Ex. No. 3	SHELL PROGRAMS	Date :

Q1. Find the output of the following shell scripts

```
$ vi ex31 echo Enter value
     for n read n sum=0 i=1
     while [ $i -le $n ] do
     sum=$((sum+i))
     i=$((i+2))
     done echo Sum is
     $sum
```

Output :

Q2. Write a program to check whether the file has execute permission or not. If not, add the permission.

```
$ vi ex32
```

CODE:

```
echo Enter name of the file
read name
if [ -x $name ]
then
  echo Yes $name has executive permission
 else
  echo no $name has no executive permission
fi
```

Q3. Write a shell script to list only the name of sub directories in the present working directory

```
$ vi ex33
```

CODE:

```
for i in *
do
if [ -d $i ]
then
  echo $i
 fi
 done
```

Q4. Write a program to check all the files in the present working directory for a pattern (passed through command line) and display the name of the file followed by a message stating that the pattern is available or not available.

```
$ vi ex34
```

CODE :

```
for i in *
do
if [ -f $i ]
then
 grep $1 $i > /dev/null
 if [ $i -eq 0 ]
 then
  echo $i pattern found
 else
  echo $i pattern not found
 fi
fi
done
```

Q5. time

Code :

```
hour $(date | cut -c12-13)
if [ $hour -ge 0 -a $hour -le 11 ]
then
echo Good Morning
elif [ $hour le 17 ]
then
echo Good afternoon
else
echo Good Evening
fi
```

Ex. No. 4	PROCESS CREATION	Date :

ID int getppid() ⑦ returns the parent process ID wait()
⑦ makes a process wait for other process to complete

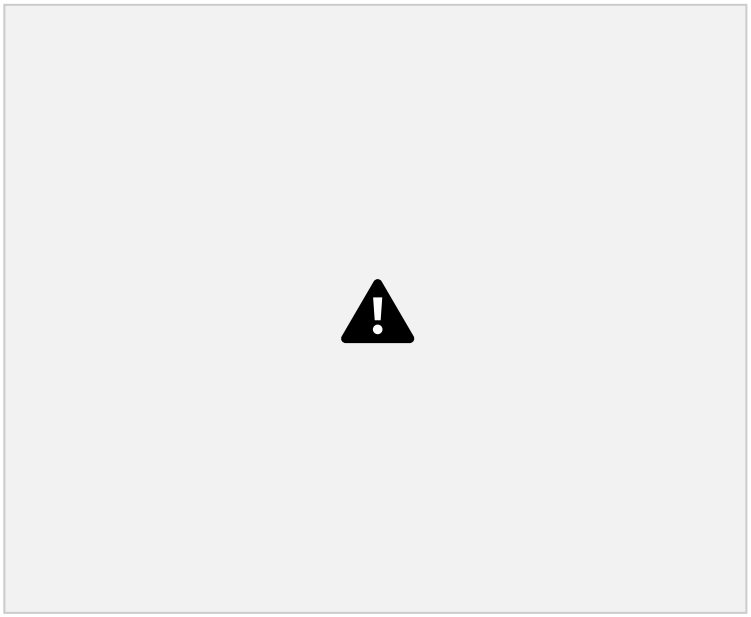
Virtual fork vfork() is similar to fork but both processes shares the same address space.

Q1. Find the output of the following program

```
#include <stdio.h>
#include<unistd.h>
int main()
{ int a=5,b=10,pid; printf("Before fork
  a=%d b=%d \n",a,b); pid=fork();

  if(pid==0)
  { a=a+1;
    b=b+1;
    printf("In child a=%d b=%d \n",a,b);
  } else {
  sleep(1); a=a
  1; b=b-1;
    printf("In Parent a=%d b=%d \n",a,b);
  } return
  0;
}
```

Output :



Q2. Rewrite the program in Q1 using vfork() and write the output #include <stdio.h>

```
#include <unistd.h>


int main() {
 int a = 5, b = 10, pid;
 printf("Before vfork a=%d b=%d \n", a, b);
 pid = vfork();
 if (pid == 0) {
 a = a + 1;
 b = b + 1;
 printf("In child a=%d b=%d \n", a, b);
 _exit(0); // Must use _exit() instead of exit() in child process.  }
else {
 sleep(1);
 a = a - 1;
 b = b - 1;
 printf("In Parent a=%d b=%d \n", a, b);
 }
 return 0;
}
```

Q3. Calculate the number of times the text "SRMIST" is printed.

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

int main()
{ fork(); fork();
   fork();
   printf("SRMIST\n");
   return 0;
}
```

Output :
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST
SRMIST

Q4. Complete the following program as described below :
The child process calculates the sum of odd numbers and the parent process calculate the sum of even numbers up to the number 'n'. Ensure the Parent process waits for the child process to finish.

CODE:

```
#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>



int main() {

 int pid, n, oddsum = 0, evensum = 0;

 printf("Enter the value of n: ");

 scanf("%d", &n);

 pid = fork();

 if (pid == 0) {

 // Child process

 for (int i = 1; i <= n; i += 2) {

 oddsum += i;

 }
```

```
printf("Sum of odd numbers: %d\n", oddsum);


} else {

// Parent process

wait(NULL); // Wait for the child process to finish

for (int i = 2; i <= n; i += 2) {

evensum += i;

}

printf("Sum of even numbers: %d\n", evensum);

}

return 0;

}
```

Sample Output :

Enter the value of n : 10
Sum of odd numbers : 25
Sum of even numbers : 30

Q5. How many child processes are created for the following code?

Hint : Check with small values of 'n'.

```
for (i=0; i<n; i++) fork();
```

Output :

2^n

Q6. Write a program to print the Child process ID and Parent process ID in both Child and Parent processes

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

int main() {
 int pid = fork(); // create a child process

 if (pid == -1) {
 // error occurred while forking
 printf("Fork failed.\n");
```

```
        return 1;
```



```
        } else if (pid == 0) {
        // child process
        printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());  }
       else {
        // parent process
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);  }

        return 0;
       }
```

Sample Output:

```
In Child Process

Parent Process ID : 18
Child Process ID Process

In Parent       : 20



Parent Process ID : 18
Child Process ID : 20
```

Q7. How many child processes are created for the following code?

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

int main()
{ fork();
    fork()&&fork()||fork();
    fork(); printf("Yes ");
    return 0;
}
```

Output :

Ex. No. 6	PIPES	Date :

Pipe is a communication medium between two or more processes. The system call for creating pipe is

```
int pipe(int p[2]);
```

This system call would create a pipe for one-way communication i.e., it creates two descriptors, first one is connected to read from the pipe and other one is connected to write into the pipe.

Descriptor p[0] is for reading and p[1] is for writing. Whatever is written into p[1] can be read from p[0].

Q1. Write the output of the following program

```
#include <stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{ int p[2]; char
  buff[25];
  pipe(p);
  if(fork()==0)
  { printf("Child : Writing to pipe
    \n"); write(p[1],"Welcome",8);
    printf("Child Exiting\n");
  }
  else
  {
    wait(NULL)
    ;
    printf("Parent : Reading from pipe \n");
    read(p[0],buff,8);
    printf("Pipe content is : %s \n",buff);
  } return
  0;
}
```

Output :

Implementing command line pipe using exec() family of functions00

Follow the steps to transfer the output of a process to pipe:

(i) Close the standard output descriptor

(ii) Use the following system calls, to take duplicate of output file descriptor of the pipe

```
int dup(int fd);
int dup2(int oldfd, int newfd);
```

(iii) Close the input file descriptor of the pipe

(iv) Now execute the process

Follow the steps to get the input from the pipe for a process:

(i) Close the standard input descriptor

(ii) Take the duplicate of input file descriptor of the pipe using dup() system call

(iii) Close the output file descriptor of the pipe

(iv) Now execute the process

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>



int main() {

 int pipefd[2];

 pid_t pid;



 if (pipe(pipefd) == -1) { // Create pipe

 perror("pipe");

 exit(EXIT_FAILURE);

 }



 pid = fork();

 if (pid == -1) { // Fork error

 perror("fork");
```

```
 exit(EXIT_FAILURE);
 } else if (pid == 0) { // Child process
 // Close standard output descriptor
close(STDOUT_FILENO);
 // Duplicate output file descriptor of the pipe
dup2(pipefd[1], STDOUT_FILENO);  // Close input
file descriptor of the pipe  close(pipefd[0]);
 // Execute the process (e.g. ls command)
execlp("ls", "ls", NULL);
 // execlp() will only return if there is an error
perror("execlp");
 exit(EXIT_FAILURE);
 } else { // Parent process
 pid = fork();
 if (pid == -1) { // Fork error
 perror("fork");
 exit(EXIT_FAILURE);
 } else if (pid == 0) { // Child process  //
Close standard input descriptor
close(STDIN_FILENO);
 // Duplicate input file descriptor of the pipe
```

```
dup2(pipefd[0], STDIN_FILENO);
```



```
// Close output file descriptor of the pipe
close(pipefd[1]);
// Execute the process (e.g. wc command)
execlp("wc", "wc", NULL);
// execlp() will only return if there is an error
perror("execlp");
exit(EXIT_FAILURE);
} else { // Parent process
// Close both file descriptors of the pipe
close(pipefd[0]);
close(pipefd[1]);
// Wait for both child processes to finish
wait(NULL);
wait(NULL);
}
}

return 0;
}
```

Q2. Write a program to implement the following command line pipe using pipe() and dup()

```
ls -l | wc -l
```

CODE:

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



```
int main() {
 int fd[2];
 pipe(fd);

 pid_t pid = fork();

 if (pid == 0) {
 // Child process - ls -l
 close(fd[0]); // Close the input end of the pipe
 dup2(fd[1], STDOUT_FILENO); // Redirect stdout to the output end of the pipe
execlp("ls", "ls", "-l", NULL);
 } else {
 // Parent process
 close(fd[1]); // Close the output end of the pipe
 dup2(fd[0], STDIN_FILENO); // Redirect stdin to the input end of the pipe
execlp("wc", "wc", "-l", NULL);
 }

 return 0;
}
```

Named pipe

Named pipe (also known as FIFO) is one of the inter process communication tool. The system for FIFO is as follows

```
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo() makes a FIFO special file with name **pathname**. Here **mode** specifies the FIFO's permissions. The permission can be like : O_CREAT|0644

Open FIFO in read-mode (O_RDONLY) to read and write-mode (O_WRONLY) to write

Q3. Write the output of the following program

```
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>

int main()
{ char buff[25]; int rfd,wfd;
  mkfifo("fif1",O_CREAT|0644);
```

```
if (fork()==0)
{ printf("Child writing into
  FIFO\n");
  wfd=open("fif1",O_WRONLY);
  write(wfd,"Hello",6);

                                rfd=open("fif1",O_RDONLY);
} else {                        read(rfd,buff,6);

  printf("Parent reads from FIFO : %s\n",buff);
 } return
 0;
}
```

Output :

Ex. No. 8	**SCHEDULING ALGORITH**	**Date :**

1. FCFS Scheduling Algorithm

Given n processes with their burst times, the task is to find average waiting time and average turn around time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue.

In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

Turn Around Time: Time Difference between completion time and arrival time. *Turn Around Time = Completion Time – Arrival Time*

Waiting Time(W.T): Time Difference between turn around time and burst time. *Waiting Time = Turn Around Time – Burst Time*

Algorithm:

Step 1. Input the processes along with their burst time (bt).

Step 2. Find waiting time (wt) for all processes.

Step 3. As first process that comes need not to wait so
waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 4. Find waiting time for all other processes i.e. for
process i ->

wt[i] = bt[i-1] + wt[i-1]

Step 5. Find *turnaround time = waiting_time + burst_time*
for all processes.

Step 6. Find *average waiting time = total_waiting_time / no_of_processes*

Step 7. Similarly, find *average turnaround time =*

total_turn_around_time / no_of_processes.

Input : Processes Numbers and their burst times

Output : Process-wise burst-time, waiting-time and turnaround-time
Also display Average-waiting time and Average-turnaround-time

Q1. Write a program to implement FCFS Scheduling algorithm

#include <stdio.h>

//Write the program here

```
#include<stdio.h>

int main()
{
 int n, i, j, wt[20], tat[20], bt[20], at[20], total_wt = 0, total_tat = 0;

  printf("Enter the number of processes: ");
 scanf("%d", &n);

 printf("Enter the burst time and arrival time for each process:\n");
for(i=0;i<n;i++)
 {
 printf("Process %d:\n", i+1);
 printf("Burst time: ");
 scanf("%d", &bt[i]);
 printf("Arrival time: ");
 scanf("%d", &at[i]);
```

```
}

// Calculate waiting time and turnaround time for each process
wt[0] = 0;
for(i=1;i<n;i++)
{
wt[i] = 0;
for(j=0;j<i;j++)
wt[i] += bt[j];
wt[i] -= at[i];
}

for(i=0;i<n;i++)
tat[i] = bt[i] + wt[i];

// Calculate total waiting time and turnaround time
for(i=0;i<n;i++)
{
total_wt += wt[i];
total_tat += tat[i];
}

// Calculate average waiting time and turnaround time
float avg_wt = (float)total_wt/n;
float avg_tat = (float)total_tat/n;

// Print the results
```



```
printf("\nFCFS Scheduling Algorithm:\n");
printf("Process\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time\n");
for(i=0;i<n;i++)
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], at[i], wt[i], tat[i]);
printf("Average Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);

return 0;
```

2. ROUND ROBIN Scheduling Algorithm

In this algorithm, each process is assigned a fixed time slot in a cyclic way

Algorithm:

Step 1. Create an array **rem_bt[]** to keep track of remaining burst time of processes.
This array is initially a copy of bt[] (burst times array)

Step 2. Create another array **wt[]** to store waiting timesof processes. Initialize this array as 0. Step 3. Initialize time : t = 0

Step 4. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet. a) if rem_bt[i] > quantum
t = t + quantum rem_bt[i]
-= quantum;
else // Last cycle for this process
t = t + rem_bt[i]; wt[i] = t
- bt[i] rem_bt[i] = 0;

Step 5. Turnaround time tat[i] = wt[i] + bt[i]. for i^{th} process

Step 6. Find average waiting time and average turnaround time

Input : Processes Numbers and their burst times, time-quantum
Output : Process-wise burst-time, waiting-time and turnaround-time
Also display Average-waiting time and Average-turnaround-time

Q2. Write a program to implement Round Robin Scheduling algorithm

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

void main()
{
   // initlialize the variable name
   int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
   float avg_wt, avg_tat;
   printf(" Total number of process in the system: ");
   scanf("%d", &NOP);
   y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time
for(i=0; i<NOP; i++)
{
printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
printf(" Arrival time is: \t");  // Accept arrival time
scanf("%d", &at[i]);
printf(" \nBurst time is: \t"); // Accept the Burst time
scanf("%d", &bt[i]);
temp[i] = bt[i]; // store the burst time in temp array
}
// Accept the Time qunat
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
// Display the process No, burst time, Turn Around Time and the waiting time
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
for(sum=0, i = 0; y!=0; )
{
```

```
if(temp[i] <= quant && temp[i] > 0) // define the conditions
{
   sum = sum + temp[i];
   temp[i] = 0;
   count=1;
   }
   else if(temp[i] > 0)
   {
      temp[i] = temp[i] - quant;
      sum = sum + quant;
   }
   if(temp[i]==0 && count==1)
   {
      y--; //decrement the process no.
      printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
      wt = wt+sum-at[i]-bt[i];
      tat = tat+sum-at[i];
      count =0;
   }
   if(i==NOP-1)
   {
      i=0;
   }
   else if(at[i+1]<=sum)
   {
      i++;
   }
   else
   {
      i=0;
   }
}
// represents the average waiting time and Turn Around time
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}
```

Ex. No. 5b	OVERLAY CONCEPTS	Date :

Exec() System Call – Overlay Calling process and run new Program
The exec() system call replaces (**overwrites**) the current process with the new process image**.** The PID of the new process remains the same however code, data, heap and stack of the process are replaced by the new program.

There are 6 system calls in the family of exec().All of these functions mentioned below are layered on top of execve(), and they differ from one another and from execve() only in the way in which the program name, argument list, and environment of the new program are specified.

Syntax

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[]) int execvp(const
char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

∑ The names of the first five of above functions are of the form **exec*XY***.

∑ X is either l or v depending upon whether arguments are given in the list format (arg0, arg1, ..., NULL) or arguments are passed in an array (vector).

∑ Y is either absent or is either a p or an e. In case Y is p, the PATH environment variable is used to search for the program. If Y is e, then the environment passed in *envp* array is used.

∑ In case of execvpe, X is v and Y is e. The execvpe function is a GNU extension. It is named so as to differentiate it from the execve system call.

Q1. Execute the Following Program and write the output

```
$vi ex51.c
     #include <stdio.h>
     #include<unistd.h>
     int main()
     { printf("Transfer to execlp function \n");
       execlp("head", "head","-2","f1",NULL);
       printf("This line will not execute \n");
       return 0;
     }
```

Output :

Why second printf statement is not executing? _____

_____

Q2. Rewrite question Q1 with execl() function. Pass the 3rd and 4th argument of the function execl() through command line arguments.

```
$vi ex52.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 5) {
        printf("Insufficient arguments\n");
        return 1;
    }

    printf("Transfer to execl function\n");
    execl("/usr/bin/head", "head", "-n", argv[3], argv[4],
NULL);
    printf("This line will not execute\n");

    return 0;
}
```

Input : `./a.out -3 f1`
Output :

Q3. Rewrite question Q1 with execv() function.

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

```
#include <unistd.h>
int main() {
 printf("Transfer to execv function \n");
 char *args[] = {"head", "-2", "f1", NULL};
 execv("/usr/bin/head", args);
 // This line will only be executed if execv fails to
replace the current process image
 printf("This line will not execute \n");
 return 0;
}
```

Ex. No. 7	MESSAGE QUEUE & SHARED MEMORY	Date :

Message Queue

Message queue is one of the interprocess communication mechanisms. here are two varieties of message queues, System V message queues and POSIX message queues. Both provide almost the same functionality but system calls for the two are different.

There are three system wide limits regarding the message queues. These are, `MSGMNI`, maximum number of queues in the system, `MSGMAX`, maximum size of a message in bytes and `MSGMNB`, which is the maximum size of a message queue. We can see these limits with the `ipcs -l` command

Include the following header files for system V message queues

`<sys/msg.h>, <sys/ipc.h>, <sys/types.h>`

System V Message Queue System Calls

To create a message queue,

```
int msgget (key_t key, int msg_flags);
```

key ⑦ Message queue identifier ex. (key_t)77

flags ⑦`IPC_CREAT|0664` : to create a message queue with permission 0644

`IPC_CREATE|IPC_EXCL|0664` : to create message if doesn't exists

To control the message queue,

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

msquid⑦ Message id returned by msgget()

cmd ⑦`IPC_STAT` : to get the status of a message queue

`IPC_SET` : to change the properties of a message queue `IPC_RMID`

: to remove a message queue

To send a message into message queue,

```
int msgsnd(int msquid, const void *msgp, size_t msqsz, int
msgflg);
```

msquid⑦ Message id returned by msgget()

msgp ⑦ message to be sent. Buffer or message structure is used here.

```
struct buffer { int len; // length of the message int mtype; // message number char buf[50]; // buffer
}x;
```

msqsz ⑦ size of the message msgflg ⑦`IPC_NOWAIT` or `IPC_WAIT` for blocking/non-blocking I/O

To receive a message from message queue,

```
int msgrcv(int msquid, void *msgp, size_t msqsz, long msgtyp, int msgflg);
```

msqp ⑦ the buffer/message structure to receive the message

msgtyp ⑦ message number

msquid, msqsz, msgflg arguments are similar to msgsnd()

Q1. Write a program to send a message (pass through command line arguments) into a message queue. Send few messages with unique message numbers

```
$ vi ex71.c
```

CODE:

```
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 20

// structure for message queue
struct mesg_buffer {
long mesg_type;
char mesg_text[100];
} message;

int main()
{
key_t key;
int msgid;

// ftok to generate unique key
key = ftok("progfile", 65);

// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;
```

```
printf("Write Data : ");
fgets(message.mesg_text,MAX,stdin);

// msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);

// display the message
printf("Data send is : %s \n", message.mesg_text);

return 0;
}
```

OUTPUT:

```
Write Data : hello dania
Data send is : hello dania
```

Q2. Write a program to receive a particular message from the message queue. Use message number to receive the particular message

```
$ vi ex72.c
```

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
long mesg_type;
char mesg_text[100];
} message;

int main()
{
key_t key;
int msgid;

// ftok to generate unique key
key = ftok("progfile", 65);

// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
```

```
// msgrcv to receive message
msgrcv(msgid, &message, sizeof(message), 1, 0);

// display the message
printf("Data Received is : %s \n",
                                message.mesg_text);

// to destroy the message queue
msgctl(msgid, IPC_RMID, NULL);

return 0;
}
```

Linux Commands to control the Interprocess communication tools (Message queue/ semaphore/ shared memory)

ipcs	⑦ to list all IPC information
ipcs –l	⑦ to list the limits of each IPC tools
ipcs –q	⑦ to list message queues details
ipcs –s	⑦ to list semaphore details
ipcs –m	⑦ to list all shared memory details
ipcs –u	⑦ to get the current usage of IPC tools
ipcs –h	⑦ ipcs help
ipcrm –q <msgid>	⑦ to remove a message queue with message-id <msgid>
ipcrm –m <shmid>	⑦ to remove a shared memory
ipcrm –s <semid>	⑦ to remove a semaphore
ipcrm –h	⑦ ipcrm help

Shared memory

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.

Include the following header files for shared memory

```
<sys/ipc.h>,  <sys/shm.h>, <sys/types.h>
```

System V Shared memory System Calls

To create a shared memory,

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

key ⑦ shared memory id size ⑦ shared memory size in bytes

shmflg ⑦ `IPC_CREATE|0664` : to create a new shared memory segment

`IPC_EXCL|IPC_CREAT|0664` : to create new segment and the call fails, if the segment already exists

To attach the shared memory segment to the address space of the calling process

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

shmid ⑦ Shared memory id returned by shmget() shmaddr ⑦ the attaching address. If shmaddr is NULL, the system by default chooses the suitable address. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary AddressShmflg ⑦ SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment)

To control the shared memory segment,

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

To detach the shared memory segment from the address space of the calling process `int shmdt(const void *shmaddr)`

shmaddr ⑦ address of the shared memory to detach

Q3. Write a program to do the following:

- ∑ Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- ∑ In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- ∑ The write process writes the data read from the standard input into the shared memory. Last byte signifies the end of buffer
- ∑ Read process would read from the shared memory and write to the standard output

```
$ vi ex73.c
```

Verified by

To write:

```
#include <iostream>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

using namespace std;
```

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

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

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

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

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

        //detach from shared memory
        shmdt(str);

        return 0;
}
```

To read :

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

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

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

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

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

        //detach from shared memory
        shmdt(str);

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

        return 0;
}
```

Ex. No. 9	PROCESS SYNCHRONIZATION	Date :

Q1. Execute and write the output of the following program for *mutual exclusion* using system V semaphore

```
#include<sys/ipc.h>
#include<sys/sem.h>
int main()
{ int pid,semid,val; struct sembuf sop;
 semid=semget((key_t)6,1,IPC_CREAT|0666;
 pid=fork();

 sop.sem_num=0;
 sop.sem_op=0;
 sop.sem_flg=0;

 if (pid!=0)
 {sleep(1);
   printf("The Parent waits for WAIT signal\n");
   semop(semid,&sop,1);
   printf("The Parent WAKED UP & doing her job\n");
   sleep(10);
   printf("Parent Over\n");
 }
 else
 { printf("The Child sets WAIT signal & doing herjob\n");
 semctl(semid,0,SETVAL,1);
 sleep(10);
   printf("The Child sets WAKE signal & finished herjob\n");
   semctl(semid,0,SETVAL,0); printf("Child Over\n");
 }
return 0;
 }
```

Output :

THREAD

Thread is the segment of a process means a process can have multiple threads and these multiple threads are contained within a process. The thread takes less time to terminate as compared to the process but unlike the process, threads do not isolate.

THREAD FUNCTIONS

The header file for POSIX thread functions is `pthread.h`. To execute the c file with thread, do as follows:

```
gcc -pthread file.c    (or) gcc
-lpthread file.c
```

To create a new thread,

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

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

To terminate a thread

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

This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

To wait for the termination of a thread.

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

This method accepts following parameters:

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

POSIX SEMAPHORE

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to include `semaphore.h` and compile the code by linking with `-lpthread lrt`

To lock a semaphore or wait

```
int sem_wait(sem_t *sem);
```

To release or signal a semaphore

```
int sem_post(sem_t *sem);
```

To initialize a semaphore

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Where, **`sem`** : Specifies the semaphore to be initialized.

`pshared` : This argument specifies whether or not the newly initialized semaphore is shared between processes/threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

`value` : Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore

```
sem_destroy(sem_t *mutex);
```

Q2. Execute and write the output of the following program for *mutual exclusion* using POSIX semaphore and threads.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
sem_t mutex;

void* thread(void* arg)
{
      //wait sem_wait(&mutex);
      printf("\nEntered..\n")
      ;

      //critical section
      sleep(4);

      //signal
      printf("\nJust Exiting...\n");
      sem_post(&mutex);
}
```

```
int main()
{ sem_init(&mutex, 0, 1); pthread_t t1,t2;
      pthread_create(&t1,NULL,thread,NULL);
      sleep(2);
      pthread_create(&t2,NULL,thread,NULL);
      pthread_join(t1,NULL);
      pthread_join(t2,NULL);
      sem_destroy(&mutex); return 0;
}
```

Ex. No. 10a	READER-WRITER PROBLEM	Date :

Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource at that time.

```
#include<semaphore.h>

#include<stdio.h>

#include<stdlib.h>

sem_t x,y;

pthread_t tid;

pthread_t writerthreads[100],readerthreads[100];

int readercount;

void *reader(void* param)

{

 sem_wait(&x);//acquire

 readercount++;

 if(readercount==1)

 sem_wait(&y);//acquires y to block the writer

 sem_post(&x);//release
```

```
 printf("\n%d reader is inside",readercount);
 sem_wait(&x);//reacquire x
 readercount--;
 if(readercount==0)
 {
 sem_post(&y);
 }
 sem_post(&x);
 printf("\n%d Reader is leaving",readercount+1);
}
void *writer(void* param)
{
 printf("\nWriter is trying to enter");
 sem_wait(&y);
 printf("\nWriter has entered");
 sem_post(&y);
 printf("\nWriter is leaving");
}
int main()
{
 int n2,i;
 printf("Enter the number of readers:");
 scanf("%d",&n2);
 int n1[n2];
 sem_init(&x,0,1); //intializes the semaphore x with the value of 1
 sem_init(&y,0,1); //intializes the semaphore x with the value of 1
```

```
for(i=0;i<n2;i++)
{
pthread_create(&writerthreads[i],NULL,reader,NULL);
pthread_create(&readerthreads[i],NULL,writer,NULL);
}
for(i=0;i<n2;i++)
{
pthread_join(writerthreads[i],NULL);
pthread_join(readerthreads[i],NULL);
}
}
```

Output:

Ex. No. 10b	DINING PHILOSOPHER PROBLEM	Date :

Problem Statement

The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begin thinking again.

```
#include <pthread.h>
```

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;
      sleep(2);
        printf("Philosopher %d takes fork %d and %d\n",
```

```
                phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);
        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
}
}

// take up chopsticks
void take_fork(int phnum)
{

sem_wait(&mutex);

// state that hungry
state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating
test(phnum);
sem_post(&mutex);
```

```
// if unable to eat wait to be signalled
sem_wait(&S[phnum]);

sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

sem_wait(&mutex);
// state that thinking
state[phnum] = THINKING;
printf("Philosopher %d putting fork %d and %d down\n",
        phnum + 1, LEFT + 1, phnum + 1);
printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);
test(RIGHT);

sem_post(&mutex);
}

void* philosopher(void* num)
{
```

```
while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
}
}

int main()
{

int i;
pthread_t thread_id[N];

// initialize the semaphores
sem_init(&mutex, 0, 1);

for (i = 0; i < N; i++)

        sem_init(&S[i], 0, 0);

for (i = 0; i < N; i++) {
```

```
        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                                philosopher, &phil[i]);

        printf("Philosopher %d is thinking\n", i + 1);
}

for (i = 0; i < N; i++)

        pthread_join(thread_id[i], NULL);
}
```