Ex. No. 7	SHELL PROGRAMS

How to run a Shell Script

- Edit and save your program using editor
- Add execute permission by *chmod* command
- Run your program using the name of your program

```
./program-name
```

Important Hints

- No space before and after the assignment operator Ex. `sum=0`
- *Single quote* ignores all special characters. Dollar sign, Back quote and Back slash are not ignored inside *Double quote*. *Back quote* is used as command substitution. *Back slash* is used to remove the special meaning of a character.
- Arithmetic expression can be written as follows : `i=$((i+1))` or `i=$(expr $i + 1)`
- Command line arguments are referred inside the programme as `$1, $2`, ..and so on
- `$*` represents all arguments, `$#` specifies the number of arguments
- `read` statement is used to get input from input device. Ex. `read a b`

Syntax for if statement

```
if [ condition ]
then
      ...
elif [ condition ]
then
      ...
else
      ...
fi
```

Syntax for case structure

```
case value in
pat1) ...
                statement;;
pat2)  ...
                Statement;;
*)     ...
                Statement;;
esac
```

Syntax for for-loop

```
for var in list-of-values
do
          ...
          ...
done
```

Syntax for While loop

```
while command_t
do
          ...
          ...
done
```

Syntax for printf statement

```
printf  "string and format"  arg1  arg2 … …
```

- Break and continue statements functions similar to C programming
- Relational operators are –lt, -le, -gt, -ge, -eq,-ne
- Ex. (i>= 10) is written as [$i -ge 10]
- Logical operators (and, or, not) are -o, -a, !
- Ex. (a>b) && (a>c) is written as [$a –gt $b –a $a –gt $c]
- Two strings can be compared using = operator

Q1. Given the following values

num=10, x=*, y=`date` a="Hello, 'he said'"

Execute and write the output of the following commands

Command	Output
echo num	
echo $num	
echo $x	
echo '$x'	
echo "$x"	
echo $y	
echo $(date)	
echo $a	
echo \$num	
echo \$$num	

Q2. Find the output of the following shell scripts

```
$ vi  ex51
     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 :

Q3. Write a program to check whether the file has execute permission or not. If not, add the permission.

```
$ vi ex52
```

Q4. Write a shell script to print a greeting as specified below.
If hour is greater than or equal to 0 (midnight) and less than or equal to 11 (up to 11:59:59), "Good morning" is displayed.
If hour is greater than or equal to 12 (noon) and less than or equal to 17 (up to 5:59:59 p.m.), "Good afternoon" is displayed.
If neither of the preceding two conditions is satisfied, "Good evening" is displayed.

```
$ vi  ex53
  hour=$(date | cut -c12-13)
  if [ "$hour" -ge 0 -a "$hour" -le 11 ]
  then                                    ← complete the program
```

Q5. Write a shell script to list only the name of sub directories in the present working directory

```
$ vi ex54
```

Q6. 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 ex55
```

Outcome:

Thus the basics of shell programming have been learnt successfully and output is verified.

Ex. No. 8	PROCESS CREATION	Date :

Syntax for process creation

int fork();

Returns 0 in child process and child process ID in parent process.

Other Related Functions

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

Virtual fork

vfork() function 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

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 :

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.

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

int main()
{
  int pid,n,oddsum=0,evensum=0;

  printf("Enter the value of n : ",a);
  scanf("%d",&n);
  pid=fork();
  // Complete the program









  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 :

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()
{
```

```
return 0;
}
```

Sample Output:

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

In Parent Process
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 :

Outcome:

Learned creation of process successfully .

Ex. No. 9 & 10	OVERLAY CONCEPTS

Objective:

To understand the overlay concepts and practice how to overlay the current process to new process in Linux using C.

Overlay

Overlay is the concept which enables the user to run another new process from the currently running process address space.

System call used :

Exec() System Call

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 envpl))

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 envpl])

- The names of the first five of above functions are of the form execXY.
- X is either 1 or v depending upon whether arguments are given in the list format (argo, argl, ..., 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.

Simple overlay concept

Procedure:

- Stepl: Create two different c programs. Name it as example.c file and hello.c file
- Step2: Make example .c is the current running process,
- Step3: call the function execv() which takes the hello.c as an argument.
- Step4: Print process id of both the processes(hello.c and example.c processes).
- Step5: trace the system control by having simple print statement in both programs.

Expected Output:

$ process id of example.c=4733

We are in hello.c

Process id of hello.c=4733

COMBINING FORK() AND EXEC() SYSTEM CALL

Procedure:

- Stepl: Create two different c programs. Name it as example.c file and hello.c file
- Step2: Make example .c is the current running process and use fork() system call to create child process,
- Step3: call the function execv() in child process which takes the hello.c as an argument.
- Step4: Print Process id of both parent, child and overlay processes (hello.c and example.c

processes).

- Step5: trace the system control by having simple print statement in both programs.

Expected Output:
$ process id of example.c=4790

The control is in parent process

The control is in child process

Process id of child = 4791

Calling hello.c from child

We are in hello.c

Process id of hello.c=4791

PRACTICE QUESTIONS:

1. Execute the Following Program and write the output
$vi ex1.c.

```
#include <stdio.h>
#include <unistd.h>
 int main()
printf("Transfer to execlp function \n");
execlp("head", "head",""-2","fl",NULL); // Assume fl is any text file
printf("This line will not execute \n");
return 0;
```

Output :

Why second printf statement is not executing?

2. Rewrite question 1 with execl() function. Pass the 3rd and 4th argument of the function

execl() through command line arguments.

$vi ex2.c

Input : /a.out -3 fi

Output:

Outcome:

Learned and implemented the overlay concept in Linux using C.

Ex. No: 11	INTER PROCESS COMMUNICATION- Pipe()

Objective:

Inter process communication between the processes using pipe concept.

Inter process communication (IPC):

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co operating process can be affected by other executing processes.

Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

- Shared Memory
- Message passing

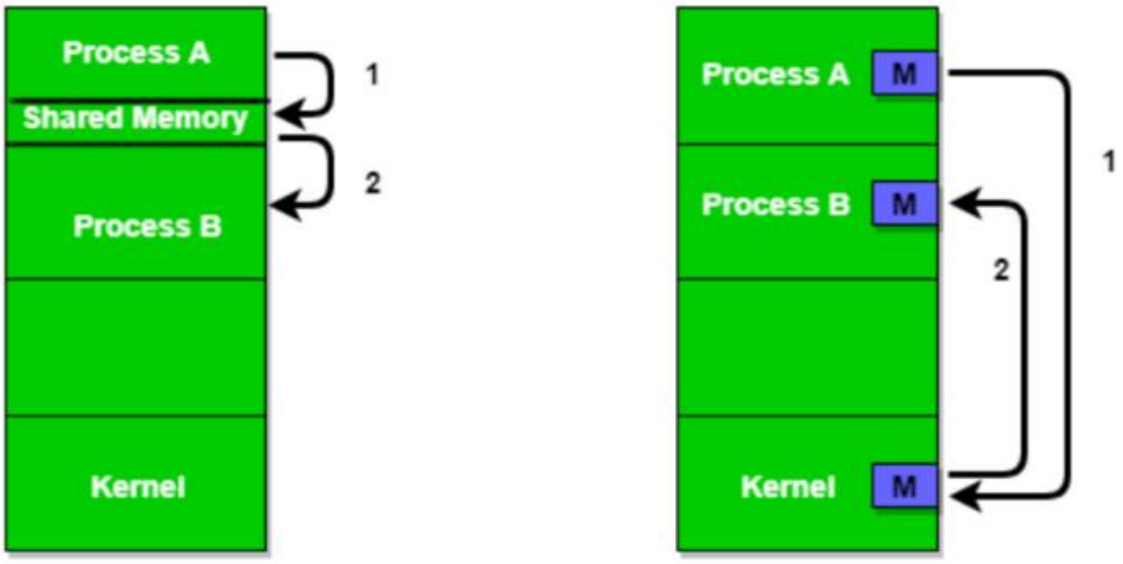


Figure 1 - Shared Memory and Message Passing

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

Important methods for inter process communication:

Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

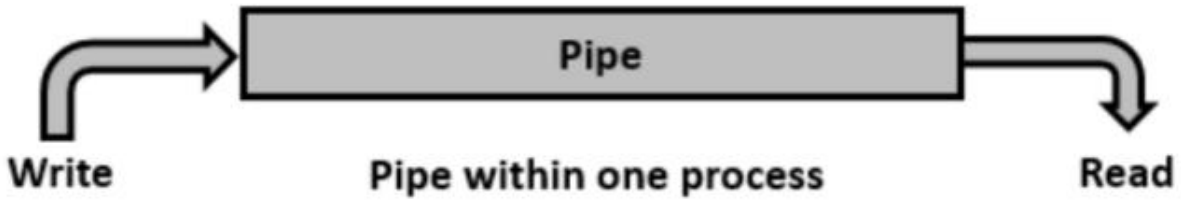
- Send (message)- message size fixed or variable
- Received (message)

Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

PIPES

8MPipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.




```
#include<unistd.h>

int pipe(int pipedes[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 pipedes[0] is for reading and pipedes[1] is for writing. Whatever is written into pipedes[1] can be read from pipedes[0].

This call would return zero on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

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

#include <fontl.h>

int open(const char *pathname, int flags);

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

Even though the basic operations for file are read and write, it is essential to open the file before performing the operations and closing the file after completion of the required operations. Usually, by default, 3 descriptors opened for every process, which are used for input (standard input - stdin), output (standard output - stdout) and error (standard error stderr) having file descriptors 0, 1 and 2 respectively.

Uni directional pipe:

Algorithm

- **Step 1** - Create a pipe using pipe() system call.
- **Step 2** – Send a message to the one end of the pipe.

- **Step 3** – Retrieve the message from the other end of the pipe and write it to the standard output.

Expected Output:

hello, world #1

hello, world #2

hello, *w*orld #3

Implementing command line pipe using exec() family of function:

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 input file descriptor of the pipe
(iv) Now execute the process

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.

Write and read two messages using pipe.

Algorithm

Step 1 - Create a pipe.

Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

Step 4 - Send another message to the pipe.

Step 5 - Retrieve the message from the pipe and write it to the standard output.

Note – Retrieving messages can also be done after sending all messages.

Expected Output:

Writing to pipe - Message 1 is Hi
Reading from pipe - Message 1 is Hi
Writing to pipe - Message 2 is Hello
Reading from pipe - Message 2 is Hello

2. Program to write and read two messages through the pipe using the parent and the child processes.

Algorithm

Step 1 - Create a pipe.

Step 2 - Create a child process.

Step 3 – Parent process writes to the pipe.

Step 4 - Child process retrieves the message from the pipe and writes it to the standard output.

Step 5 – Repeat step 3 and step 4 once again.

Expected Output:

Parent Process - Writing to pipe - Message 1 is Hi

Parent Process - Writing to pipe - Message 2 is Hello

Child Process - Reading from pipe - Message 1 is Hi

Child Process - Reading from pipe – Message 2 is Hello

Two-way Communication Using Pipes

Following are the steps to achieve two-way communication -

Step 1 - Create two pipes. First one is for the parent to write and child to read, say as pipe l. Second one is for the child to write and parent to read, say as pipe2.

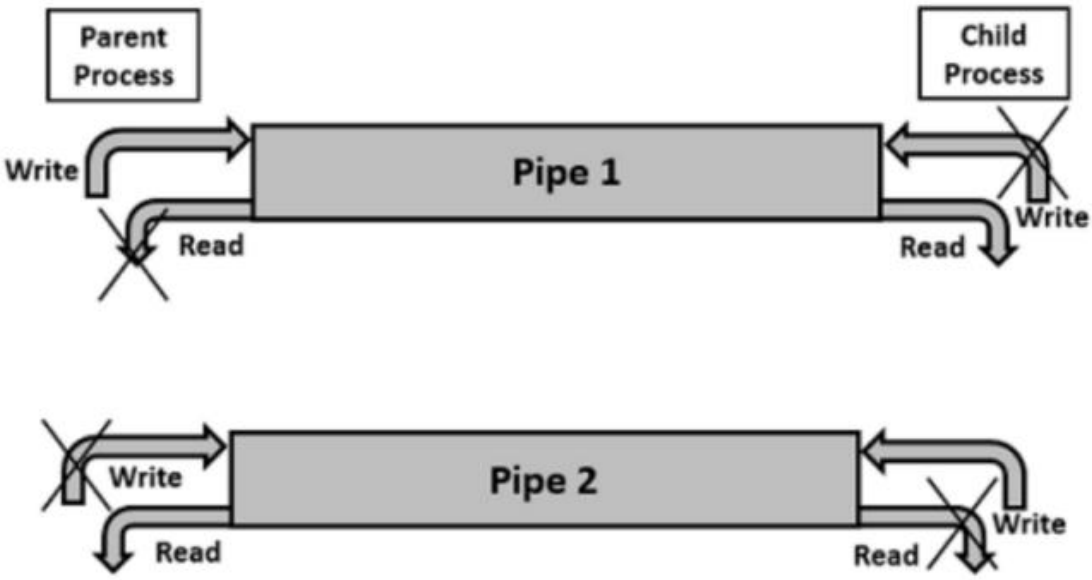
Step 2 - Create a child process.

Step 3 – Close unwanted ends as only one end is needed for each communication.

Step 4 - Close unwanted ends in the parent process, read end of pipel and write end of pipe2.

Step 5 – Close the unwanted ends in the child process, write end of pipel and read end of pipe2.

Step 6 - Perform the communication as required.



3. Achieving two-way communication using pipes

Algorithm

Step 1 - Create pipel for the parent process to write and the child process to read.

Step 2 - Create pipe2 for the child process to write and the parent process to read.

Step 3 – Close the unwanted ends of the pipe from the parent and child side.

Step 4 - Parent process to write a message and child process to read and display on the screen. **Step 5** – Child process to write a message and parent process to read and display on the screen.

Expected Output:

In Parent: Writing to pipe 1 - Message is Hi

In Child: Reading from pipe 1 - Message is Hi

In Child: Writing to pipe 2 – Message is Hello

In Parent: Reading from pipe 2 – Message is Hello

Practice:

Q. 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]; 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:

Outcome:

Inter process communication using pipes concept learned and implemented.

Ex. No: 12 (a)	INTER PROCESS COMMUNICATION -Shared memory

Objective:

The program to implement Interprocess Communication using shared memory and message queue concept.

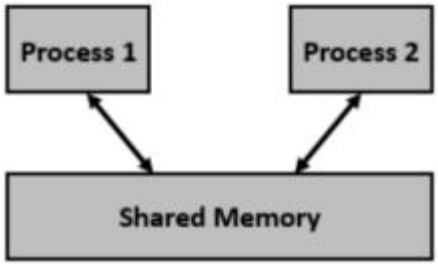
Inter Process Communication:

The popular IPC techniques are Shared Memory and Message Queues.

Shared memory:

We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls,

- Create the shared memory segment or use an already created shared memory segment (shmget())
- Detach the process from the already attached shared memory segment (shmdt())
- Control operations on the shared memory segment (shmctl())



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.
- **shm**dt(): 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); The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are -

- **IPC_STAT** - Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.
- **IPC_SET** - Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.
- **IPC_RMID –** Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it. **IPC_INFO** - Returns the information about the shared memory limits and parameters in the structure pointed by buf.
- **SHM_INF**O – Returns a shm_info structure containing information about the consumed system resources by the shared memory.

Program:1- Shared memory implementation using readers writers problem.

Writer process:

Algorithm:

- **Step 1** - Create a shared memory using (shmget()) function.
- **Step 2** - attach the current process in to created shared memory be calling shmat() function.
- **Step 3** - Write into shared memory after attaching in to it.
- **Step 4-** After completing write operation detach the process from shared memory area.

Reader process:

Algorithm:

- **Step 1** - Create a shared memory using (shmget()) function.
- **Step 2** - attach the current process in to created shared memory be calling shmat() function.
- **Step 3** - read the data which is already written by the reader process from shared memory after attaching in to it.
- **Step 4-** Print the string and detach the process from shared memory area.

Expected Output:

Writer.c

Write Data : Operating System Data Written in memory: Operating System

Reader.c

Data read from memory: Operating System

Outcome:

Interprocess communication using shared memory concept learned and implemented.

Ex. No: 12b	INTER PROCESS COMMUNICATION -Message Queue

Objective:

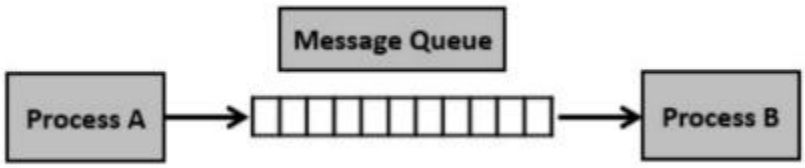
The program to implement Inter process communication using message queue concept.

Inter Process Communication-Message Queue

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.

Program :To perform communication using message queues, following are the steps -

Writer Process:

- **Step 1** - Create a message queue or connect to an already existing message queue (msgget())
- **Step 2** – specify the message type as 1.
- **Step 3-** Write into message queue (msgsnd())
- **Step 4**- terminate the process

Reader Process:

- **Step 1** - Create a message queue or connect to an already existing message queue (msgget())
- **Step 2** – specify the message type as 1.
- **Step 3** – Read from the message queue (msgrev())
- **Step 4** - Perform control operations on the message queue (msgctl())
- **Step 5** – terminate the reader process

Expected Output:

Writer.c

Write Message : Hello

Sent Message : Hello

Reader.c

Received Message is : Hello

Outcome:

Thus the concept of Interprocess Communication using message Queue has been implemented using readers writers problem.