**LAB - 2**

**IPC COMMUNICATION**

**Objective:**

To establish client server communication using Inter Process Communication(IPC) mechanisms FIFO, Message queues and Shared memory.

**Introduction:**

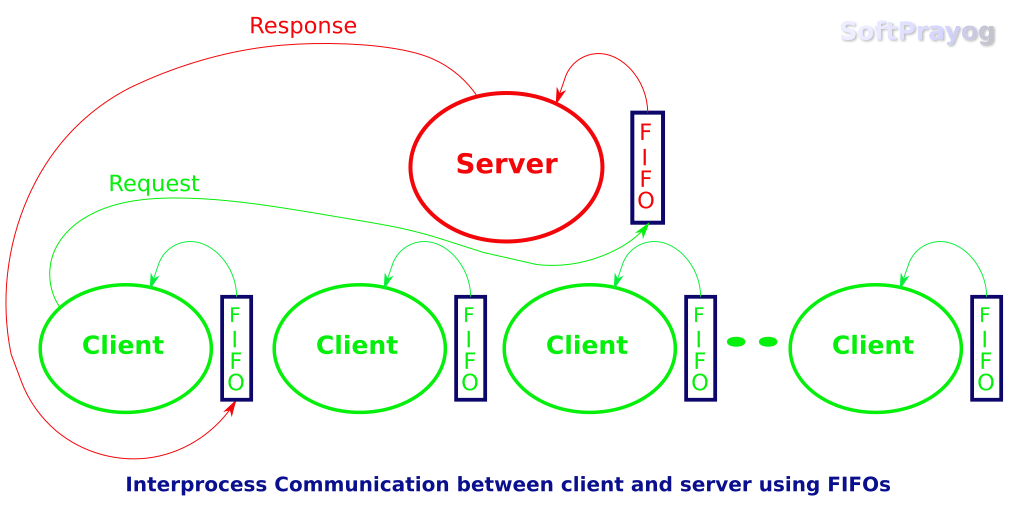
In user space each process have its own separated address space. Suppose if any process wants to send any information or data to the another process which is running in different address space then we need these IPC techniques to communicate among the processes .

**FIFO:**

FIFOs are pipes with a name and are also commonly referred to as named pipes. Pipes are common on Linux command lines but do not have a system-wide name. So, any two processes that wish to communicate using a pipe need to be related, either parent and child or, sharing a common parent, who sets up the pipe and passes its file descriptors to individual processes. A FIFO is different because it has a name like a file. Also, since it exists in the filesystem, it is not necessary for the processes to be related to communicate using the FIFO.

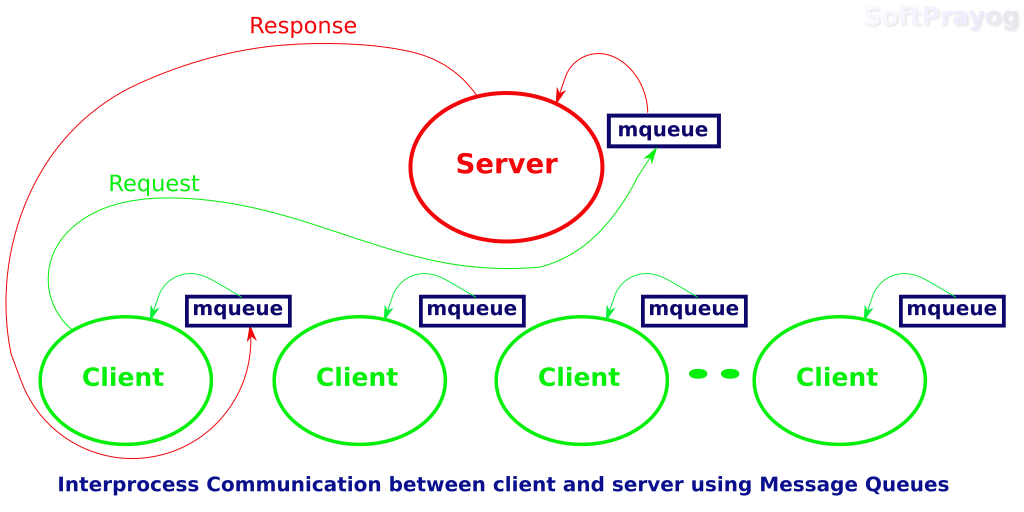
The client-server paradigm comprises of a single server process, which works all the time, receives requests from clients and gives them responses. A client is the process that manages the inputs and outputs for a live user. Clients come and go but the server works all the time. The clients communicate with the server using an interprocess communication mechanism. Each process in the paradigm has a system-wide mechanism for receiving messages. In the example in a later section, we will use the FIFO as the mechanism for receiving messages. That is, the server will have a FIFO, where clients can put messages for the server. Similarly, each client will have a FIFO, where the server can put in messages for that client.

A FIFO can be opened using the open system call. After open, we can use the read and write system calls for reading from and writing to the FIFO, using the file descriptor returned by open. Of course, as per our design, we will either read or write to a FIFO but not do both. A process, be it a client or the server, reads from its own FIFO for receiving data and writes on other process's FIFO for sending data to that process.



### **POSIX Message queues:**

System V message queues are identified using keys obtained with the ftok function call. POSIX message queues are identified using name strings. On Linux, POSIX queues are named as string starting with a forward slash (/) followed by one or more characters, none of which is a slash and ending with the null character. Any process knowing the queue name and having appropriate permissions can send or receive messages from the queue and also do other operations on it.



The example below demonstrates interprocess communication between a server and clients using POSIX message queues in Linux. The server manages token numbers, which could be seat numbers for a flight, or something similar. It is server's job to give a token number to a client on request. In a typical scenario, there might be multiple clients requesting the server for token numbers. The server's message queue name is known to clients. Each client has its own message queue, in which server posts responses. When a client sends a request, it sends its message queue name. The server opens client's message queue and sends its response. The client picks up the response from its message queue and reads the token number in it. The process architecture looks like this.

**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.

**Shared Memory:**

IPC 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,int shmflg); upon successful completion, shmget() returns an identifier for the shared memory segment.

**shmat():** Before you can use a shared memory segment, you have to attach yourself

to it using shmat(). void \*shmat(int shmid ,void \*shmaddr ,int shmflg);

shmid is shared memory id. shmaddr specifies specific address to use but we should set

it to zero and OS will automatically choose the address.

**shmdt():** When you’re done with the shared memory segment, your program should

detach itself from it using shmdt(). int shmdt(void \*shmaddr);

**shmctl():** when you detach from shared memory,it is not destroyed. So, to destroy

**shmctl()** is used. shmctl(int shmid,IPC\_RMID,NULL);

Signature