.h are normally automatically included by shm.h.

shmget() It is used to create shared memory

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

As with semaphores, the program provides key, which effectively names the shared memory segment, and the shmget function returns a shared memory identifier that is used in subsequent shared memory functions. There's a special key value, IPC_PRIVATE, that creates shared memory private to the process.The second parameter, size, specifies the amount of memory required in bytes.The third parameter, shmflg, consists of nine permission flags that are used in the same way as the mode flags for creating files. A special bit defined by IPC_CREAT must be bitwise ORed with the permissions to create a new shared memory segment. It's not an error to have the IPC_CREAT flag set and pass the key of an existing shared memory segment. The IPC_CREAT flag is silently ignored if it is not required.If the shared memory is successfully created, shmget returns a nonnegative integer, the shared memory identifier. On failure, it returns –1.

shmat()

When we first create a shared memory segment, it's not accessible by any process. Toenable access to the shared memory, we must attach it to the address space of a process. We do this with the shmat function:

void *shmat(int shm_id, const void *shm_addr, int shmflg);

The first parameter, shm_id, is the shared memory identifier returned from shmget. The second parameter, shm_addr, is the address at which the shared memory is to be attached to the current process. This should almost always be a null pointer, which allows the system to choose the address at which the memory appears.The third parameter, shmflg, is a set of bitwise flags. The two possible values are SHM_R for writing and SHM_W for write access. If the shmat call is successful, it returns a pointer to the first byte of shared memory. On failure –1 is returned.

shmctl( )

it is used for controlling functions for shared memory.

int shmctl(int shm_id, int command, struct shmid_ds *buf);

The shmid_ds structure has at least the following members:

struct shmid_ds{uid_t shm_perm.uid;uid_t shm_perm.gid;

mode_t shm_perm.mode;}

The first parameter, shm_id, is the identifier returned from shmget. The second parameter,command, is the action to take. It can take three values, shown in the following table.

Command	**Description**
IPC_STAT :	Sets the data in the shmid_ds structure to reflect the values associated with the shared memory.
IPC_SET:	Sets the values associated with the shared memory to those provided in the shmid_ds data structure, if the process has permission to do so.
IPC_RMID:	Deletes the shared memory segment.

The third parameter, buf, is a pointer to the structure containing the modes and permissions for the shared memory.

References :

1. “Beginning Linux Programming” by Neil Mathew and Richard Stones, Wrox Publications.

2. “Operating System Internals and Design Implementation” by William Stallings, Pearson Education.

ASSIGNMENT No: 8

Title : Disk scheduling algorithms

AIM : To implement C programs for Disk scheduling algorithms

1. SSTF
2. SCAN
3. C-LOOK

Theory :

INTRODUCTION

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.

PURPOSE

The purpose of this material is to provide one with help on disk scheduling algorithms. Hopefully with this, one will be able to get a stronger grasp of what disk scheduling algorithms do.

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.

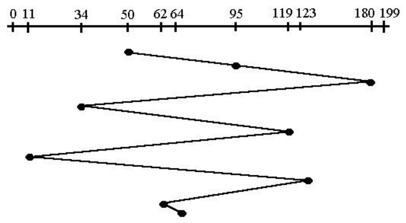


Figure 1: First Come -First Serve (FCFS)

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.

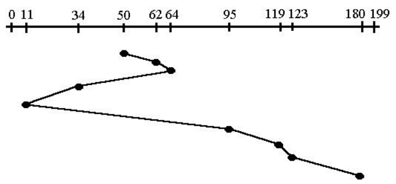


Figure 2: Shortest Seek Time First (SSTF)

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.

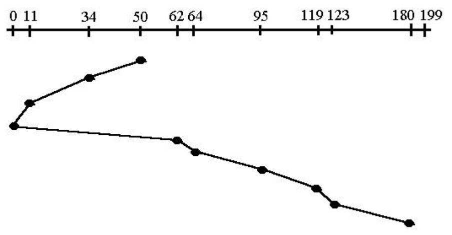


Figure 3. Elevator (SCAN)

4. Circular Scan (C-SCAN)

Circular scanning works just like the elevator to some extent. It begins its scan toward the nearest end and works it way all the way to the end of the system. Once it hits the bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for this algorithm is only 187 track, but still this isn't the mose sufficient.

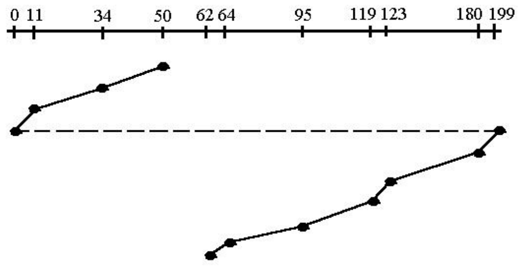


Figure 4: Circular Scan (C-SCAN)

5. C-LOOK

This 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.

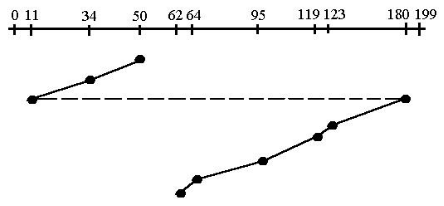


Figure 5: C-LOOK

Assignment:

Given the following queue -- 195, 80, 134, 19, 111, 132, 162, 164 with the Read-write head initially at the track 150 and the tail track being at 199, implement disk scheduling to calculate head movements for SSTF, SCAN, C-LOOK

ASSIGNMENT No: 9

Title : System Call

AIM : Implement a new system call, add this new system call in the Linux kernel (any kernel source, any architecture and any Linux kernel distribution) and demonstrate the use of same.

OBJECTIVE: add a new system call, swipe(), to the Linux kernel that transfers the remaining time slice of each process in a specified set to a target process. You will also demonstrate various uses of the system call (both advantageous and detrimental)

THEORY:

Adding a simple system call

1. Download the kernel source:

In your terminal type the following command:

```
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.17.4.tar.xz
```

Else go to kernel.org and download the latest version.

2. Extract the kernel source code

```
sudo tar -xvf linux-4.17.4.tar.xz -C/usr/src/
```

tar — Tar stores and extracts files from a tape or disk archive.

-x — extract files from an archive

-v — requested using the –verbose option, when extracting archives

-f — file archive; use archive file or device archive

-C — extract to the directory specified after it.(in this case /usr/src/)

Now, we'll change the directory to where the files are extracted:

```
cd /usr/src/linux-4.17.4/
```

3. Define a new system call sys_hello( )

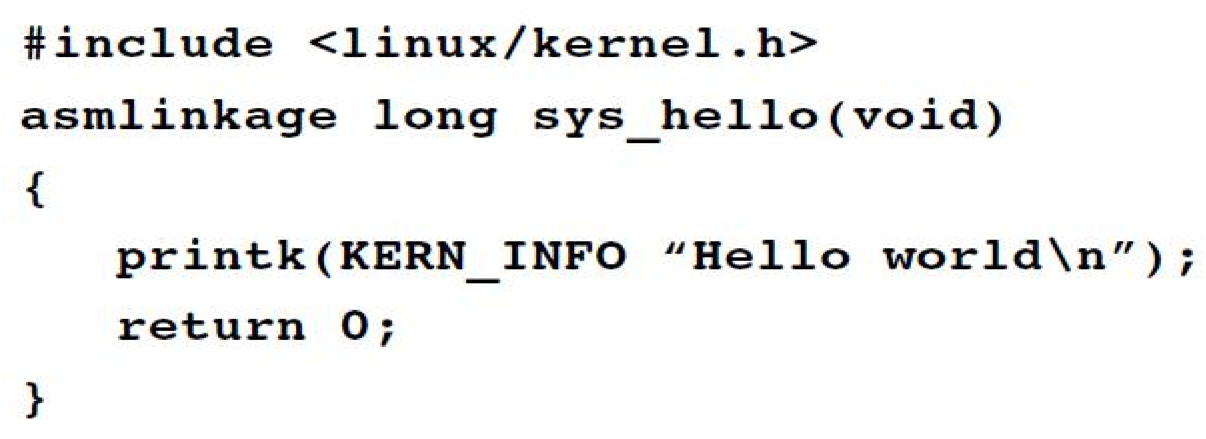
Create a directory named **hello/** and change the directory to hello/:

```
mkdir hello
cd hello
```

Create a file **hello.c** using text editor:

gedit hello.c

Write the following code in the editor:



```
#include <linux/kernel.h>
asmlinkage long sys_hello(void)
{
    printk(KERN_INFO "Hello world\n");
    return 0;
}
```

printk prints to the kernel's log file.

Create a "Makefile" in the hello directory:

gedit Makefile

and add the following line to it:

obj-y := hello.o

This is to ensure that the hello.c file is compiled and included in the kernel source code.

Note: There is no space in between"obj-y".

4. Adding hello/ to the kernel's Makefile:

Go back to the parent dir i.e. **cd ../** and open "Makefile"

gedit Makefile

search for core-y in the document, you'll find this line as the second instance of your search:

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

Add 'hello/' to the end of this line:

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

Note: There is a space between "block/" and "hello/". (Doing such a mistake may cause errors in further steps)

This is to tell the compiler that the source files of our new system call (sys_hello()) are in present in the hello directory.

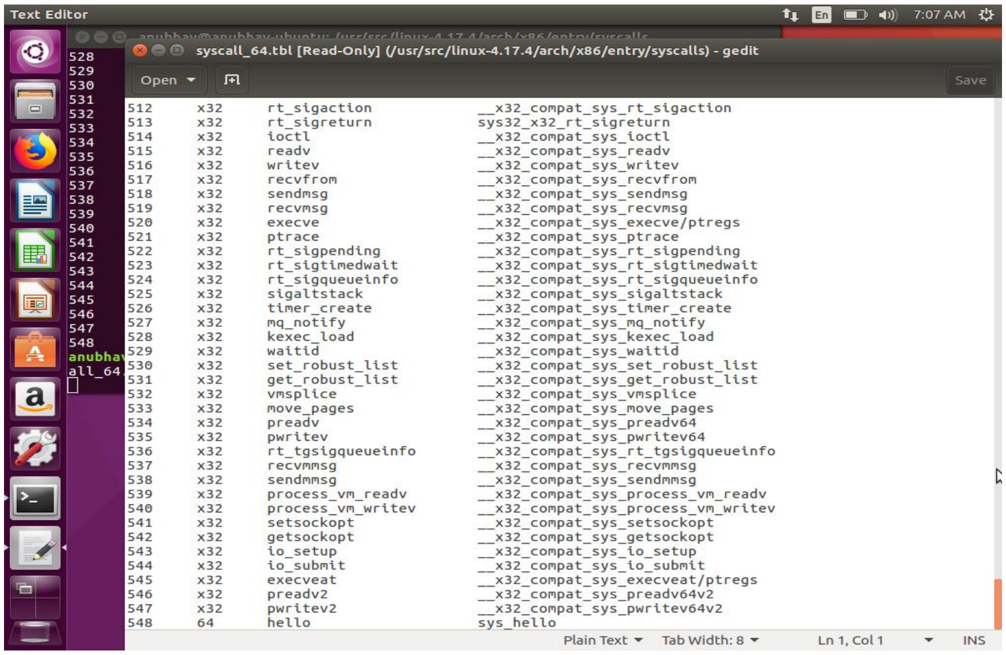
5. Add the new system call to the system call table:

If you are on a 32-bit system you'll need to change 'syscall_32.tbl'. For 64-bit, change 'syscall_64.tbl'.

Run the following commands in your terminal from linux-4.17.4/ directory:

```
cd arch/x86/entry/syscalls/
gedit syscall_64.tbl
```

You'll get a file like the following in your editor:



syscall_64.tbl [Read-Only] (/usr/src/linux-4.17.4/arch/x86/entry/syscalls) - gedit

```
512     x32     rt_sigaction            __x32_compat_sys_rt_sigaction
513     x32     rt_sigreturn            sys32_x32_rt_sigreturn
514     x32     ioctl                   __x32_compat_sys_ioctl
515     x32     readv                   __x32_compat_sys_readv
516     x32     writev                  __x32_compat_sys_writev
517     x32     recvfrom                __x32_compat_sys_recvfrom
518     x32     sendmsg                 __x32_compat_sys_sendmsg
519     x32     recvmsg                 __x32_compat_sys_recvmsg
520     x32     execve                  __x32_compat_sys_execve/ptregs
521     x32     ptrace                  __x32_compat_sys_ptrace
522     x32     rt_sigpending           __x32_compat_sys_rt_sigpending
523     x32     rt_sigtimedwait         __x32_compat_sys_rt_sigtimedwait
524     x32     rt_sigqueueinfo         __x32_compat_sys_rt_sigqueueinfo
525     x32     sigaltstack             __x32_compat_sys_sigaltstack
526     x32     timer_create            __x32_compat_sys_timer_create
527     x32     mq_notify               __x32_compat_sys_mq_notify
528     x32     kexec_load              __x32_compat_sys_kexec_load
529     x32     waitid                  __x32_compat_sys_waitid
530     x32     set_robust_list         __x32_compat_sys_set_robust_list
531     x32     get_robust_list         __x32_compat_sys_get_robust_list
532     x32     vmsplice                __x32_compat_sys_vmsplice
533     x32     move_pages              __x32_compat_sys_move_pages
534     x32     preadv                  __x32_compat_sys_preadv64
535     x32     pwritev                 __x32_compat_sys_pwritev64
536     x32     rt_tgsigqueueinfo       __x32_compat_sys_rt_tgsigqueueinfo
537     x32     recvmmsg                __x32_compat_sys_recvmmsg
538     x32     sendmmsg                __x32_compat_sys_sendmmsg
539     x32     process_vm_readv        __x32_compat_sys_process_vm_readv
540     x32     process_vm_writev       __x32_compat_sys_process_vm_writev
541     x32     setsockopt              __x32_compat_sys_setsockopt
542     x32     getsockopt              __x32_compat_sys_getsockopt
543     x32     io_setup                __x32_compat_sys_io_setup
544     x32     io_submit               __x32_compat_sys_io_submit
545     x32     execveat                __x32_compat_sys_execveat/ptregs
546     x32     preadv2                 __x32_compat_sys_preadv64v2
547     x32     pwritev2                __x32_compat_sys_pwritev64v2
548     64      hello                   sys_hello
```

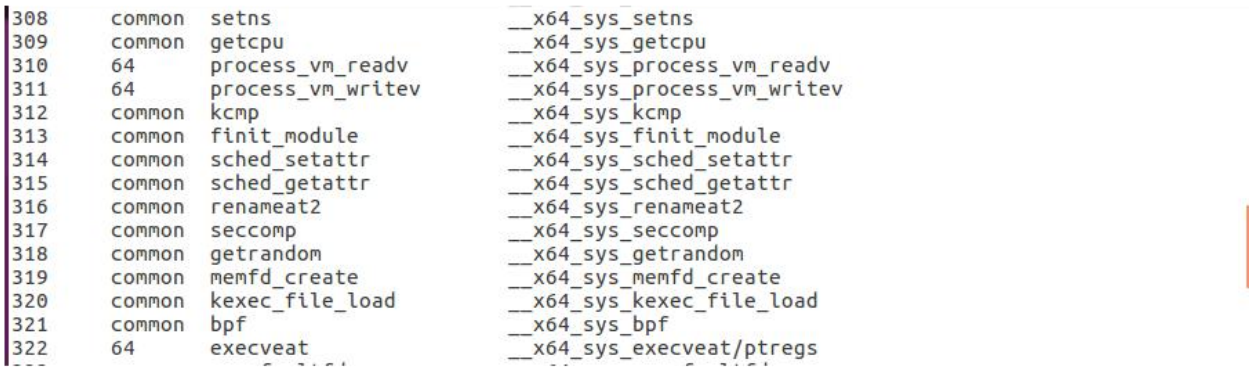
Go to the last of the document and add a new line like so:

```
548      64      hello       sys_hello
```

Note:

Here 548 is written because in the previous line the number entry was 547. **Remember this number it will be used in the later steps.**

Also, note that I've written 64 in my system because it is 64 bit. You may have to write i586 or x32. For knowing what is to be written check in this file itself in many of the lines you may find entries like so:



```
308     common  setns                    __x64_sys_setns
309     common  getcpu                   __x64_sys_getcpu
310     64      process_vm_readv         __x64_sys_process_vm_readv
311     64      process_vm_writev        __x64_sys_process_vm_writev
312     common  kcmp                     __x64_sys_kcmp
313     common  finit_module             __x64_sys_finit_module
314     common  sched_setattr            __x64_sys_sched_setattr
315     common  sched_getattr            __x64_sys_sched_getattr
316     common  renameat2                __x64_sys_renameat2
317     common  seccomp                  __x64_sys_seccomp
318     common  getrandom                __x64_sys_getrandom
319     common  memfd_create             __x64_sys_memfd_create
320     common  kexec_file_load          __x64_sys_kexec_file_load
321     common  bpf                      __x64_sys_bpf
322     64      execveat                 __x64_sys_execveat/ptregs
```

64 written at 310, 311 and 322 line numbers

This will tell you whether to write i586 or something else.

Save and exit.

6. Add new system call to the system call header file:

Go to the **linux-4.17.4/** directory and type the following commands:

```
cd include/linux/
gedit syscalls.h
```

Add the following line to the end of the document before the #endif statement:

asmlinkage long sys_hello(void);

Save and exit. This defines the prototype of the function of our system call. "asmlinkage" is a key word used to indicate that all parameters of the function would be available on the stack.

7. Compile the kernel:

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

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

to configure your kernel use the following command in your **linux-4.17.4/** directory:

```
sudo make menuconfig
```

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.

Now to compile the kernel you can use the make command:

```
sudo make
```

Pro Tip: *The make command can take a lot of time in compiling, to speed up the process you can take advantage of the multiple cores that our systems have these days. Simply type,*

```
sudo make -jn
```

where n is the number of cores that you have in your linux system. For example if you have a Quad core(4) processor, you can write:

```
sudo make -j4
```

this will speed up my make process 4x times. ;)

This might take an hours or more depending on your system.

8. Install / update Kernel:

Run the following command in your terminal:

```
sudo make modules_install install
```

It 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.

1. System.map-4.17.4
2. vmlinuz-4.17.4
3. initrd.img-4.17.4
4. config-4.17.4

Now to update the kernel in your system reboot the system . You can use the following command:

```
shutdown -r now
```

After rebooting you can verify the kernel version using the following command:

```
uname -r
```

It will display the kernel version like so:

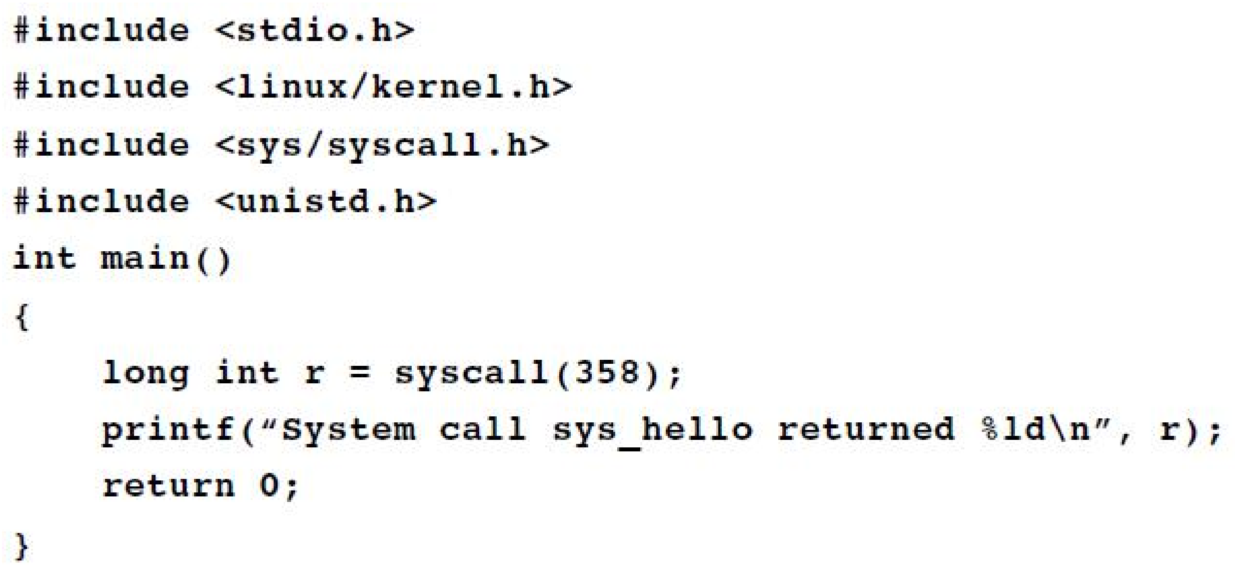
```
4.17.4
```

9. Test system call:

Go to your home(~) directory using the following commands and create a **userspace.c** file.

```
cd ~
gedit userspace.c
```

Write the following code in this file:



```
#include <stdio.h>
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>
int main()
{
    long int r = syscall(358);
    printf("System call sys_hello returned %ld\n", r);
    return 0;
}
```

Note: Remember to keep in mind the number of system call that is added in syscalls_64.tbl? In my case the number was 548. Write that same number in your userspace.c file as an argument in syscall() function.

Now, compile and run the program:

```
gcc userspace.c
./a.out
```

If all the steps are done correctly you'll get an output like below:

```
System call sys_hello returned 0
```

Now, to check the message of your kernel run the following command:

```
dmesg
```

This will display *Hello world* at the end of the kernel's message.