Lab Manual
Software Lab – II

TE – Information Technology

Pune Institute of Computer Technology, Pune - 411043

ASSIGNMENT No: 1

Shell Programming

AIM:

Write a program to implement an address book with options given below:

1. a) Create address book.
2. b) View address book.
3. c) Insert a record.
4. d) Delete a record.
5. e) Modify a record.
6. 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

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 that uses this sorted array for performing the binary search to search the particular item in the array.

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

AIM : Matrix Multiplication using POSIX pthreads

OBJECTIVES: To implement C program for Matrix Multiplication using POSIX pthreads

THEORY:

What is thread?

A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

What are pthreads?

Historically, hardware vendors have implemented their own proprietary versions of threads.These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.

Why pthreads?

The primary motivation for using Pthreads is to realize potential program performance gains.

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

    a. Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.

    b. Priority/real-time scheduling: tasks, which are more important, can be scheduled to supersede or interrupt lower priority tasks.

    c. Asynchronous event handling: tasks, which service events of indeterminate frequency and duration, can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

- Multi-threaded applications will work on a uniprocessor system; yet naturally take advantage of a multiprocessor system, without recompiling.

- In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism. This will be the focus of the remainder of this session.

The pthreads API :

The subroutines which comprise the Pthreads API can be informally grouped into three major classes:

1. **Thread management:** The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutexes:** The second class of functions deal with a coarse type of synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. **Condition variables:** The third class of functions deal with a finer type of synchronization - based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming conventions: All identifiers in the threads library begin with pthread_

pthread_	Threads themselves and miscellaneous subroutines
pthread_t	Thread objects
pthread_attr	Thread attributes objects
pthread_mutex	Mutexes
pthread_mutexattr	Mutex attributes objects.
pthread_cond	Condition variables
pthread_condattr	Condition attributes objects
pthread_key	Thread-specific data keys

Thread Management Functions:

The function pthread_create is used to create a new thread, and a thread to terminate itself uses the function pthread_exit. A thread to wait for termination of another thread uses the function pthread_join

Function:	int pthread_create (pthread_t * threadhandle, */* Thread handle returned by reference */* pthread_attr_t *attribute, */* Special Attribute for starting thread, may be NULL */* void *(*start_routine)(void *), */* Main Function which thread executes */* void *arg */* An extra argument passed as a pointer */* *);*
Info:	Request the PThread library for creation of a new thread. The return value is 0 on success. The return value is negative on failure. The pthread_t is an abstract

	datatype that is used as a handle to reference the thread.

Function:	void pthread_exit (void *retval /* return value passed as a pointer */);
Info:	This Function is used by a thread to terminate. The return value is passed as a pointer. This pointer value can be anything so long as it does not exceed the size of (void *). Be careful, this is system dependent. You may wish to return an address of a structure, if the returned data is very large.

Function:	int pthread_join (pthread_t threadhandle, /* Pass threadhandle */ void **returnvalue /* Return value is returned by ref. */);
Info:	Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.

Thread Initialization:

Include the pthread.h library :
#include <pthread.h>

Declare a variable of type
pthread_t : pthread_t the_thread

When you compile, add -lpthread to the linker flags :
cc or gcc threads.c -o threads -lpthread

Initially, threads are created from within a process. Once created, threads are peers, and may create other threads. Note that an "initial thread" exists by default and is the thread, which runs main.

Terminating Thread Execution:

int pthread_cancel (pthread_t thread)

`pthread_cancel` sends a cancellation request to the thread denoted by the thread argument. If there is no such thread, **`pthread_cancel`** fails. Otherwise it returns 0.

A cancel is a mechanism by which a calling thread informs either itself or the called thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread receives or handles the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

The programmer may specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.

There are several ways in which a Pthread may be terminated:

The thread returns from its starting routine (the main routine for the initial thread). By default, the Pthreads library will reclaim any system resources used by the thread. This is similar to a process terminating when it reaches the end of main.
The thread makes a call to the **`pthread_exit`** subroutine.
The thread is canceled by another thread via the **`pthread_cancel`** routine (not covered here).

The thread receives a signal that terminates it the entire process is terminated due to a call to either the **exec** or **exit** subroutines.

Thread Attributes:

Threads have a number of attributes that may be set at creation time. This is done by filling a thread attribute object attr of type pthread_attr_t, then passing it as second argument to

pthread_create. Passing NULL is equivalent to passing a thread attribute object with all attributes set to their default values. Attribute objects are consulted only when creating a new thread. The same attribute object can be used for creating several threads. Modifying an attribute object after a call to pthread_create does not change the attributes of the thread previously created.

int pthread_attr_init (pthread_attr_t *attr)

pthread_attr_init initializes the thread attribute object attr and fills it with default values for the attributes. Each attribute attrname can be individually set using the function pthread_attr_setattrname and retrieved using the function pthread_attr_getattrname.

int pthread_attr_destroy (pthread_attr_t *attr)

pthread_attr_destroy destroys the attribute object pointed to by attr releasing any resources associated with it. attr is left in an undefined state, and you must not use it again in a call to any pthreads function until it has been reinitialized.

int pthread_attr_setattr (pthread_attr_t *obj, int value)

Set attribute attr to value in the attribute object pointed to by obj. See below for a list of possible attributes and the values they can take. On success, these functions return 0.

int pthread_attr_getattr (const pthread_attr_t *obj, int *value)

Store the current setting of **attr** in **obj** into the variable pointed to by value. These functions always return 0.

INPUT: Accept two matrices for multiplication.

OUTPUT: result of matrix multiplication performed by using pthreads is stored in resultant matrix.

FAQ:

1. What is thread?
2. What is the difference between thread & process?
3. how threads are created using pthread?
4. what is the use of pthread_join()function?

Example 1: Pthread Creation and Termination:

```
#include <stdio.h>
#include <pthread.h>
void *kidfunc(void *p)
{
        printf ("Kid ID is ---> %d\n", getpid( ));
}
```

```
main ( )
{
        pthread_t kid ;
        pthread_create (&kid, NULL, kidfunc, NULL) ;
        printf ("Parent ID is ---> %d\n", getpid( )) ;
        pthread_join (kid, NULL) ;
        printf ("No more kid!\n") ;
}
```

Sample output

```
Parent ID is ---> 29085
 Kid ID is ---> 29085
 No more kid!
```

Example: 2

```
/* Multithreaded C Program Using the Pthread API */
#include<pthread.h>
#include<stdio.h>
int sum; /*This data is shared by the thread(s) */
void *runner(void *param); /* the thread */
main(int argc, char *argv[]) {
        pthread_t tid; /* the thread identifier */
        pthread_attr_t attr; /* set of thread attributes */
        if(argc != 2) {
                fprintf(stderr,"usage: a.out <integer value>\n");
                exit();
        }
        if(atoi(argv[1]) < 0)
        {
                fprintf(stderr, "%d must be >= 0 \n", atoi(argv[1]));
                exit();
        }
        /* get the default attributes */
        pthread_attr_init(&attr);
        /*create the thread */
        pthread_create(&tid,&attr,runner,argv[1]);
        /* Now wait for the thread to exit */
        pthread_join(tid,NULL);
        printf("sum = %d\n",sum);
}
/*The thread will begin control in this function */
void *runner(void *param)
{
        int upper = atoi(param);
        int i;
        sum=0;
        if(upper > 0)
        {
                for(i=1; i <= upper;i++)
```

```
                sum += i;
        }
        pthread_exit(0);
}
```

Sample Output:

```
vlsi> gcc lab3.c -o lab3 -lpthread
vlsi> lab3 10
sum = 55
vlsi>
```

Explanation:

Above Program creates a separate thread that determines the summation of a non-negative integer. In a thread program, separate thread begin execution in a specified function . In above program it is the runner function. When this program begins, a single thread of control begins in main. After some initialization, main creates a second thread that begins control in the summer function. All Pthread programs must include the pthread.h header file. The pthread_attr_t attr declaration represents the attributes for the thread. Because we did not explicitly set any attributes, we will use the default attribute provided. A separate thread is created with the pthread_create function call . In addition to passing the thread identifier and the attributes for the thread. We also pass the name of the function where the new thread will execution, in this case runner function. Lastly , we pass the integer parameter that was provided on the command line, argv[1].

At this point , the program has two threads : the initial thread in main and the thread performing the summation in the runner function. After creating the second thread, the main thread will wait for the runner thread to complete by calling the pthread_join function. The runner thread will complete when it calls the function pthread_exit.

ASSIGNMENT NO: 4

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:

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

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

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

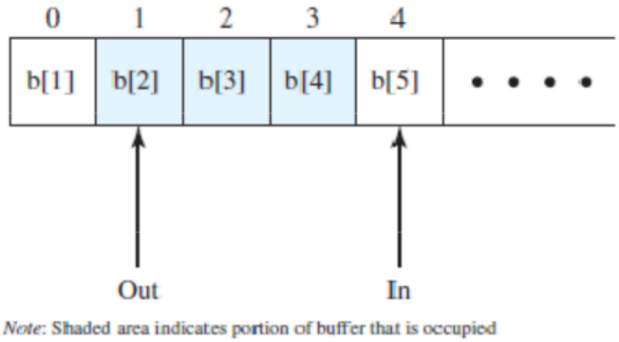


Figure: Infinite buffer for producer/consumer problem

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: 5

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: 6

AIM: To implement Dining Philosopher's problem using 'C' in Linux

OBJECTIVE: Implement the deadlock-free solution to Dining Philosophers problem to illustrate the problem of deadlock and/or starvation that can occur when many synchronized threads are competing for limited resources.

THEORY:

What is Deadlock?

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

Conditions for Deadlock

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

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

In this problem, there are 5 philosophers present who spend their life in eating and thinking. Philosophers share a common circular table surrounded by 5 chairs, each belonging to 1 philosopher .In the center of the table. A bowl of slippery food(Noodles or rice) is present and across each philosopher a pair of chopstick is present. When a philosopher thinks, he does not interact with his colleague.

Whenever a philosopher gets hungry, he tries to pick up 2 chopsticks that are close to him. Philosopher may pick up one chopstick at a time. He cannot pick up a chopstick that is already in the hand of neighbor. When a hungry philosopher has both chopsticks at the same time, he start eating without releasing his chopstick and starts thinking again .The problem could be raised when all the philosopher try to keep the chopstick at the same time.

This may lead to deadlock situations. To synchronize all philosophers, semaphore chopsticks [5] are used as a variable where all the elements are first initialized to 1.The structure of philosophers is shown below;

SEMAPHORE CHOPSTICK [5]

Think: After eating

Eat: Hungry

Do

{

wait(chopstick[i]);

wait(chopstick[(i+1)%5]);

---

eat

signal(chopstick[i]);

signal(chopstick[(i+1)%5]);

---

think

}while(1);

1) The following three functions lock and unlock a mutex:

```
# include<pthread.h>
intpthread_mutex_lock(pthread_mutex_t * mptr);
intpthread_mutex_trylock(pthread_mutex_t * mptr);
intpthread_mutex_unlock(pthread_mutex_t * mptr);
```

2) The mutex or condition variable is initialized or destroyed with the following functions.

```
# include<pthread.h>
intpthread_mutex_init(pthread_mutex_t * mptr,constpthread_mutexattr_t * attr);
intpthread_mutex_destroy(pthread_mutex_t * mptr);
```

Functions used with their syntax :
Routines:

```
pthread_create (thread,attr,start_routine,arg)

pthread_exit (status)
```

Creating Threads:

Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.

`pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

`pthread_create` arguments:

a. `thread`: An opaque, unique identifier for the new thread returned by the subroutine.
b. `attr`: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
c. `start_routine`: the C routine that the thread will execute once it is created.
d. `arg`: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.

When a program is started by exec, a single thread is created , called the initial thread or main thread . Additional threads are by pthread_create.

include<pthraed.h>

intpthread_create(pthread_t * tid,constpthread_attr_t * attr,void * (* func)void *),void* arg);

Thread Joining:

- "Joining" is one way to accomplish synchronization between threads.
- The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates.
- The programmer is able to obtain the target thread's termination return `status` if it was specified in the target thread's call to `pthread_exit()`.
- A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.

CONCLUSION:

Thus, we have implemented dining philosopher's problem using 'C' in Linux.

FAQ:

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

ASSIGNMENT NO: 7(a)

AIM: Inter process communication in Linux using pipes.

OBJECTIVES:

Implementation of Full duplex communication between parent and child processes. Parent process writes a pathname of a file (the contents of the file are desired) on one pipe to be read by child process and child process writes the contents of the file on second pipe to be read by parent process and displays on standard output.

THEORY:

PIPES:

Pipe create an inter-process channel.

Header file required for pipe:

```
#include<unistd.h>
pipe(int fildes[2])
```

The pipe() function shall create a pipe and place two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the pipe() call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The fcntl() function can be used to set both these flags.)

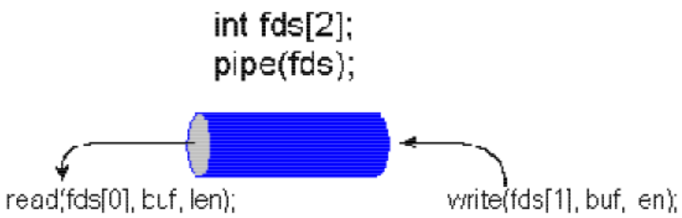
Data can be written to the file descriptor fildes[1] and read from the file descriptor fildes[0]. A read on the file descriptor fildes[0] shall access data written to the file descriptor fildes[1] on a first-in-first-out basis.

A UNIX pipe provides a half duplex communication.

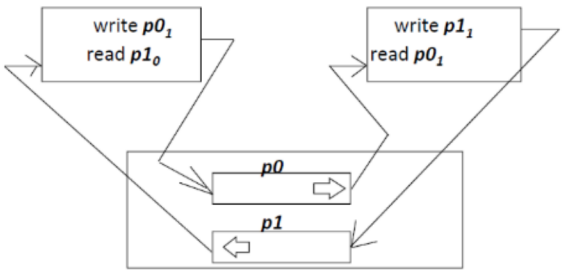
- A pipe can be explicitly created in Unix using the pipe system call. Two file descriptors are returned--fildes[0] and fildes[1], and they are both open for reading and writing. A read from fildes[0] accesses the data written to fildes[1] on a first-in-first-out (FIFO) basis and a read from fildes[1] accesses the data written to fildes[0] also on a FIFO basis.
- When a pipe is used in a Unix command line, the first process is assumed to be writing to stdout and the second is assumed to be reading from stdin. So, it is common practice to assign the pipe write device descriptor to stdout in the first

process and assign the pipe read device descriptor to stdin in the second process. This is elaborated below in the discussion of multiple command pipelines.

The pipe(2) system call returns two file descriptors that form a "pipe", a one-way communication channel with a "read end" and a "write end".



A UNIX pipe provides a Full duplex communication via two pipes.
If we want two way communication generally we create two pipes one for each direction. Create two separate pipes, say p0 and p1.



EXAMPLE:

Using a Pipe to Pass Data between a Parent Process and a Child Process: The following example demonstrates the use of a pipe to transfer data between a parent process and a child process. Error handling is excluded, but otherwise this code demonstrates good practice when using pipes: after the fork() the two processes close the unused ends of the pipe before they commence transferring data.

```
#include <stdlib.h>
#include <unistd.h>
...
intfildes[2];
constint BSIZE = 100;
charbuf[BSIZE];
ssize_tnbytes;
int status;
```

```
status = pipe(fildes);
if (status == -1 ) {
                                        /* an error occurred */
   ...
}
switch (fork()) {
case -1:                                /* Handle error */
break;
case 0:                                 /* Child - reads from pipe */
close(fildes[1]);                       /* Write end is unused */
nbytes = read(fildes[0], buf, BSIZE);   /* Get data from pipe */
                                        /* At this point, a further read would see end of file
... */
close(fildes[0]);                       /* Finished with pipe */
exit(EXIT_SUCCESS);


default:                                /* Parent - writes to pipe */
close(fildes[0]);                       /* Read end is unused */
write(fildes[1], "Hello world\n", 12);  /* Write data on pipe */
close(fildes[1]);                       /* Child will see EOF */
exit(EXIT_SUCCESS);
}
```

CONCLUSION:

Thus, we studied inter process communication using pipes.

ASSIGNMENT NO: 7 (b)

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: 8

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:

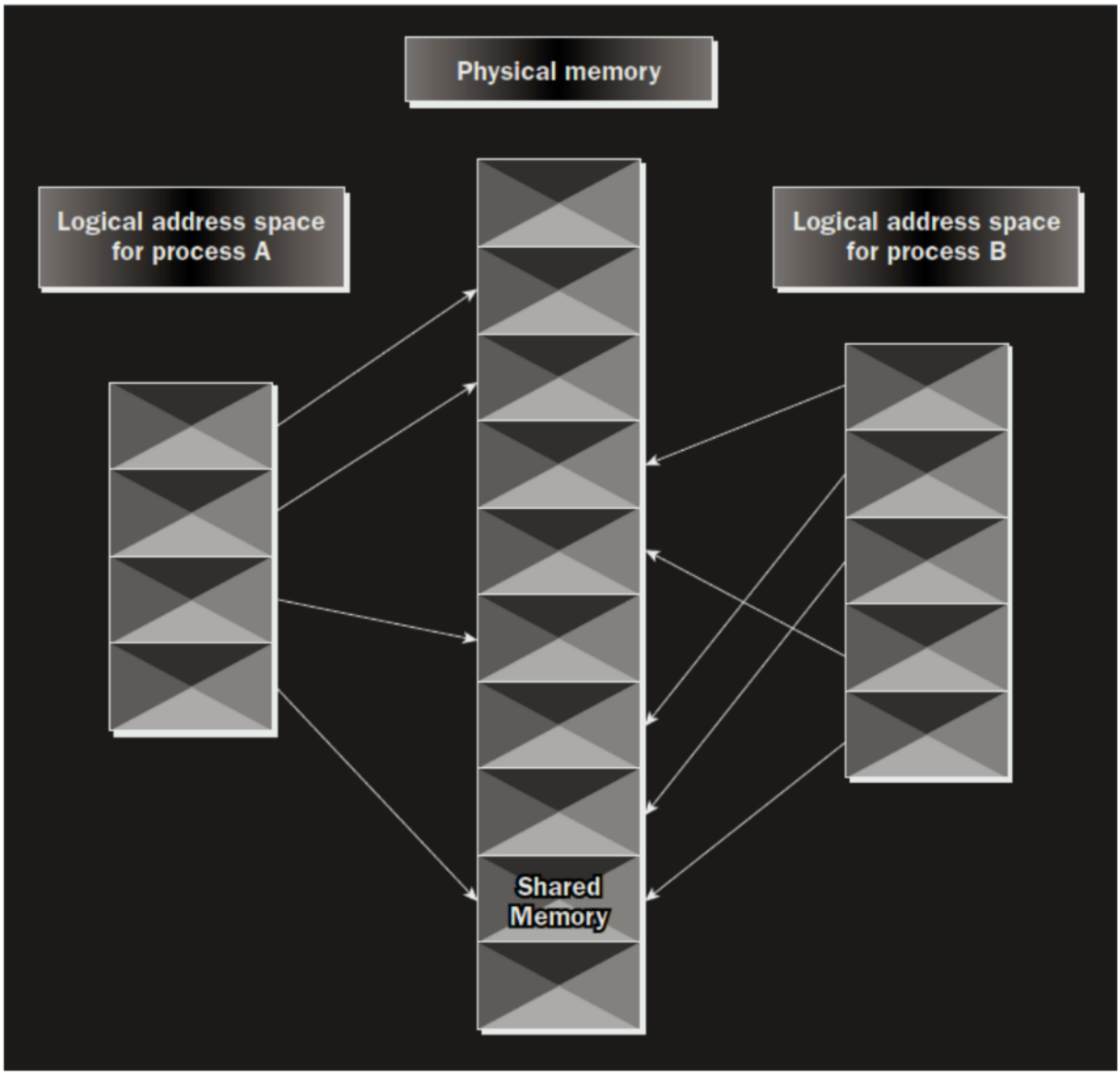
Shared Memory

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 synchronizeaccess 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.



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: 9

AIM

File Handling System Calls: Implement an assignment using File Handling System Calls (Low level system calls like open, read, write, etc).

OBJECTIVE

Write a C program to update the details of employee using C low level library functions for file handling.

C files I/O functions handles data on secondary storage device, such as a hard disk. C can handle files as **Stream-oriented data (Text) files** and **System oriented data (Binary)** files

Stream oriented data files	The data is stored in same manner as it appears on the screen. The I/O operations like buffering, data conversions etc. take place automatically.
System oriented data files	System-oriented data files are more closely associated with the OS and data stored in memory without converting into text format.

C File Operations

There are five major operations that can be performed on a file are:

- Creation of a new file.
- Opening an existing file.
- Reading data from a file.
- Writing data in a file.
- Closing a file.

Steps for Processing a File

- Declare a file pointer variable.
- Open a file using fopen() function.

- Process the file using suitable function.
- Close the file using fclose() function.

Example: **This C program is used to copy a file.**

Firstly you will specify the file to copy and then you will enter the name of target file, You will have to mention the extension of file also. We will open the file that we wish to copy in read mode and target file in write mode.

```
#include<stdio.h>

#include<stdlib.h>

int main() {char ch, source_file[25], target_file[25];

FILE *source, *target;

printf("Enter name of file to copy\n");

gets(source_file);

source = fopen(source_file, "r");

if( source == NULL )

{ printf("Press any key to exit...\n");

exit(EXIT_FAILURE); }

printf("Enter name of target file\n"); gets(target_file);

target = fopen(target_file, "w");

if( target == NULL ) { fclose(source);

printf("Press any key to exit...\n"); exit(EXIT_FAILURE);

} while( ( ch = fgetc(source) ) != EOF ) fputc(ch, target); printf("File copied successfully.\n");
fclose(source);

fclose(target);

return 0; }
```

CONCLUSION:

Thus we have implemented file handling system calls.

ASSIGNMENT No: 10

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

Simple Example
Say, we wanted to add our own version of the system call getpid(). Let's call our version mygetpid(). The implementation of mygetpid() is:

```
asmlinkage long sys_getpid(void)
{
return current->tgid;
}
```

NOTE: asmlinkage must appear before every system call.

It tells the compiler to only look on the stack for the functions arguments (aka compiler magic).
Here are concise steps you need to follow to add this system call:

1. Implement the function call and put it in the appropriate file. Since getpid() is defined in kernel/timer.c we'll put the above implementation of mygetpid() in the same file.
2. Add an entry to the end of the system call table.
   (a) vi arch/i386/kernel/syscall_table.S
      i. For 2.6.24, change to "arch/x86/kernel/syscall_table.S
   (b) Add the line ".long sys_mygetpid" after the line ".long sys_inotify_rm_watch" (the last one in the list of system calls).
      i. For 2.6.17, add after ".long sys_vmsplice".
      ii. For 2.6.24, just go to then end of the list.
3. Define the system call number in include/asm/unistd.h
   (a) vi include/asm-i386/unistd.h
      i. For 2.6.24, change to "include/asm-x86/unistd_32.h"
   (b) Add the line "#define __NR_mygetpid 294" after the line "#define__NR_inotify_rm_watch293"
      i. For 2.6.17, add after "#define __NR_vmsplice 316".
      ii. For 2.6.24, change "294" to "325".

(c) Change the line "#define NR_syscalls 294" to be "#define NR_syscalls 295"
i. For 2.6.17, change to "#define __NR_syscalls 317"
ii. For 2.6.24, change to "#define __NR_syscalls 326"
4. Recompile your kernel and boot to it.

Accessing the System Call from User-Space

To test this new system call use the following code:
Compile Command:
gcc -I/The/Location/of/your/linux/include testSysCall.c

Note: Change the above to reflect the location of your kernel source code.

Code:

```
#include<stdio.h>
#include<linux/unistd.h>          /* Defines _syscall0 and has mygetpid syscall number
*/
#include<errno.h>                 /* need this for syscall0 macro too */
_syscall0(long, mygetpid)          /*Can call "mygetpid" now */
                                  /* If you are using 2.6.24, delete the above line */
int main()
{
printf("Process ID: %d\n", mygetpid());
                        /* For 2.6.24, change to "printf("Process ID: %d\n", syscall(325)); */
return 0;
}
```

Tutorial for adding a System Call to the 2.4 kernel
The tutorial below provides a good explanation of adding a system call to the linux 2.4 kernel. While the steps in the tutorial are slightly different from the ones given above, I still highly recommend that you read it.

"How To Add a System Call to Linux on an i386" http://www.superfrink.net/docs/sys_call_howto.html

Adding the swipe() System Call
You are going to add a system call, swipe(), that steals the collective timeslices for a specified set of processes and adds it to the target process' timeslice. The swipe() system call takes a target process id, an integer that provides the type of the set of processes, and an integer that provides the process set. The system call returns the total amount of timeslice swiped or a negative number if an error occured. Therefore, the prototype of the function is:

```
int swipe(pid_t target, int which, int who)
```

The parameter target provides a process id for the process the will receive the extra timeslice. If this process id is invalid, then swipe() should return -EINVAL. The which parameter can take the value 0, 1, or 2 to indicate that the parameter who specifies a process id, a process group id, or a user id, respectively. If who is a process id, then the process set is that process plus all of its decendants. If who is a process group id, then the process set is all processes in that group. If who is a user id, then the process set is all processes owned by that user.
Look at the use of the functions sys_kill(), sys_setpriority(), and scheduler_tick() for clues on how to implement swipe().

Using swipe()
Now, you're going to explore two uses of swipe(). Use swipe() to:
1. Create a process that monopolizes the CPU.
2. Create a wrapper program that takes a simple command as an argument and arguments to specify a set of processes and executes the simple command starting with the collective timeslices of all processes in the set. Here is an example of the wrapper (called swipe_it):

> swipe_it 0 3500 ls -l

In the above command, the wrapper (swipe_it) will steal all the current timeslices from the process 3500 and all of its children and give it to the simple command "ls -l".

ASSIGNMENT No: 9

AIM : Implementing a file system in a Linux OS.

OBJECTIVE: To implement a C program for different page replacement algorithms.

THEORY:

1. Virtual File System

The Linux kernel implements the concept of Virtual File System (VFS, originally Virtual Filesystem Switch), so that it is (to a large degree) possible to separate actual "low-level" filesystem code from the rest of the kernel. This API was designed with things closely related to the ext2 filesystem in mind. For very different filesystems, like NFS, there are all kinds of problems.

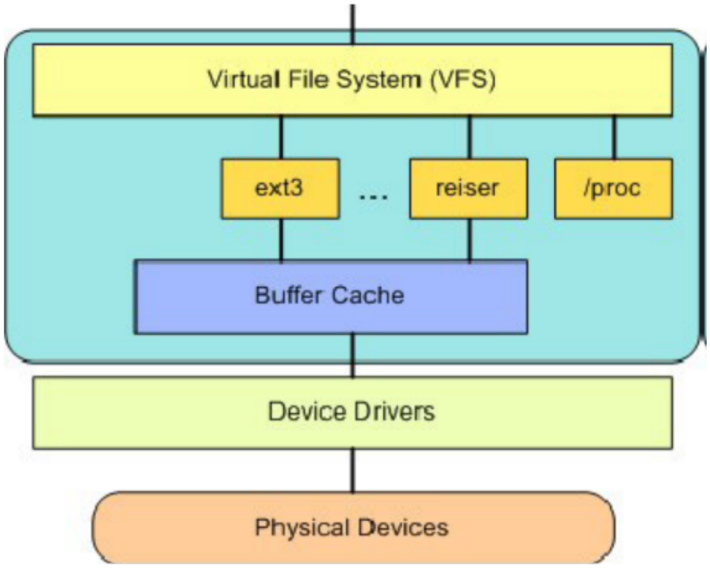
Virtual File System- Main Objects
The kernel keeps track of files using in-core inodes ("index nodes"), usually derived by the low-level filesystem from on-disk inodes. A file may have several names, and there is a layer of dentries ("directory entries") that represent pathnames, speeding up the lookup operation. Several processes may have the same file open for reading or writing, and file structures contain the required information such as the current file position. Access to a filesystem starts by mounting it. This operation takes a filesystem type (like ext2, vfat, iso9660, nfs) and a device and produces the in-core superblock that contains the information required for operations on the filesystem; a third ingredient, the mount point, specifies what pathname refers to the root of the filesystem.

The /proc filesystem
The /proc filesystem contains a illusionary filesystem. It does not exist on a disk. Instead, the kernel creates it in memory. It is used to provide information about the system (originally about processes, hence the name). The proc filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mountedat /proc. Most of it is read-only, but some files allow kernel variables to be changed. The /proc filesystem is described in more detail in the proc manual page.

- **/proc/1**:A directory with information about process number 1. Each process has a directory below /proc with the name being its process identification number.
- **/proc/cpuinfo**: Information about the processor, such as its type, make, model, and performance.
- **/proc/devices**: List of device drivers configured into the currently running kernel.
- **/proc/filesystems**: Filesystems configured into the kernel.

- **/proc/ioports**: Which I/O ports are in use at the moment.
- **/proc/meminfo**: Information about memory usage, both physical and swap.
- **/proc/version**: The kernel version.

VFS in Linux



Create filesystem as a module

- Write a hello_proc.c program.
- Create a Makefile.
- The program and Makefile should be kept in a single folder.
- Change directory to this folder and execute following:
  - make
  - insmod hello_proc.ko
  - dmesg (see the kernel buffer contents, reads the kernel log file /var/log/syslog)
  - lsmod
  - rmmod hello_proc.ko

Makefile

```
obj-m += hello_proc.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD)
modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD)
clean
```

hello_proc.c

```
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

static int hello_proc_show(struct seq_file *m, void *v) {
seq_printf(m, "Hello proc!\n");
return 0;
}

static int hello_proc_open(struct inode *inode, struct file *file) {
return single_open(file, hello_proc_show, NULL);
}

static const struct file_operations hello_proc_fops = {
.owner = THIS_MODULE,
.open = hello_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};

static int __init hello_proc_init(void)
{
proc_create("hello_proc", 0, NULL,
&hello_proc_fops);
return 0;
}

static void __exit hello_proc_exit(void)
{
remove_proc_entry("hello_proc", NULL);
}

MODULE_LICENSE("GPL");
module_init(hello_proc_init);
module_exit(hello_proc_exit);
```

ls -l /proc

- **cat /proc/hello_proc**

- proc_create: It creates a virtual file in the /proc directory.
- remove_proc_entry: It removes a virtual file from the /proc directory.
- hello_proc_show(): It shows the output.
- seq_printf : It uses sequential operations on the file.
- hello_proc_open() : This is the open callback, called when the proc file is opened.
- single_open(): All the data is output at once.

Limitations to /proc file system

- Our module cannot output more than one page of data to the pseudo-file at once.
- A page is a pre-defined amount of memory, typically 4096 bytes (4K defined by processor), and is available in the PAGE_SIZE macro.
- This limitation is bypassed by using sequence files.