OPERATING SYSTEMS
LAB MANUAL

Subject Code: CS
Class: IV Semester(CSE)

Prepared By Mr. Biswajit Sarma
Assistant Professor

Department of Computer Science & Engineering
JORHAT ENGINEERING COLLEGE
JORHAT : 785007, ASSAM

Vision of the Department

To become a prominent department of Computer Science and Engineering for producing quality human resources to meet the needs of the industry and society

Mission of the Department

1: To impart quality education through well-designed curriculum and academic facilities to meet the computing needs of the industry and society
2: To inculcate the spirit of creativity, team work, innovation, entrepreneurship and professional ethics among the students
3: To facilitate effective interactions to foster networking with alumni, industries, institutions of learning and research and other stake-holders
4: To promote research and continuous learning in the field of Computer Science and Engineering

OBJECTIVE:This lab complements the operating systems course. Students will gain practical experience with designing and implementing concepts of operating systems such as system calls, CPU scheduling, process management, Inter Process Communication, memory management, file systems and deadlock handling using C language in Linux environment

Program Outcomes

PO1	Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4	Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal,health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes

PSO1	Gain ability to employ modern computer languages, environments and platforms in creating innovative career paths
PSO2	Achieve an ability to implement, test and maintain computer based system that fulfils the desired needs

OPERATING SYSTEMS LAB SYLLABUS
(Practical Hours: 04, Credits: 02)

Implement the following programs on Linux platform using C language.

Exp. No.	Division of Experiments	List of Experiments	Page No.
1.	Make file, Header file, Command line arguments	Write a C program which takes all the inputs from the command line. Variables and prototypes are declared in a header file for a small calculator which can add, subtract, multiply, divide two numbers. Program should be executed using a Makefile.	6-11
2.	CPU Scheduling Algorithms	Write program(s) in C language to achieve the following requirements 1. Use the FIFO Scheduling Algorithm 2. Use the SJF Scheduling Algorithm 3. Use the SRTN Scheduling Algorithm 4. Use the Round Robin Scheduling Algorithm (with time slice 3)	12-13
3.	Process Creation	Write a C program to perform the following: Declare two global variables a=5, b=7; Create a Child process using the fork system call, the child process displays the id of itself and id of its parent and add those variables and store the result in b variable. After this the parent runs and display the id of itself and id of its parent and display the value of variable b.	14
4.	Memory Management Techniques	Implementation of Memory Allocation and Paging.	15
5.	Inter Process Communication (IPC) PIPES	Write C program to perform the IPC using pipes, the sender sends an array of 10 random integer, upon receiving those integers the receiver finds the highest number from them and return the highest number to the sender and sender display it.	16

6.	Inter Process Communication (IPC) MESSAGE QUEUE	Write C program to perform the IPC using message queue, the sender sends an array of strings, upon receiving the string the receiver count the number of vowels and return to the sender, the sender displays that numbers in the screens.	17-18
7.	Inter Process Communication (IPC) SHARED MEMORY	Write C program to perform the IPC using shared memory, the sender sends an array of strings, upon receiving the string the receiver converts the string to uppercase and return it to the sender, the sender displays that string in the screens.	19-20
8.	Thread VS Process	Write a C program which Create 3 threads, the first thread displays Hello, Second thread displays CSE, third thread displays JEC.	21-27
9.	Deadlock and Mutual Exclusion	write a C program to perform the following, Create 2 threads, the second thread run first and fill up an array of 10 random numbers(which is a global array), then first thread runs and finds the average number from them and displays in the screen. *** This should not produce and Deadlock***	28-30
10.	Shell Script.	Write shell Script a)Unix Shell programming commands b)Concatenation of two strings c)Comparison of two strings d)Maximum of three numbers e)Fibonacci series f)Arithmetic operation using case	31-33

Experiment No:1

AIM: To study how to create user define header file, Make file and how to take inputs from the command line.

h extension are called header files in C. Header files are simply files in which you can declare your own functions that you can use in your main program or these can be used while writing large C programs. NOTE:Header files generally contain definitions of data types, function prototypes and C preprocessor commands.

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Makefile is a tool to simplify or to organize code for compilation. Makefile is a set of commands (similar to terminal commands) with variable names and targets to create object file and to remove them. In a single make file we can create multiple targets to compile and to remove object, binary files.

Step1:
For header file :
Create a file with the extension .h (example abc.h)
start the file with following way :
#ifndef ABC_H
#define ABC_H
/ declare your variables and prototypes here*/*
#endif

step2:
Create 4 different files
add.c, sub.c, mul.c, div.c
implement addition, subtraction, multiplication, division functions on those files according to the prototypes given in the header file. ALL files Must include header file in the following way:

#include”abc.h”

step3:
declare the main function in the following way in a file main.c :

```
int main(int argc,char *argv[])
{
/* write your code here which call all the functions implemented in other files*/
}
```

the first argument argc is of type integer, which count the number of arguments in the command line. The next argument argv is character pointer array, which points to each command line argument .
For example:
If your program requires 2 integer variables and you need to send it by command line arguments

```
int main(int argc,char *argv[])
{
        int a,int b;
        a=atoi(argv[1]);//atoi character to integer conversion
        b=atoi(argv[2]);//atoi character to integer conversion
}
```

run the program:
$./a.out 35 45

here argc=3

argv[0]--------→./a.out
argv[1]--------→ 35
argv[2]--------→ 45

Step4:
After finishing all the files (main.c, add.c, sub.c, mul.c, div.c)
Normally, you would compile this collection of code by executing the following command:

```
gcc -o hellomake main.c add.c sub.c mul.c div.c -I
```

This compiles the five .c files and names the executable hellomake. The `-I.` is included so that gcc will look in the current directory (.) for the include file abc.h. Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

Makefile 1

```
hellomake: main.c add.c sub.c mul.c div.c
     gcc -o hellomake main.c add.c sub.c mul.c div.c -I.
```

If you put this rule into a file called `Makefile` or `makefile` and then type `make` on the command line it will execute the compile command as you have written it in the makefile. Note that `make` with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the `:`, make knows that the rule `hellomake` needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the gcc command in the makefile. There must be a tab at the beginning of any command, and `make` will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

Makefile 2

```
CC=gcc
CFLAGS=-I.

hellomake: main.c add.c sub.c mul.c div.c
     $(CC) -o hellomake main.c add.c sub.c mul.c div.c -I.
```

So now we've defined some constants `CC` and `CFLAGS`. It turns out these are special constants that communicate to `make` how we want to compile the files main.c add.c sub.c mul.c div.c. In particular, the macro `CC` is the C compiler to use, and `CFLAGS` is the list of flags to pass to the compilation command. By putting the object files--main.o add.o sub.o mul.o div.o--in the dependency list and in the rule, `make` knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to abc.h, for example, `make` would not recompile the .c files, even though they needed to be. In order to fix this, we need to tell `make` that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

Makefile 3

```
CC=gcc
CFLAGS=-I.
DEPS = abc.h

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: main.o add.o sub.o mul.o div.o
     $(CC) -o hellomake main.o add.o sub.o mul.o div.o -I.
```

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the DEPS macro. The rule then says that to generate the .o file, `make` needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the `-o $@` says to put the output of the compilation in the file named on the left side of the `:`, the `$<` is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros `$@` and `$^`, which are the left and right sides of the `:`, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

Makefile 4

```
CC=gcc
CFLAGS=-I.
DEPS = abc.h
OBJ = main.o add.o sub.o mul.o div.o

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS)
```

So what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place? The answer, of course, is yes. The following makefile defines paths to the include and lib directories, and places the object files in an obj subdirectory within the src directory. It also has a macro defined for any libraries you want to include, such as the math library `-lm`. This makefile should be located in the src directory. Note that it also includes a rule for cleaning up your source and object directories if you type `make clean`. The .PHONY rule keeps `make` from doing something with a file named clean.

Makefile 5

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = abc.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = main.o add.o sub.o mul.o div.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS) $(LIBS)
```

```
.PHONY: clean

clean:
        rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

So now you have a perfectly good makefile that you can modify to manage small and medium-sized software projects. You can add multiple rules to a makefile; you can even create rules that call other rules. For more information on makefiles and the `make` function, check out the GNU Make Manual, which will tell you more than you ever wanted to know (really).

Experiment No:2
AIM: To understand the CPU scheduling algorithms

The program Will accept an integer (Say n) as input that is the number of Process. The Length of the CPU bursts lies between 2 and 10, and are generated **randomly**. Initially the Arrival time of the first CPU burst of every process will be generated **randomly between 2 and 6**. For the System Time Use a integer counter with a loop structure, Increase the Counter when this loop is executed Once. For this set of processes do the scheduling using the specified Scheduling Algorithm. The Out put of the program will be the Status of the Ready Queue and the process that is scheduled at this moment. When All the CPU Burst Of All The process is over Display the average Waiting Times and Average Turnaround Time of the processes.

For Random number use Rand function. Please use linux manual pages for rand function.
$ man rand

FCFS CPU SCHEDULING ALGORITHM For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

SJF CPU SCHEDULING ALGORITHM For SJF scheduling algorithm, read the number of processes/jobs in the system,their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time,and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

SRTN CPU SCHEDULING ALGORITHM for SRTN Shortest remaining time, also known as shortest remaining time first, is a scheduling method that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.

ROUND ROBIN CPU SCHEDULING ALGORITHM For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time

slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

Sample output:

PROCESS	ARRIVAL TIME	BURST TIME
P0	1	5
P1	2	8
P2	4	3
P3	5	6

FCFS:
Average Waiting Time—xxxx Average Turnaround Time—yyyy

SJF:
Average Waiting Time—xxxx Average Turnaround Time—yyyy

SRTN:
Average Waiting Time—xxxx Average Turnaround Time—yyyy

RR:
Average Waiting Time—xxxx Average Turnaround Time—yyyy

Experiment No:3
AIM: To understand the basic linux system calls like fork, getpid, getppid, wait, sleep etc.

ALGORITHM:
STEP 1: Start the program.
STEP 2: Declare pid as integer.
STEP 3: Create the process using Fork system call.
STEP 4: Check pid is equal to 0 then child process else parent process .
STEP 5: Use wait or sleep system call in parent process to keep on waiting the parent process until child finish its operation.
STEP 6: To get the id of a process use getpid and to get the parent id of a process use getppid system call.
STEP 7. Stop.

Use linux manual pages for all the system call

$man fork
$man getpid

Experiment No:4

AIM: To understand the Memory Allocation and Paging.

Assume that the System have a memory of 10,000. Design a menu to access different funtionalities like Creation of process, Termination of a Process, Translation of address. Viewing the Memory information of all Processes. Within the Creation, it will ask for the Process size(or Number of segment and their sizes.) and Update the PDT(or Segment/page table) and PCB of the Process accordingly, if possible. Assign a Unique ID for the process. Within Termination, Display the Process numbers and their sizes. User will select one of them to be deleted at a time. Deletion will remove the entry of the process from the data structures and PCB of it will be destroyed. Within Translation, user will select a process first by giving the ID. after that the program will accept a logical address from the user, and corresponding Physical address will be calculated if it is a valid and will be displayed along with the memory related information for that process. if the user enters a -ve logical address he will go back to the menu. Within the Viewing The complete PDT (or all Segment/page tables) will be displayed in a systematic way.

1.Use Equal size static Partitioning technique to allocate memory, and there are 5 partitions. process Size will be in the range 500 to 2500. Amount of internal fragmentation should also be produced as one output.

Or

2.Use Variable size static Partitioning technique to allocate memory, with five partitions smallest one being of size 500 and the largest being 2500. process Size will be in the range 500 to 2600.

or

3.Use Dynamic Partitioning Partitioning, with maximum 10 partitions. and process size being in the range 500 to 2600. don not create a hole with size less than the smallest process.

Or

4.Use Simple Segmentation,number of segments between 3 to 5. Segment sizes lies between 200 to 1000.

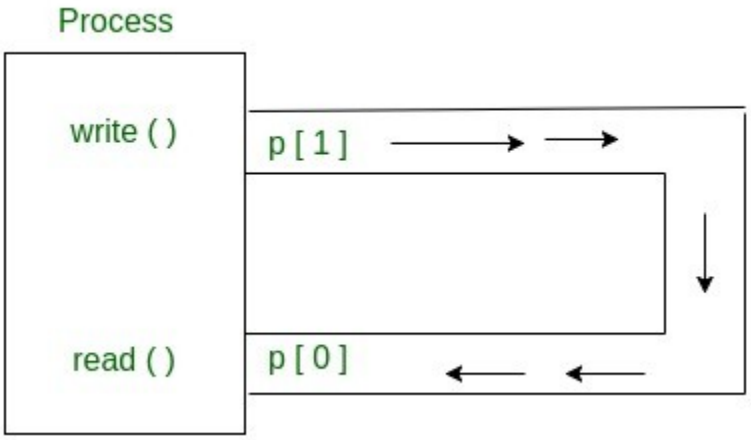
or

5.Use Simple Paging , use a page size of 128 bytes and the process size lie between 500 to 2600 bytes.

Experiment No:5

AIM: To understand the linux Inter process communication system using PIPES system call. Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).

- Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a *"virtual file"*.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.



You may require the following system calls

1. Pipe
2. write
3. read
4. mkfifo.

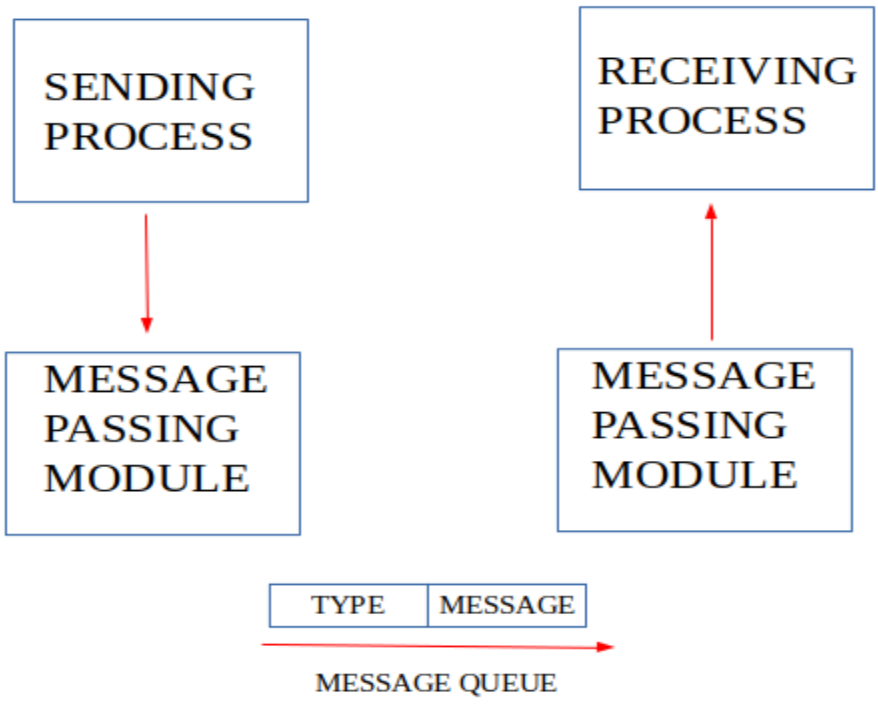
Use linux manual pages for the above mention system calls

Experiment No:6

AIM: To understand the linux Inter process communication system using message queue system call.

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget().
New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.



System calls used for message queues:

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

msgget(): either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.

msgsnd(): Data is placed on to a message queue by calling msgsnd().

msgrcv(): messages are retrieved from a queue.

msgctl(): It performs various operations on a queue. Generally it is use to destroy message queue.

Use linux manual pages for the above mention system calls.

Experiment No:7

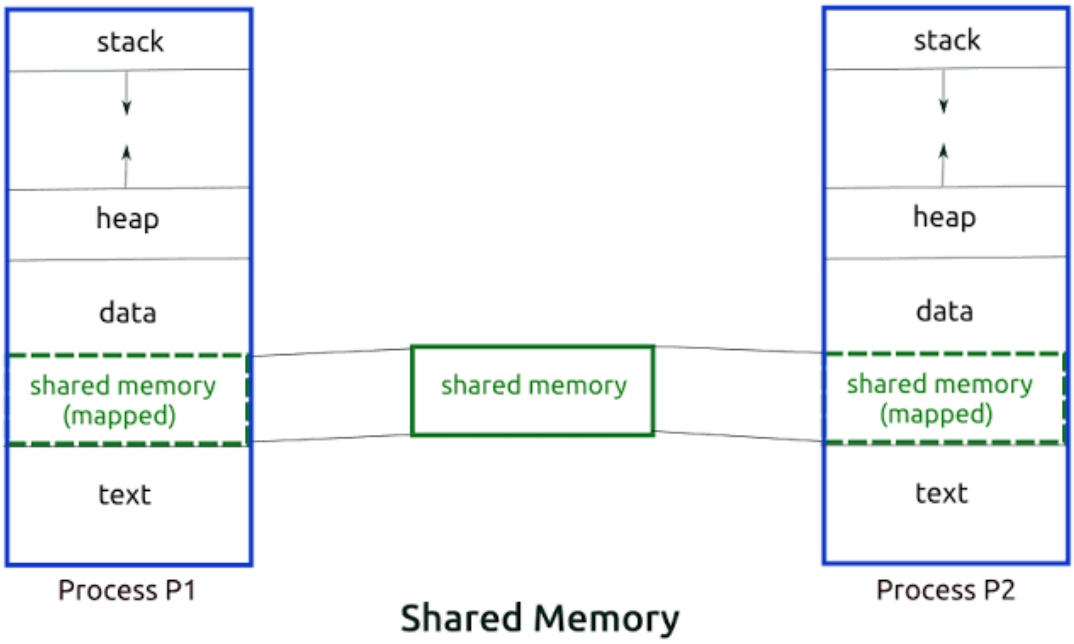
AIM: To understand the linux Inter process communication system using shared memory system call.

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

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

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

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



SYSTEM CALLS USED ARE:

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

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

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

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

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

Use linux manual pages for the above mention system calls.

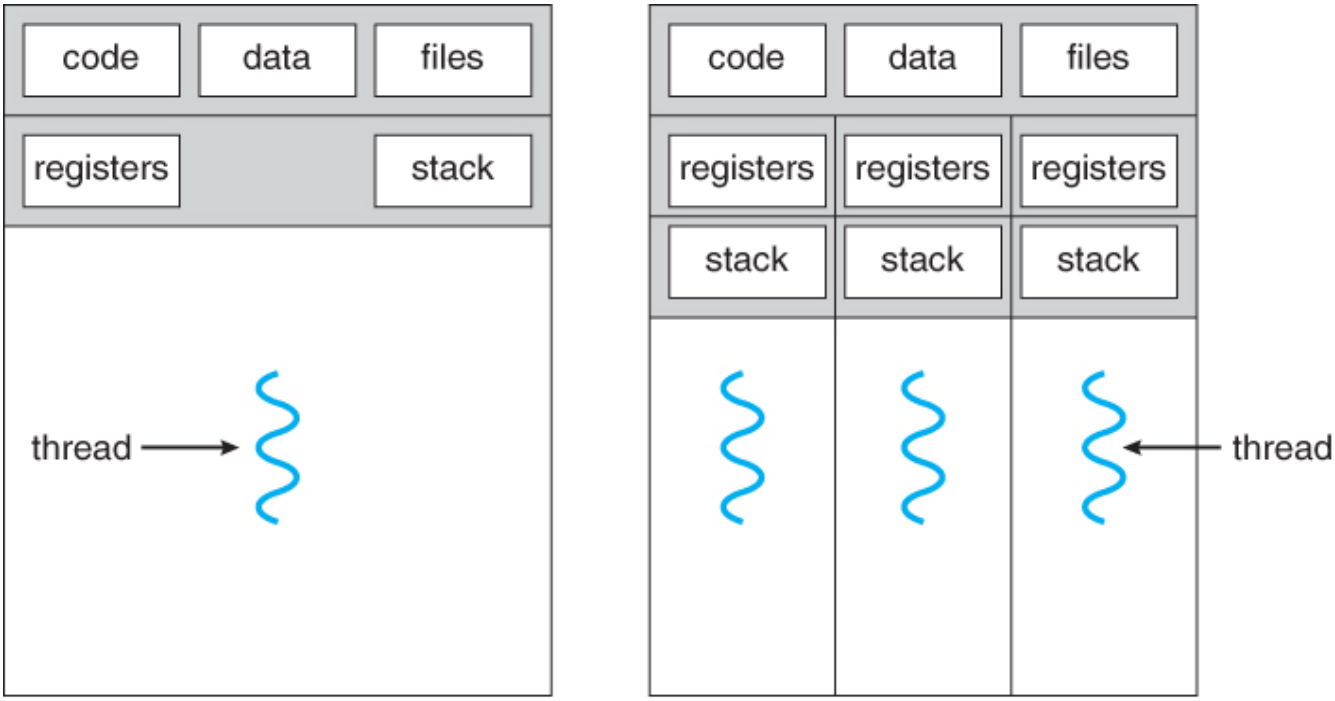
Experiment No:8

AIM: To understand what is the difference between thread and process and also how to implement thread using posix library.

Difference between Process and Thread

Process:

Process means any program is in execution. Process control block controls the operation of any process. Process control block contains the information about processes for example: Process priority, process id, process state, CPU, register etc. A process can creates other processes which are known as **Child Processes**. Process takes more time to terminate and it is isolated means it does not share memory with any other process.



single-threaded process multithreaded process

Thread:

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. A thread have 3 states: running, ready, and blocked.Thread takes less time to terminate as compared to process and like process threads do not isolate.

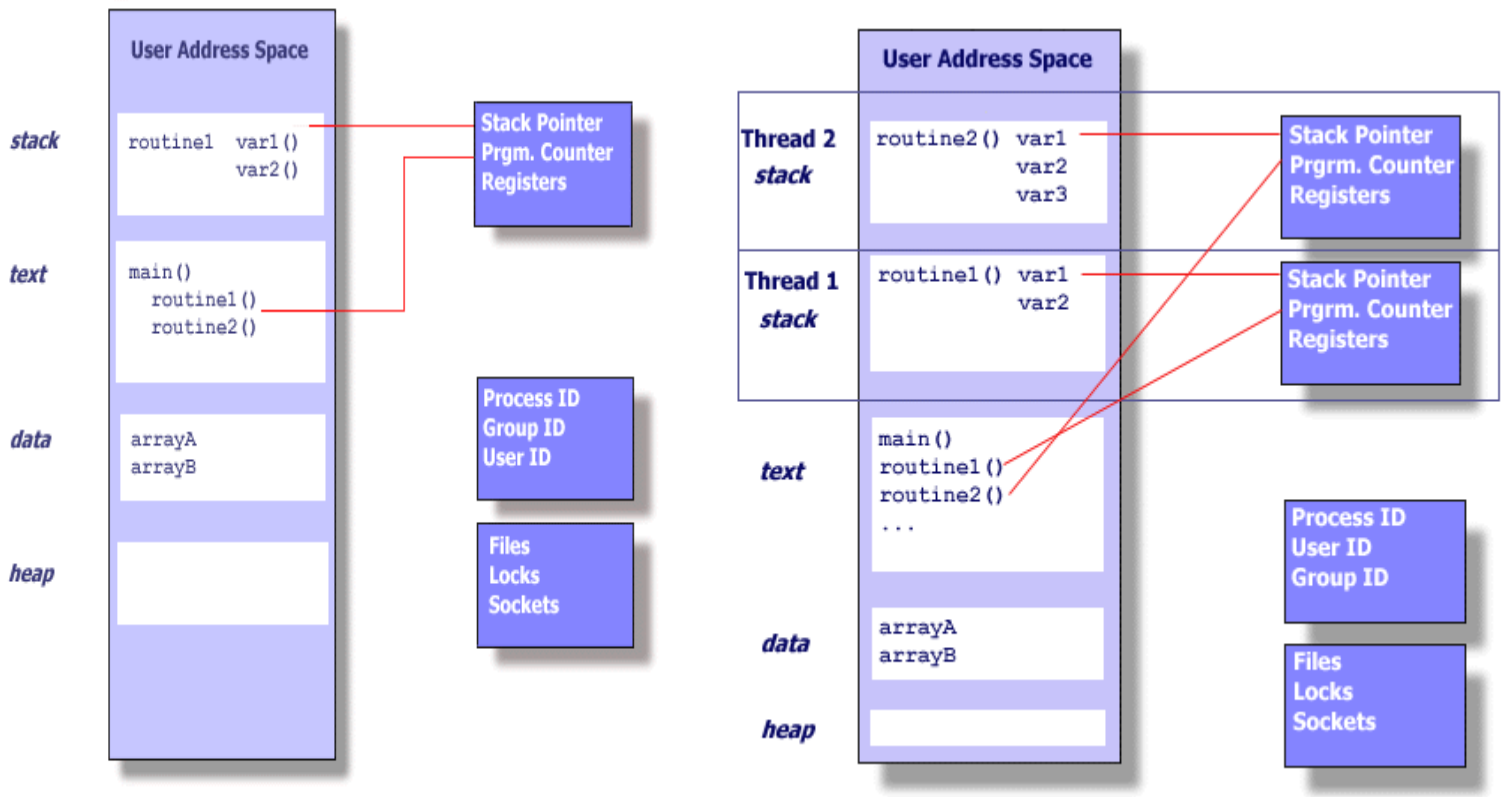
Difference between Process and Thread:

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means segment of a process.
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6.	Process consume more resources.	Thread consume less resources.
7.	Process is isolated.	Threads share memory.

What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.
- How is this accomplished?
- Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:
    - Process ID, process group ID, user ID, and group ID
    - Environment
    - Working directory.
    - Program instructions
    - Registers
    - Stack
    - Heap
    - File descriptors

- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).



Thread Basics:

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
    - Process instructions
    - Most data
    - open files (descriptors)
    - signals and signal handlers
    - current working directory
    - User and group id
- Each thread has a unique:
    - Thread ID
    - set of registers, stack pointer
    - stack for local variables, return addresses
    - signal mask
    - priority
    - Return value: `errno`
- pthread functions return "0" if OK.

Thread Creation and Termination:

Example: `pthread1.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;
    /* Create independent threads each of which will execute function */
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
     /* Wait till threads are complete before main continues. Unless we  */
```

```
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Compile:

- C compiler: `gcc -lpthread pthread1.c`
  or
- C++ compiler: `g++ -lpthread pthread1.c`

Run: `./a.out`

Results:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Details:

- In this example the same function is used in each thread. The arguments are different. The functions need not be the same.
- Threads terminate by explicitly calling `pthread_exit`, by letting the function return, or by a call to the function `exit` which will terminate the process including any threads.
- Function call: **`pthread_create`**

```
    int pthread_create(pthread_t * thread,
                       const pthread_attr_t * attr,
                       void * (*start_routine)(void *),
```

```
                        void *arg);
```

Arguments:

- `thread` - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- `attr` - Set to NULL if default thread attributes are used. (else define members of the struct `pthread_attr_t` defined in bits/pthreadtypes.h) Attributes include:
  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter
  - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
  - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
  - guard size
  - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
  - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
- `void * (*start_routine)` - pointer to the function to be threaded. Function has a single argument: pointer to void.
- `*arg` - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

- Function call: **`pthread_exit`**

```
void pthread_exit(void *retval);
```

Arguments:

- `retval` - Return value of thread.

This routine kills the thread. The `pthread_exit` function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.
Note: the return pointer `*retval`, must not be of local scope otherwise it would cease to exist once the thread terminates.

- [C++ pitfalls]: The above sample program **will** compile with the GNU C **and** C++ compiler g++. The following function pointer representation below will work for C but not C++. Note the subtle differences and avoid the pitfall below:

```
    void print_message_function( void *ptr );
    ...
    ...
    iret1 = pthread_create( &thread1, NULL, (void*)&print_message_function,
(void*) message1);
    ...
    ...
```

Experiment No:9

AIM: To understand what is thread synchronization, deadlock and mutual exclusion.

Thread Synchronization:

The threads library provides three synchronization mechanisms:

mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

joins - Make a thread wait till others are complete (terminated).

condition variables - data type `pthread_cond_t`

Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Example threaded function:

Without Mutex	With Mutex

`int counter=0;`
 	
`/* Function C */`
`void functionC()`
`{`
 	
`  counter++`
 	
`}`
	`/* Note scope of variable and mutex are the same */`
	`pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;`
	`int counter=0;`
	`/* Function C */`
	`void functionC()`
	`{`
	`  pthread_mutex_lock( &mutex1 );`
	`  counter++`
	`  pthread_mutex_unlock( &mutex1 );`

		}	
Possible execution sequence			
Thread 1	**Thread 2**	**Thread 1**	**Thread 2**
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable `counter`
			counter = 2

If register load and store operations for the incrementing of variable counter occurs with unfortunate timing, it is theoretically possible to have each thread increment and overwrite the same variable with the same value. Another possibility is that thread two would first increment counter locking out thread one until complete and then thread one would increment it to 2.

Sequence	Thread 1	Thread 2
1	counter = 0	counter=0
2	Thread 1 locked out. Thread 2 has exclusive use of variable `counter`	counter = 1
3	counter = 2	

```
#include   <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;
main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;
   /* Create independent threads each of which will execute functionC */
   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }
```

```
   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }
   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);
   exit(0);
}
void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

Compile: gcc -lpthread mutex1.c

Run: ./a.out

Results:

```
Counter value: 1
Counter value: 2
```

When a mutex lock is attempted against a mutex which is held by another thread, the thread is blocked until the mutex is unlocked. When a thread terminates, the mutex does not unless explicitly unlocked. Nothing happens by default.

Experiment No:10

AIM:To study about the Unix Shell Programming Commands.

INTRODUCTION :Shell programming is a group of commands grouped together under single file name. After logging onto the system a prompt for input appears which is generated by a Command String Interpreter program called the shell. The shell interprets the input, takes appropriate action, and finally prompts for more input. The shell can be used either interactively -enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled. Common Shells.

C-Shell -csh: The default on teaching systems Good for interactive systems Inferior programmable features

Bourne Shell -bsh or sh -also restricted shell -bsh: Sophisticated pattern matching and file name substitution

Korn Shell: Backwards compatible with Bourne Shell Regular expression substitution emacs editing mode

Thomas C-Shell -tcsh: Based on C-Shell Additional ability to use emacs to edit the command line Word completion & spelling correction Identifying your shell.

01. SHELL KEYWORDS :echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

02. General things SHELL

The shbang line The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #!followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.

EXAMPLE#!/bin/sh

Comments: Comments are descriptive material preceded by a #sign. They are in effect until the end of a line and can be started anywhere on the line.

EXAMPLE# this text is not
interpreted by the shell

Wildcards:There are some characters that are evaluated by the shell in a special way. They are called shell meta characters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ]are used for file name expansion. The <, >, 2>, >>, and |symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

EXAMPLE
Filename expansion:
rm *; ls??; cat file[1-3];
Quotes protect metacharacter:
echo "How are you?"

03. SHELL VARIABLES :

Shell variables change during the execution of the program .The C Shell offers a command "Set" to assign a value to a variable.

For example:

% set myname= Fred
% set myname = "Fred Bloggs"
% set age=20
A $sign operator is used to recall the variable values.
For example:
% echo $mynamewill display Fred Bloggs on the screen
A @ sign can be used to assign the integer constant values.
For example:
%@myage=20
%@age1=10
%@age2=20

%@age=$age1+$age2
%echo $ageList variables
% set programming_languages= (C LISP)
% echo $programming _languagesC LISP
% set files=*.*
% set colors=(red blue green)
% echo $colors[2]
blue
% set colors=($colors yellow)/add to list
Local variables

Local variables are in scope for the current shell. When a script ends, they are no longer available; i.e., they go out of scope. Local variables are set and assigned values.

EXAMPLE variable_name=valuename="John Doe"x=5Global variablesGlobal variables are called environment variables. They are set for the currently running shell and any process spawned from that shell. They go out of scope when the script ends.EXAMPLEVARIABLE_NAME=valueexport VARIABLE_NAMEPATH=/bin:/usr/bin:.export PATH