Inter process communication

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types −

- Between related processes initiating from only one process, such as parent and child processes.
- Between unrelated processes, or two or more different processes

Following are some important terms that we need to know before proceeding further on this topic.

- **Pipes** − Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.
- **FIFO** − Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.
- **Message Queues** − Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.
- **Shared Memory** − Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.
- **Semaphores** − Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.
- **Signals** − Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

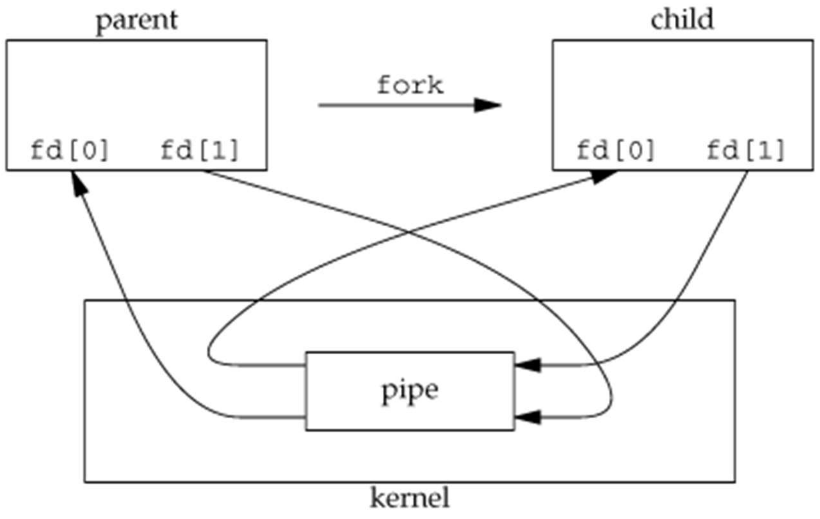
IPC between processes on a Single System:

It includes four general approaches:

- Shared memory
- Messages
- Pipes
- Sockets

Pipes-creation and IPC between Related Processes using unnamed pipes/Ordinary pipes

- Pipe is used to combine two or more command and in this the output of one command act as input to another command and this command output may act as input to next command and so on.You can make it do so by using the pipe character **'|'**.
- 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).
  - When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



pipe( ) - create pipe

Synopsis:

#include <unistd.h>
int pipe(int filedes**[2]);**

Description:

pipe( ) creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes:

- ✓ filedes[0] is for reading,
- ✓ filedes[1] is for writing.

The following rules apply to processes that read from a pipe:

- If a process reads from a pipe whose write end has been closed, the read ( ) returns a 0, indicating end-of-input.
- If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available.
- If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and read ( ) returns the number of bytes actually read.

The following rules apply to processes that write to a pipe:

- If a process writes to a pipe whose read end has been closed, the write fails and the writer is sent a SIGPIPE signal.
  - ✓ The default action of this signal is to terminate the writer.
- If a process writes fewer bytes to a pipe than the pipe can hold, the write ( ) is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
- If a process writes more bytes to a pipe than the pipe can hold, no similar guarantees of atomicity apply.

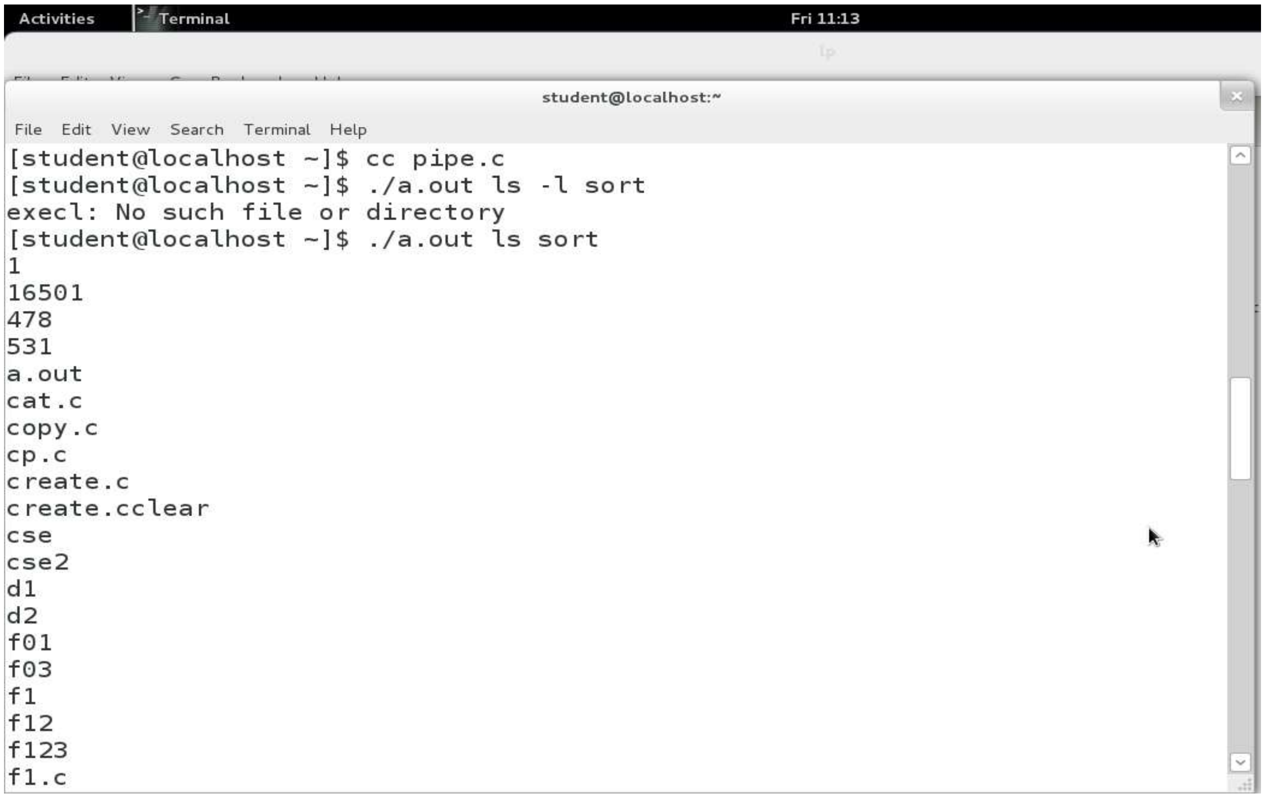
Return Value:

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Example: pipe program for ordinary pipe or unnamed pipe

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
int main(int argc,char *argv[])
{
int fd[2],pid,k;
k=pipe(fd);
if(k==-1)
{
perror("pipe");
exit(1);
}
pid=fork();
if(pid==0)
{
close(fd[0]);
dup2(fd[1],1);
close(fd[1]);
execlp(argv[1],argv[1],NULL);

perror("execl");
}
else
{
wait(2);
close(fd[1]);
dup2(fd[0],0);
close(fd[0]);
execlp(argv[2],argv[2],NULL);
perror("execl");
}
}
```

OUTPUT:



```
[student@localhost ~]$ cc pipe.c
[student@localhost ~]$ ./a.out ls -l sort
execl: No such file or directory
[student@localhost ~]$ ./a.out ls sort
1
16501
478
531
a.out
cat.c
copy.c
cp.c
create.c
create.cclear
cse
cse2
d1
d2
f01
f03
f1
f12
f123
f1.c
```

- ❖ <u>One of the major disadvantages of pipes is that</u> the they cannot be accessed using their names by any other process other than child and the parent as they do not get listed in the directory tree.
- ❖ The work around for this problem is to create a named pipe which is also called as a FIFO, which stands for First in First out, meaning the data that is written into the pipe first will be read out first always

FIFOs – Creation and IPC between unrelated Processes using Named Pipes:

- FIFOs are created using the function **mkfifo( ).** Which takes as arguments.
  - ✓ The name of the fifo that has to be created
  - ✓ The permissions for the file.
- The FIFOs get listed in the directory tree and any process can access it using its name by providing the appropriate path.

❖ **mkfifo ( )** - make a FIFO special file (a named pipe)

- Here i want create two named pipe files or special files using mkfifo system call or command.

```
cse@nnrg] $ mkfifo fifo_server
cse@nnrg] $ mkfifo fifo_client
```

❖ Now to use these pipes, we will write two program's one to represent the server and other to represent the client. The server will read from the pipe fifo_server to which the client will send a request. On reciveing the request, the server will send the information to the client on fifo_client.

server.c

```
#include<stdio.h>
#include<fcntl.h>
main( )
{
FILE *file1;
int fds,fdc;
int choice;
char *buf;
fds = open("fifo_server",O_RDWR);
if(fds<1) {
 printf("Error opening file");
 }
read(fds,&choice,sizeof(int));
sleep(10);
fdc = open("fifo_client",O_RDWR);

if(fdc<1) {
 printf("Error opening file");
 }
switch(choice) {

case 1:
 buf="Linux";
 write(fifo_client,buf,10*sizeof(char));
 printf("\n Data sent to client \n");
 break;
case 2:

 buf="Fedora";
 write(fifo_client,buf,10*sizeof(char));
 printf("\nData sent to client\n");
 break;
case 3:
 buf="2.6.32";
 write(fifo_client,buf,10*sizeof(char));
 printf("\nData sent to client\n");
}
close(fds);
close(fdc);
}
```

10*sizeof(int)

To allocate block of memory dynamically.

sizeof is greatly used in dynamic memory allocation. For example, if we want to allocate memory for which is sufficient to hold 10 integers and we don't know the sizeof(int) in that particular machine. We can allocate with the help of sizeof.

- The above code,server.c, reads the choice from fifo_server to which the client writes and depending on the request,the server responds with the relevant data by writing to fifo_client.

  Save the file and server.c and compile it as follows

```
$cc server.c
```

client.c:

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main()
{
 FILE *file1;
 int fds,fdc;
 char str[256];
 char *buf;
 int choice=1;
 printf("Choose the request to be sent to server from options below");
 printf("\n\t\t Enter 1 for O.S.Name \n \
              Enter 2 for Distribution \n \
              Enter 3 for Kernel version \n");
 scanf("%d",&choice);

 fds=open("fifo_server",O_RDWR);
 if(fds < 0) {
  printf("Error in opening file");
  exit(-1);
  }

 write(fds,&choice,sizeof(int));
fdc=open("fifo_client",O_RDWR);

if(fdc< 0) {
 printf("Error in opening file");
 exit(-1);
 }

buf=malloc(10*sizeof(char));
read (fifo_client,buf,10*sizeof(char));
printf("\n ***Reply from server is %s***\n",buf);
close(fds);
close(fdc);
}
```

- The above code,client.c, sends a request to the server by writing to the pipe fifo_server, and recieves the reply from the server by reading the pipe fifo_client.

  Save the file and client.c and compile it as follows

```
$cc client.c
```

- To see the pipe in operation, open two terminals and go the folder where the pipes have been created.

  Note: The server.c and client.c codes assume that the pipes exist in the same directory as the executables of server and client are, if they exist in any other directory you will have to provide the path to the same in the system call open().

Terminal 1:

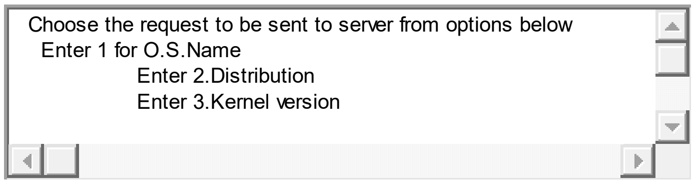
```
. / server
```

The terminal will go into a wait state with the cursor blinking, waiting for request from client.

Terminal 2:

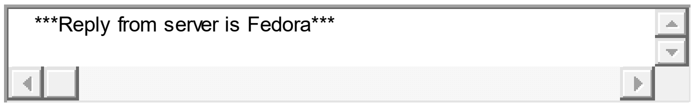
```
. / clientt
```

The client will prompt to make a choice of request to be sent to the server, enter number 1,2 or 3



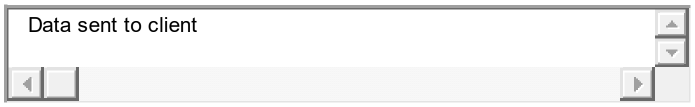
```
Choose the request to be sent to server from options below
  Enter 1 for O.S.Name
                Enter 2.Distribution
                Enter 3.Kernel version
```

- This number will be sent to the server and the client will go into a wait state, waiting for the server to respond.
- After a few seconds you should see the reponse from the server being printed on the client terminal. **Note**: wait for 10 seconds for the output to appear as the server has a sleep for 10 seconds, this is required to allow some time for the client to start its read operation on the fifo_client pipe.
  - **For eg:** if we entered option 2, the response would be output on **client terminal**:



```
***Reply from server is Fedora***
```

Ouput on **server terminal** :



```
Data sent to client
```

  - Thus we were able to communicate between the two processes using the FIFOs, even though the pipes were not created by them.

The major differences between named and unnamed pipes are:-

1. As suggested by their names, a named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name.
   On the other hand, unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.
2. An unnamed pipe is only used for communication between a child and it's parent process, while a named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.

3. A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes.

   On the other hand, an unnamed pipe vanishes as soon as it is closed, or one of the process (parent or child) completes execution.
4. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system. Unnamed pipes are always local; they cannot be used for communication over a network.
5. A Named pipe can have multiple processes communicating through it, like multiple clients connected to one server. On the other hand, an unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.

Message Queues:

Why do we need message queues when we already have the shared memory?

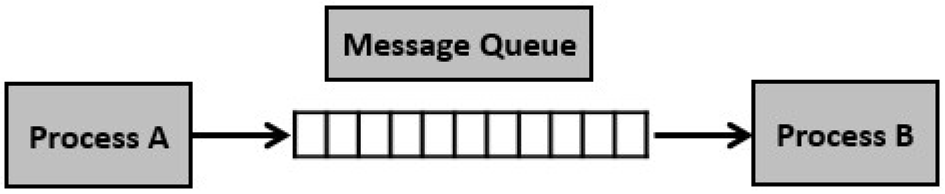
It would be for multiple reasons, let us try to break this into multiple points for simplification −

- As understood, once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.
- If we want to communicate with small message formats.
- Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.
- Frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality. Not worth with regard to utilization in this kind of cases.
- What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to implement with message queues.
- If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simpler to implement with message queues. To simplify the given message type as 1, 10, 20, it can be either 0 or +ve or –ve as discussed below.
- Of course, the order of message queue is FIFO (First In First Out). The first message inserted in the queue is the first one to be retrieved.

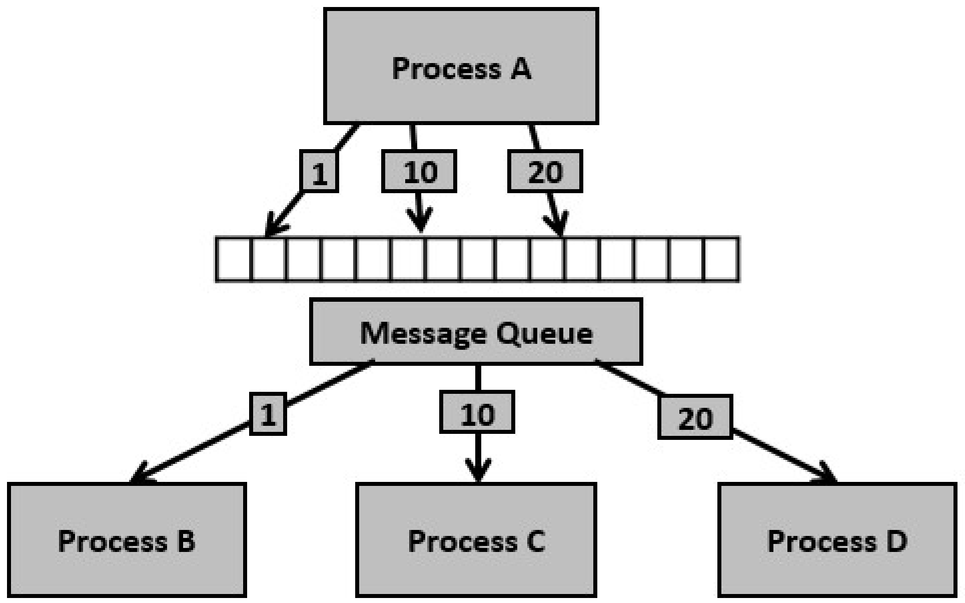
❖ Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.

Communication using message queues can happen in the following ways −

- Writing into the shared memory by one process and reading from the shared memory by another process. As we are aware, reading can be done with multiple processes as well.



- Writing into the shared memory by one process with different data packets and reading from it by multiple processes, i.e., as per message type.
-



Having seen certain information on message queues, now it is time to check for the system call (System V) which supports the message queues.

- ❖ System V (System 5) was an early form of the Unix operating system, originally developed by AT&T (American Telephone and Telegraph). The first release, Release 1 (SVR1), appeared in 1983. Release 2 (SVR2) followed in 1984, Release 3 (SVR3) in 1987, and Release 4 (SVR4, the last and most popular version) in 1990.
- ❖ System V has been compared to BSD (which originally stood for Berkeley Software Distribution), another "flavor" of Unix. The first three versions of System V were preferred by businesses, while BSD was favored by university professors and research scientists.

❖ **To perform communication using message queues, following are the steps −**

Step 1 − Create a message queue or connect to an already existing message queue (**msgget( )**)

Step 2 − Write into message queue (**msgsnd( )**)

Step 3 − Read from the message queue (**msgrcv(**))

Step 4 − Perform control operations on the message queue (**msgctl( )**)

Now, let us check the syntax and certain information on the above calls

API for message Queues:

Creation and accessing of a message queue:

❖ **mesgget( ) –** Create a message Queue **or** get a System V message queue identifier.

Synopsis:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

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

Description:

This system call creates or allocates a System V message queue. Following arguments need to be passed −

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok( ).
- The second argument, msgflg, specifies the required message queue flag/s such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

Return Value:

If successful, the return value will be the message queue identifier (a nonnegative integer), otherwise -1 with errno indicating the error.

Sending a message queue:

❖ **msgsnd( ) -** sends/appends a message into the message queue

Synopsis:

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

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)
```

Description:

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()
- The second argument, msgp, is the pointer to the message, sent to the caller, defined in the structure of the following form −

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

- ➢ The variable mtype is used for communicating with different message types, explained in detail in msgrcv( ) call.
- ➢ The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of message (the message should end with a null character)
- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

Return Value:

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror( ) function.

Retrieving a message:

❖ **msgrcv( )** - retrieves the message from the message queue.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

Description:

This system call retrieves the message from the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget( )
- The second argument, msgp, is the pointer of the message received from the caller. It is defined in the structure of the following form –

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

- ➢ The variable mtype is used for communicating with different message types. The variable mtext is an array or other structure whose size is specified by msgsz (positive value).
- ➢ If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of the message received (message should end with a null character)
- The fouth argument, msgtype, indicates the type of message −
  - **If msgtype is 0** − Reads the first received message in the queue
  - **If msgtype is +ve** − Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)
  - **If msgtype is –ve** − Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)
- The fifth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes).

Return Value:

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

Controlling the message queue:

❖ **msgctl ( )** - message control operations

Synopsis:

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

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

Description:

This system call performs control operations of the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget( )
- The second argument, cmd, is the command to perform the required control operation on the message queue. Valid values for cmd are −
  - ✓ **IPC_STAT** − Copies information of the current values of each member of struct msqid_ds to the passed structure pointed by buf. This command requires read permission on the message queue.
  - ✓ **IPC_SET** − Sets the user ID, group ID of the owner, permissions etc pointed to by structure buf.
  - ✓ **IPC_RMID** − Removes the message queue immediately.
  - ✓ **IPC_INFO** − Returns information about the message queue limits and parameters in the structure pointed by buf, which is of type struct msginfo
  - ✓ **MSG_INFO** − Returns an msginfo structure containing information about the consumed system resources by the message queue.
- The third argument, buf, is a pointer to the message queue structure named struct msqid_ds. The values of this structure would be used for either set or get as per cmd.

Return Value:

- This call would return the value depending on the passed command.

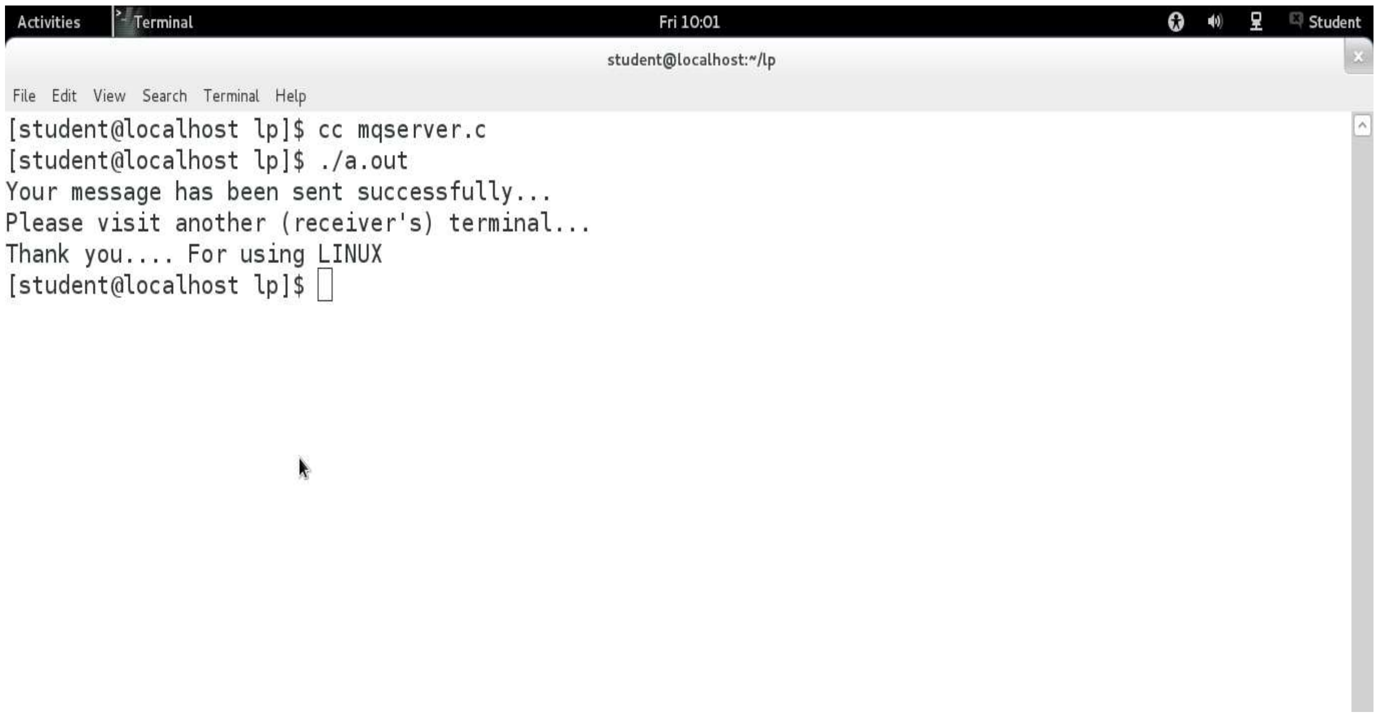
- Success of IPC_INFO and MSG_INFO or MSG_STAT returns the index or identifier of the message queue or 0 for other operations and -1 in case of failure.

Clent/server Example:

mqserver.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
#include<stdlib.h>
#define SIZE 2000
void main()
{
int mfd,mfd2,mfd3;
struct
{
double mtype;
char mtext[2000];
}s1,s2,s3;
if((mfd=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
exit(1);
}
s1.mtype=1;
sprintf(s1.mtext,"%s","Hi friends... My name is message1");
if(msgsnd(mfd,&s1,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}
if((mfd2=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
exit(1);
}
s2.mtype=1;
sprintf(s2.mtext,"%s","Hi friends... My name is message2");
if(msgsnd(mfd2,&s2,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}

if((mfd3=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
exit(1);
}
s3.mtype=1;
sprintf(s3.mtext,"%s","Hi friends... My name is message3");
if(msgsnd(mfd3,&s3,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}
printf("Your message has been sent successfully...\n");
printf("Please visit another (receiver's) terminal...\n");
printf("Thank you.... For using LINUX\n");
}
```

OUTPUT:

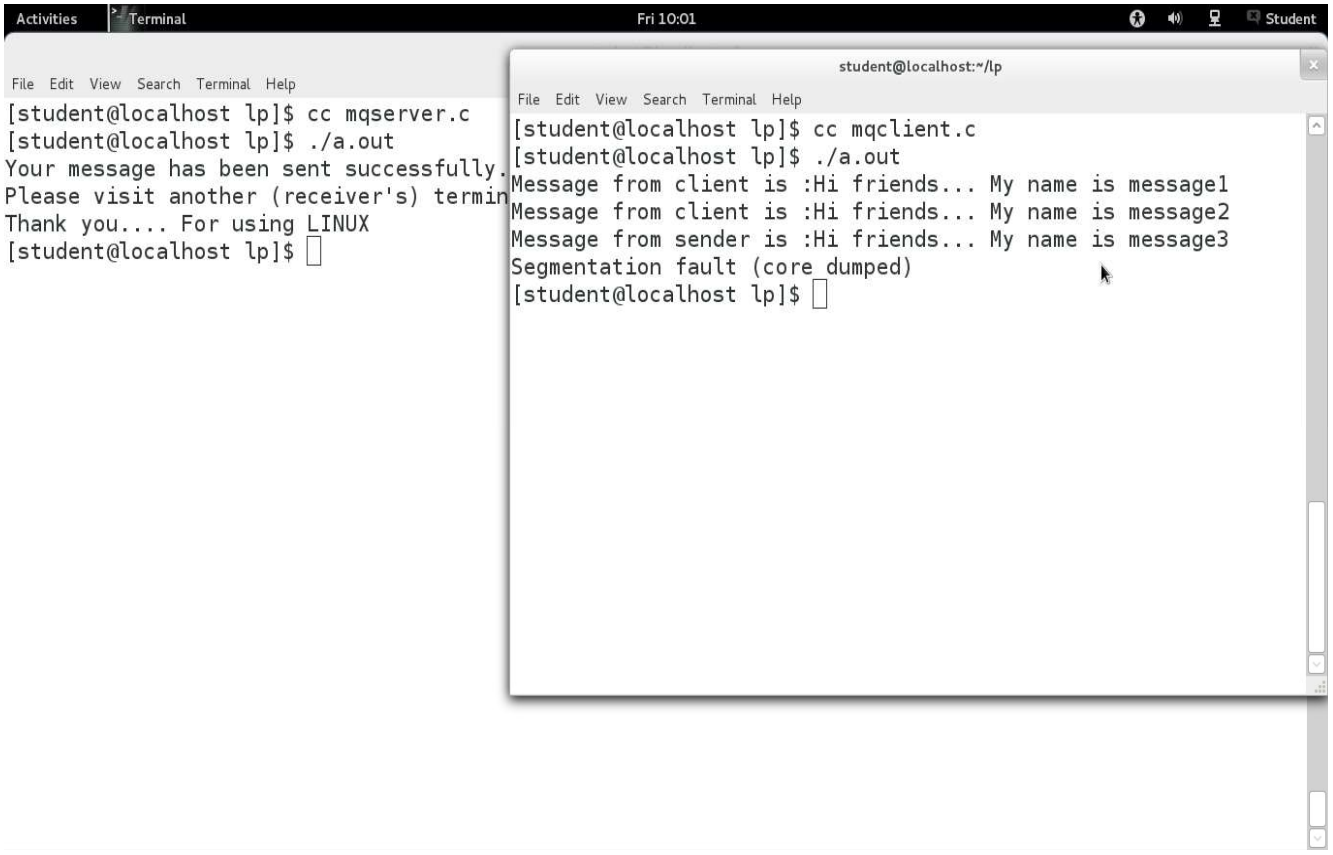


```
[student@localhost lp]$ cc mqserver.c
[student@localhost lp]$ ./a.out
Your message has been sent successfully...
Please visit another (receiver's) terminal...
Thank you.... For using LINUX
[student@localhost lp]$ 
```

mqclient.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
#define SIZE 40
void main()
{
int mfd,mfd2,mfd3;
struct
{
long mtype;
char mtext[6];
}s1,s2,s3;
if((mfd=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd,&s1,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from client is :%s\n",s1.mtext);
if((mfd2=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd2,&s2,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from client is :%s\n",s2.mtext);
if((mfd3=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd3,&s3,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from sender is :%s\n",s3.mtext);
}
```

OUTPUT:


Activities
Terminal
Fri 10:01
Student
File Edit View Search Terminal Help
student@localhost:~/lp
File Edit View Search Terminal Help

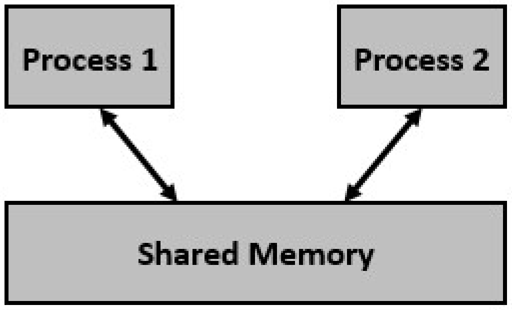
```
[student@localhost lp]$ cc mqserver.c
[student@localhost lp]$ ./a.out
Your message has been sent successfully.
Please visit another (receiver's) termin
Thank you.... For using LINUX
[student@localhost lp]$ 
```

```
[student@localhost lp]$ cc mqclient.c
[student@localhost lp]$ ./a.out
Message from client is :Hi friends... My name is message1
Message from client is :Hi friends... My name is message2
Message from sender is :Hi friends... My name is message3
Segmentation fault (core dumped)
[student@localhost lp]$ 
```

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 anther 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.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.



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, let us see this –

System Calls for Shared Memory:

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

Let us look at a few details of the system calls related to shared memory.

1. **shmget ( )** - allocates a shared memory segment

Synopsis:

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

#include <sys/shm.h>

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

Description:

- The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok( ). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communication. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.
- The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.
- The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Return Value: A valid segment identifier, *shmid*, is returned on success, -1 on error.

Errors:

Tag	Description
EACCES	The user does not have permission to access the shared memory segment, and does not have the**CAP_IPC_OWNER** capability.
EEXIST	**IPC_CREAT | IPC_EXCL** was specified and the segment exists.

2. **shmat ( )** - shared memory attach operation

Syntax:

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

#include <sys/shm.h>

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

Description:

- **The first argument, shmid,** is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget( ) system call.
- **The second argument, shmaddr,** is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. 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 Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.
- **The third argument, shmflg,** specifies the required shared memory flag/s such as 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).

Return Value:

On success **shmat**( ) returns the address of the attached shared memory segment; on -1 is returned, and errno is set to indicate the cause of the error.

Errors:

When **shmat**( ) fails, *errno* is set to one of the following:

EACCES:

The calling process does not have the required permissions for the requested attach type, and does not have the **CAP_IPC_OWNER** capability.

EIDRM: *shmid* points to a removed identifier.

3. **shmdt ( )** - shared memory operations

Syntax:

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

#include <sys/shm.h>

int shmdt(const void *shmaddr)
```

Description:

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

Return value:

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror( ) function.

4. **Shmctl ( ) -**shared memory control

Synopsis:

```
#include <sys/ipc.h>
 #include <sys/shm.h>

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

Description:

- ❖ The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget( ) system call.
- ❖ 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_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

- ❖ The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

Return value:

- ❖ This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure

Shared memory example

Writer.c

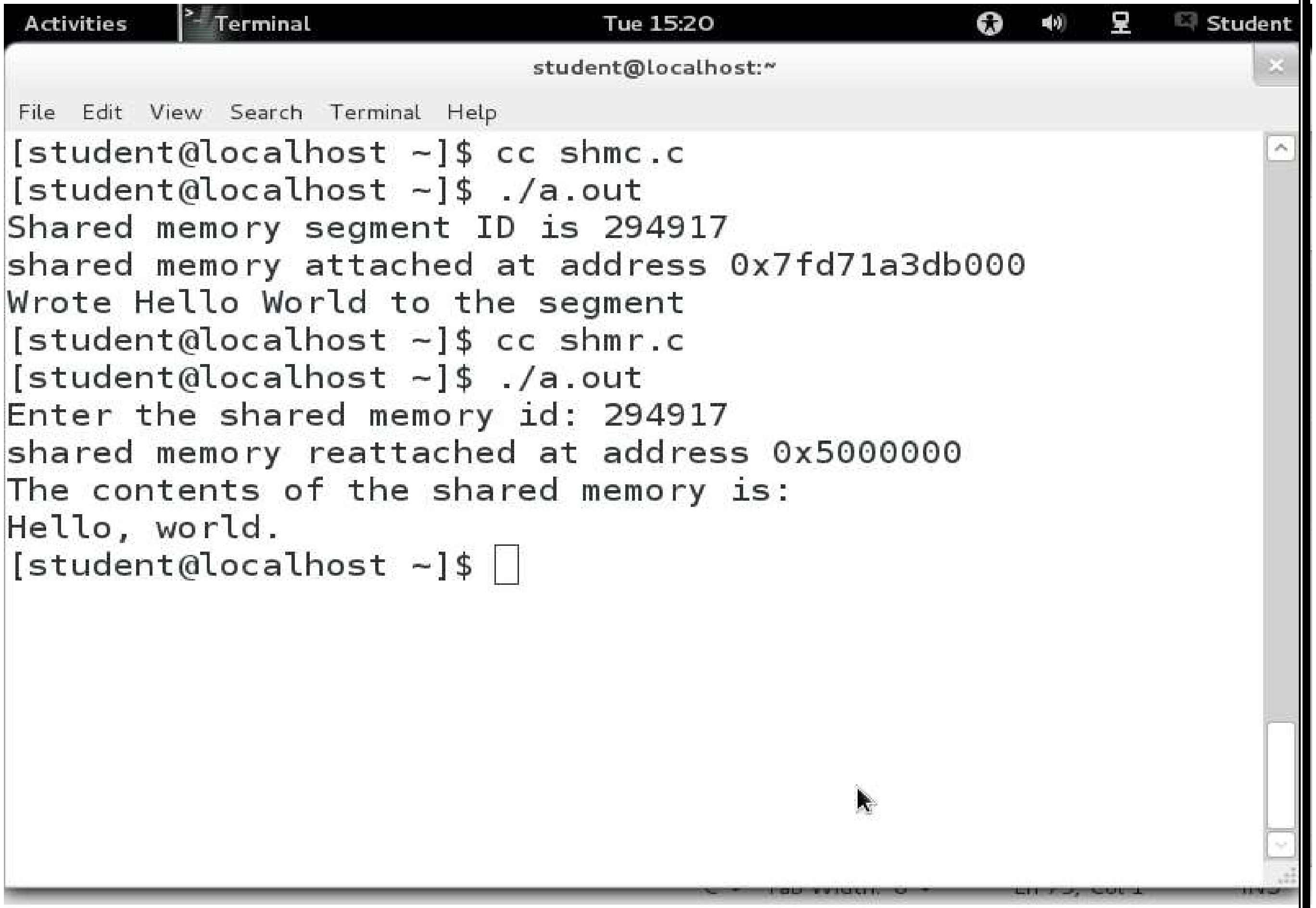
```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ( )
{
  int segment_id;
  char bogus;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;

  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size, IPC_CREAT | IPC_EXCL |
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
  /* Attach the shared memory segment.  */
  printf("Shared memory segment ID is %d\n", segment_id);
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size. */
  /*
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size  =               shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  */
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);
  printf("Wrote Hello World to the segment\n");


}
```

Reader.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
  int segment_id;
  char bogus;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;
  printf("Enter the shared memory id: ");
  scanf("%d", &segment_id);
```

```
 /* Reattach the shared memory segment, at a different address.  */

 shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);

 printf ("shared memory reattached at address %p\n", shared_memory);

 /* Print out the string from shared memory.  */

 printf ("The contents of the shared memory is:\n%s\n", shared_memory);

 /* Detach the shared memory segment.  */

 shmdt (shared_memory);

 return 0;

}
```

OUTPUT:



```
[student@localhost ~]$ cc shmc.c
[student@localhost ~]$ ./a.out
Shared memory segment ID is 294917
shared memory attached at address 0x7fd71a3db000
Wrote Hello World to the segment
[student@localhost ~]$ cc shmr.c
[student@localhost ~]$ ./a.out
Enter the shared memory id: 294917
shared memory reattached at address 0x5000000
The contents of the shared memory is:
Hello, world.
[student@localhost ~]$ 
```