1.Introduction

In computer science, inter-process communication or interprocess communication (IPC) refers specifically to the mechanisms an operating system that allows the exchange of data between processes.By providing a user with a set of programming interfaces, IPC helps a programmer organize the activities among different processes. IPC allows one application to control another application, thereby enabling data sharing without interference. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests.[1] Many applications are both clients and servers, as commonly seen in distributed computing. Methods for doing IPC are divided into categories which vary based on software requirements, such as performance and modularity requirements, and system circumstances, such as network bandwidth and latency.

IPC enables data communication by allowing processes to use segments, semaphores, and other methods to share memory and information. IPC facilitates efficient message transfer between processes. The idea of IPC is based on Task Control Architecture (TCA). It is a flexible technique that can send and receive variable length arrays, data structures, and lists. It has the capability of using publish/subscribe and client/server data-transfer paradigms while supporting a wide range of operating systems and languages.

IPC is very important to the design process for microkernels and nanokernels. Microkernels reduce the number of functionalities provided by the kernel. Those functionalities are then obtained by communicating with servers via IPC, increasing drastically the number of IPC compared to a regular monolithic kernel.

2. Interprocess communication methods

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms. Signals and pipes are two of them but Linux also supports the System V IPC mechanisms named after the Unix release in which they first appeared.
The IPC mechanism can be classified into pipes, first in, first out (FIFO), and shared memory. Pipes were introduced in the UNIX operating system. In this mechanism, the data flow is unidirectional. A pipe can be imagined as a hose pipe in which the data enters through one end and flows out from the other end. A pipe is generally created by invoking the pipe system call, which in turn generates a pair of file descriptors. Descriptors are usually created to point to a pipe node. One of the main features of pipes is that the data flowing through a pipe is transient, which means data can be read from the read descriptor only once. If the data is written into the write descriptor, the data can be read only in the order in which the data was written. The working principle of FIFO is very similar to that of pipes. The data flow in FIFO is unidirectional and is identified by access points. The difference between the two is that FIFO is

identified by an access point, which is a file within the file system, whereas pipes are identified by an access point.

2.1 Signals

Signals are one of the oldest inter-process communication methods used by Unix ™ systems. They are used to signal asynchronous events to one or more processes. A signal could be generated by a keyboard interrupt or an error condition such as the process attempting to access a non-existent location in its virtual memory. Signals are also used by the shells to signal job control commands to their child processes.
There are a set of defined signals that the kernel can generate or that can be generated by other processes in the system, provided that they have the correct privileges. You can list a system's set of signals using the kill command (kill -l), on my Intel Linux box this gives:

- SIGHUP
- SIGINT
- SIGQUIT
- SIGILL
- SIGTRAP
- SIGIOT
- SIGBUS
- SIGFPE
- SIGKILL
- SIGUSR1
- SIGSEGV
- SIGUSR2
- SIGPIPE
- SIGALRM
- SIGTERM
- SIGCHLD
- SIGCONT
- SIGSTOP
- SIGTSTP
- SIGTTIN
- SIGTTOU
- SIGURG
- SIGXCPU
- SIGXFSZ
- SIGVTALRM
- SIGPROF
- SIGWINCH
- SIGIO
- SIGPWR

The numbers are different for an Alpha AXP Linux box. Processes can choose to ignore most of the signals that are generated, with two notable exceptions: neither the SIGSTOP signal which causes a process to halt its execution nor the SIGKILL signal which causes a process to exit can be ignored. Otherwise though, a process can choose just how it wants to handle the various signals. Processes can block the signals and, if they do not block them, they can either choose to handle them themselves or allow the kernel to handle them. If the kernel handles the signals, it will do the default actions required for this signal. For example, the default action when a process receives the SIGFPE (floating point exception) signal is to core dump and then exit. Signals have no inherent relative priorities. If two signals are generated for a process at the same time then they may be presented to the process or handled in any order. Also there is no

mechanism for handling multiple signals of the same kind. There is no way that a process can tell if it received 1 or 42 SIGCONT signals.

Linux implements signals using information stored in the task_struct for the process. The number of supported signals is limited to the word size of the processor. Processes with a word size of 32 bits can have 32 signals whereas 64 bit processors like the Alpha AXP may have up to 64 signals. The currently pending signals are kept in the signal field with a mask of blocked signals held in blocked. With the exception of SIGSTOP and SIGKILL, all signals can be blocked. If a blocked signal is generated, it remains pending until it is unblocked. Linux also holds information about how each process handles every possible signal and this is held in an array of sigaction data structures pointed at by the task_struct for each process. Amongst other things it contains either the address of a routine that will handle the signal or a flag which tells Linux that the process either wishes to ignore this signal or let the kernel handle the signal for it. The process modifies the default signal handling by making system calls and these calls alter the sigaction for the appropriate signal as well as the blocked mask.

Not every process in the system can send signals to every other process, the kernel can and super users can. Normal processes can only send signals to processes with the same *uid* and *gid* or to processes in the same process group[1]. Signals are generated by setting the appropriate bit in the task_struct's signal field. If the process has not blocked the signal and is waiting but interruptible (in state Interruptible) then it is woken up by changing its state to Running and making sure that it is in the run queue. That way the scheduler will consider it a candidate for running when the system next schedules. If the default handling is needed, then Linux can optimize the handling of the signal. For example, if the signal SIGWINCH (the X window changed focus) and the default handler is being used then there is nothing to be done.

Signals are not presented to the process immediately they are generated., they must wait until the process is running again. Every time a process exits from a system call its signal and blocked fields are checked and, if there are any unblocked signals, they can now be delivered. This might seem a very unreliable method but every process in the system is making system calls, for example to write a character to the terminal, all of the time. Processes can elect to wait for signals if they wish, they are suspended in state Interruptible until a signal is presented. The Linux signal processing code looks at the sigaction structure for each of the current unblocked signals.

If a signal's handler is set to the default action then the kernel will handle it. The SIGSTOP signal's default handler will change the current process's state to Stopped and then run the scheduler to select a new process to run. The default action for the SIGFPE signal will core dump the process and then cause it to exit. Alternatively, the process may have specified its own signal handler. This is a routine which will be called whenever the signal is generated and the sigaction structure holds the address of this routine. The kernel must call the process's signal handling routine and how this happens is processor specific but all CPUs must cope with the fact that the current process is running in kernel mode and is just about to return to the process that called the kernel or system routine in user mode. The problem is solved by manipulating the stack and registers of the process. The process's program counter is set to the address of its signal

handling routine and the parameters to the routine are added to the call frame or passed in registers. When the process resumes operation it appears as if the signal handling routine were called normally.
Linux is POSIX compatible and so the process can specify which signals are blocked when a particular signal handling routine is called. This means changing the blocked mask during the call to the processes signal handler. The blocked mask must be returned to its original value when the signal handling routine has finished. Therefore Linux adds a call to a tidy up routine which will restore the original blocked mask onto the call stack of the signalled process. Linux also optimizes the case where several signal handling routines need to be called by stacking them so that each time one handling routine exits, the next one is called until the tidy up routine is called.

2.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. These are inter-process communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.
The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.
Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.
Figure 2.1 demonstrates pipes are given as follows:

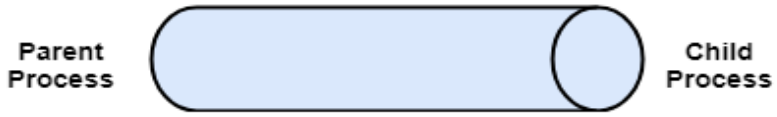


FIG 2.1 A TYPICAL PIPE

Pipes then are unidirectional byte streams which connect the standard output from one process into the standard input of another process. Neither process is aware of this redirection and behaves just as it would normally. It is the shell which sets up these temporary pipes between the processes.

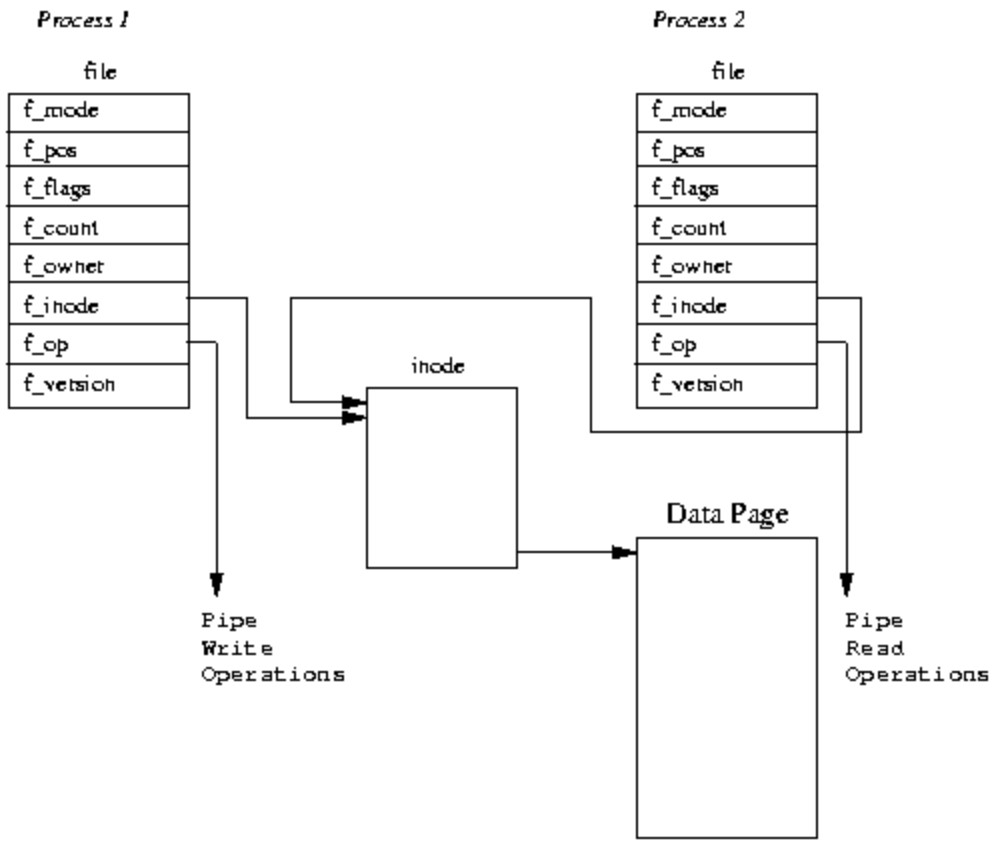


FIG 2.2 PIPES

From the above figure 2.2, we conclude that, In Linux, a pipe is implemented using two file data structures which both point at the same temporary VFS inode which itself points at a physical page within memory. Figure 5.1 shows that each file data structure contains pointers to different file operation routine vectors; one for writing to the pipe, the other for reading from the pipe.
This hides the underlying differences from the generic system calls which read and write to ordinary files. As the writing process writes to the pipe, bytes are copied into the shared data page and when the reading process reads from the pipe, bytes are copied from the shared data page. Linux must synchronize access to the pipe. It must make sure that the reader and the writer of the pipe are in step and to do this it uses locks, wait queues and signals.
When the writer wants to write to the pipe it uses the standard write library functions. These all pass file descriptors that are indices into the process's set of file data structures, each one representing an open file or, as in this case, an open pipe. The Linux system call uses the write routine pointed at by the file data structure describing this pipe. That write routine uses information held in the VFS inode representing the pipe to manage the write request.

If there is enough room to write all of the bytes into the pipe and, so long as the pipe is not locked by its reader, Linux locks it for the writer and copies the bytes to be written from the process's address space into the shared data page. If the pipe is locked by the reader or if there is not enough room for the data then the current process is made to sleep on the pipe inode's wait queue and the scheduler is called so that another process can run. It is interruptible, so it can receive signals and it will be woken by the reader when there is enough room for the write data or when the pipe is unlocked. When the data has been written, the pipe's VFS inode is unlocked and any waiting readers sleeping on the inode's wait queue will themselves be woken up.
Reading data from the pipe is a very similar process to writing to it.
Processes are allowed to do non-blocking reads (it depends on the mode in which they opened the file or pipe) and, in this case, if there is no data to be read or if the pipe is locked, an error will be returned. This means that the process can continue to run. The alternative is to wait on the pipe inode's wait queue until the write process has finished. When both processes have finished with the pipe, the pipe inode is discarded along with the shared data page.
Pipes have two limitations:
Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

Linux also supports *named* pipes, also known as FIFOs because pipes operate on a First In First Out principle. FIFOs get around the second limitation, and that UNIX domain sockets get around both limitations. The first data written into the pipe is the first data read from the pipe. Unlike pipes, FIFOs are not temporary objects, they are entities in the file system and can be created using the mkfifo command. Processes are free to use a FIFO so long as they have appropriate access rights to it. The way that FIFOs are opened is a little different from pipes. A pipe (its two file data structures, its VFS inode and the shared data page) is created in one go whereas a FIFO already exists and is opened and closed by its users. Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it. That aside, FIFOs are handled almost exactly the same way as pipes and they use the same data structures and operations.

A pipe is created by calling the pipe function.

```
#include <unistd.h>
int pipe(int fd[2]);
/* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

Two ways to picture a half-duplex pipe are shown in the figure 2.3 below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

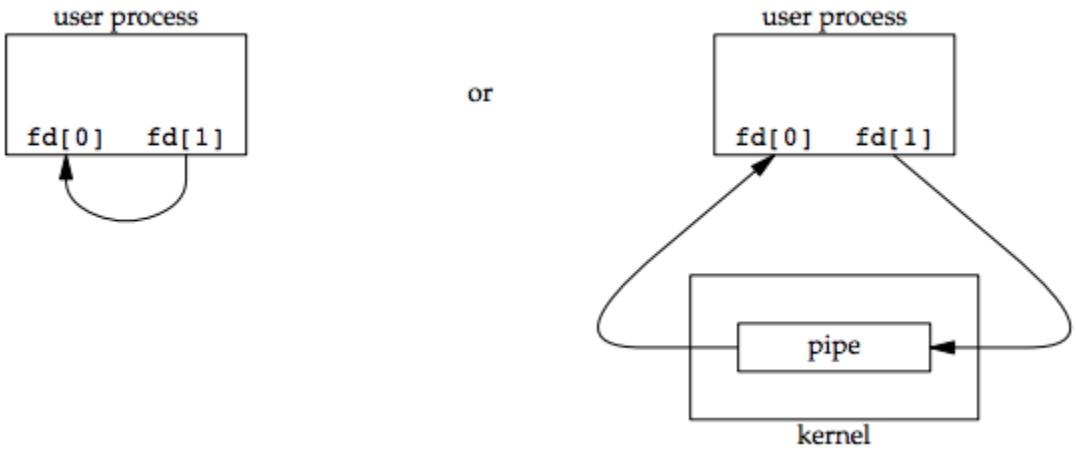


FIG 2.3 DATA FLOW IN A PIPE

The fstat function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the S_ISFIFO macro.
POSIX.1 states that the st_size member of the stat structure is undefined for pipes. But when the fstat function is applied to the file descriptor for the read end of the pipe, many systems store in st_size the number of bytes available for reading in the pipe, which is nonportable.
A pipe in a single process is next to useless. Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa. The following figure 2.4 shows this scenario:

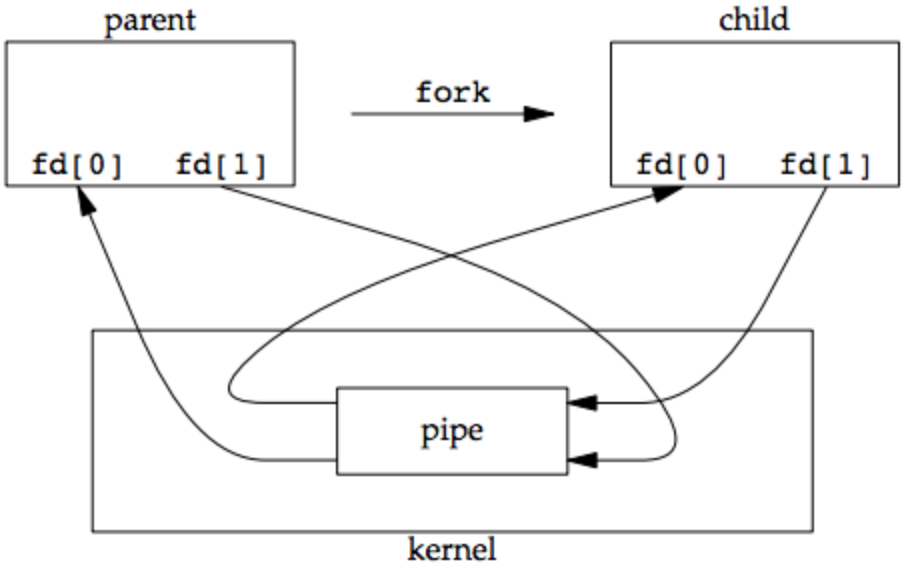


FIG 2.4 A TYPICAL PIPE IN A FORK

What happens after the fork depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure 2.5 shows the resulting arrangement of descriptors.

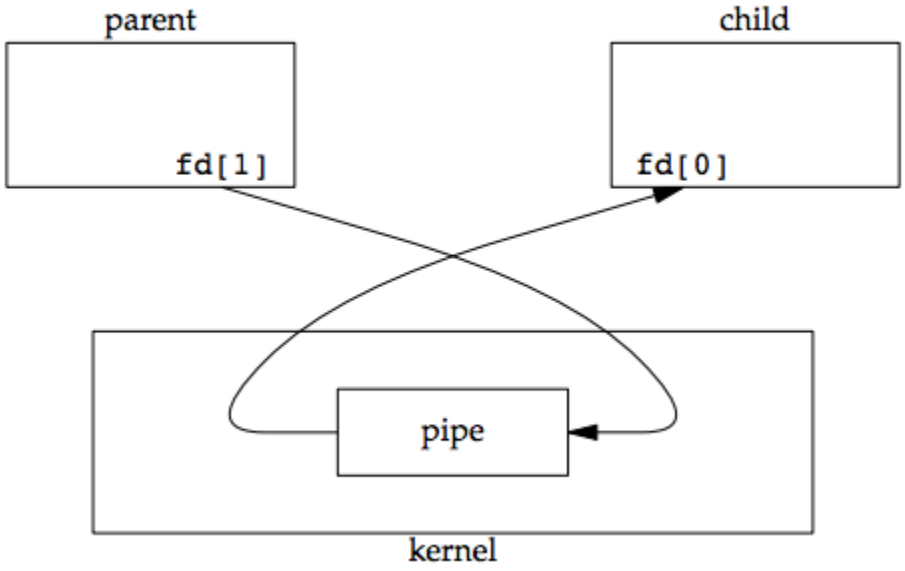


FIG 2.5 A TYPICAL PIPE IN FORK REARRANGED

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].
When one end of a pipe is closed, two rules apply:
If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)
If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.
When we're writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size. A write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers. We can determine the value of PIPE_BUF by using pathconf or fpathconf.

3. The Client-Server Paradigm

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 as shown in figure 3.1.

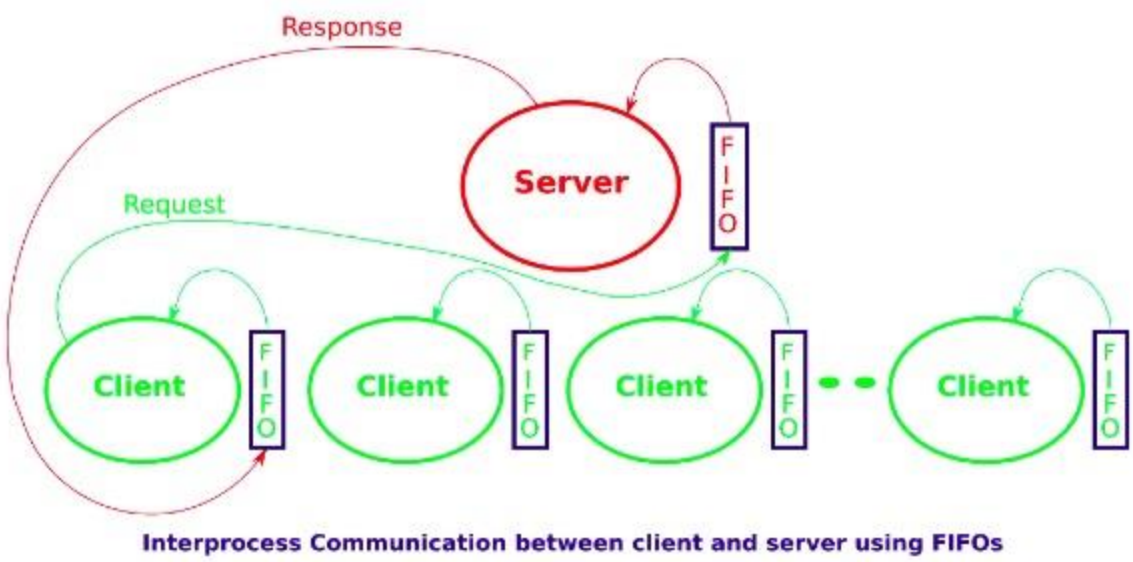


FIG 3.1 Inter Process Communication

3.1. mkfifo system call

We can create a FIFO from a program using the mkfifo system call.

```
#include <sys/stat.h>
int mkfifo (const char *path, mode_t perms);
// Returns -1 on error.
```

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.
In the program, we have created three fifo files namely res.fifo, req.fifo, choice.fifo. The fifo file res.fifo stores the response sent by the server, req.fifo stores the request sent by the client and choice.fifo stores the choice selected by the client. All the three fifo files are opened in 0777 mode which include read, write, & execute for owner, group and others permissions. The server process read the choice from choice.fifo and executes the appropriate instructions. If choice is 1, the server would read the clients message from req.fifo and writes its response in res.fifo. If the choice is 2 then the server searches for the requested file and displays the content. If the file is not found, then the server would display the error message stating the absence of the file. If the

choice is 0, then both the client and the server process exits. The functions read and write are used to read and write the data from the files.

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the filesystem by calling **mkfifo**(). Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

4. PROGRAMS

4.1. FILE SEARCH:

4.1.1 USING TCP/IP SOCKETS:

Client:

```
#include <stdio.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
   int soc, n;
   char buffer[1024], fname[50];
   struct sockaddr_in addr;

   soc = socket(PF_INET, SOCK_STREAM, 0);

   addr.sin_family = AF_INET;
   addr.sin_port = htons(7891);
   addr.sin_addr.s_addr = inet_addr("127.0.0.1");

   while(connect(soc, (struct sockaddr *) &addr, sizeof(addr))) ;
   printf("\nClient is connected to Server");
   printf("\nEnter file name: ");
   scanf("%s", fname);
   send(soc, fname, sizeof(fname), 0);

   printf("\nRecieved response\n");
   /*  keep printing any data received from the server */
   while ((n = recv(soc, buffer, sizeof(buffer), 0)) > 0)
      printf("%s", buffer);

   return 0;
}
```

Server:

```
#include <stdio.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{

int welcome, new_soc, fd, n;
char buffer[1024], fname[50];

struct sockaddr_in addr;
welcome = socket(PF_INET, SOCK_STREAM, 0);
addr.sin_family = AF_INET;
addr.sin_port = htons(7891);
addr.sin_addr.s_addr = inet_addr("127.0.0.1");

bind(welcome, (struct sockaddr *) &addr, sizeof(addr));
printf("\nServer is Online");
listen(welcome, 5);
new_soc = accept(welcome, NULL, NULL);
recv(new_soc, fname, 50, 0);
printf("\nRequesting for file: %s\n", fname);
fd = open(fname, O_RDONLY);
if (fd < 0)
send(new_soc, "\nFile not found\n", 15, 0);
else
while ((n = read(fd, buffer, sizeof(buffer))) > 0)
send(new_soc, buffer, n, 0);
printf("\nRequest sent\n");

close(fd);
return 0;

}
```

4.1.2. USING FIFO FILES:

Client:

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

int main()
{
  char fname[50], buffer[1025],message[80],downlod[35];
  int req, res, n,m,k,flag=1,opt,u,v,choice;
  char ch[2];
  req = open("req.fifo", O_WRONLY);
  res = open("res.fifo", O_RDONLY);
  if(req < 0 || res < 0)
    {
       printf("Please Start the server first\n");
       exit(-1);
    }
  printf("FILE SEARCH\n");
  printf("Enter filename to request:\n");
  scanf("%s", fname);
  write(req, fname, sizeof(fname));
  printf("Received response\n");
  while((n = read(res, buffer, sizeof(buffer)))>0)
 {
    write(1, buffer, n);
 }
   close(choice);
   close(req);
   return 0;
}
```

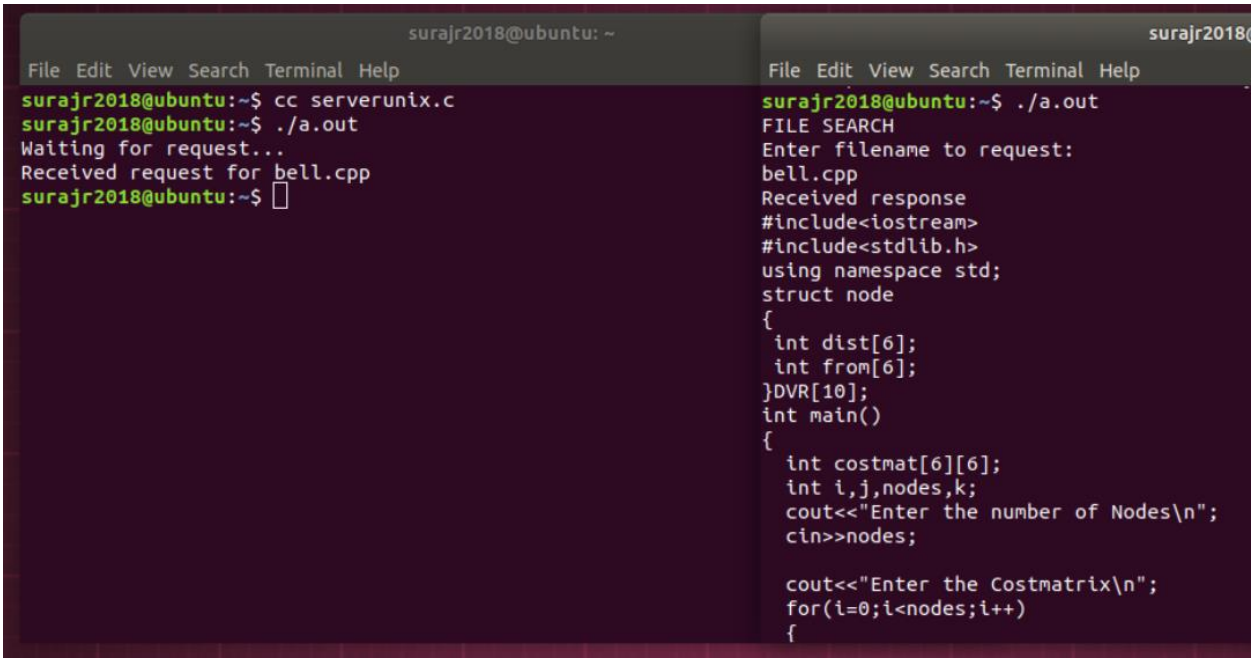
Server:

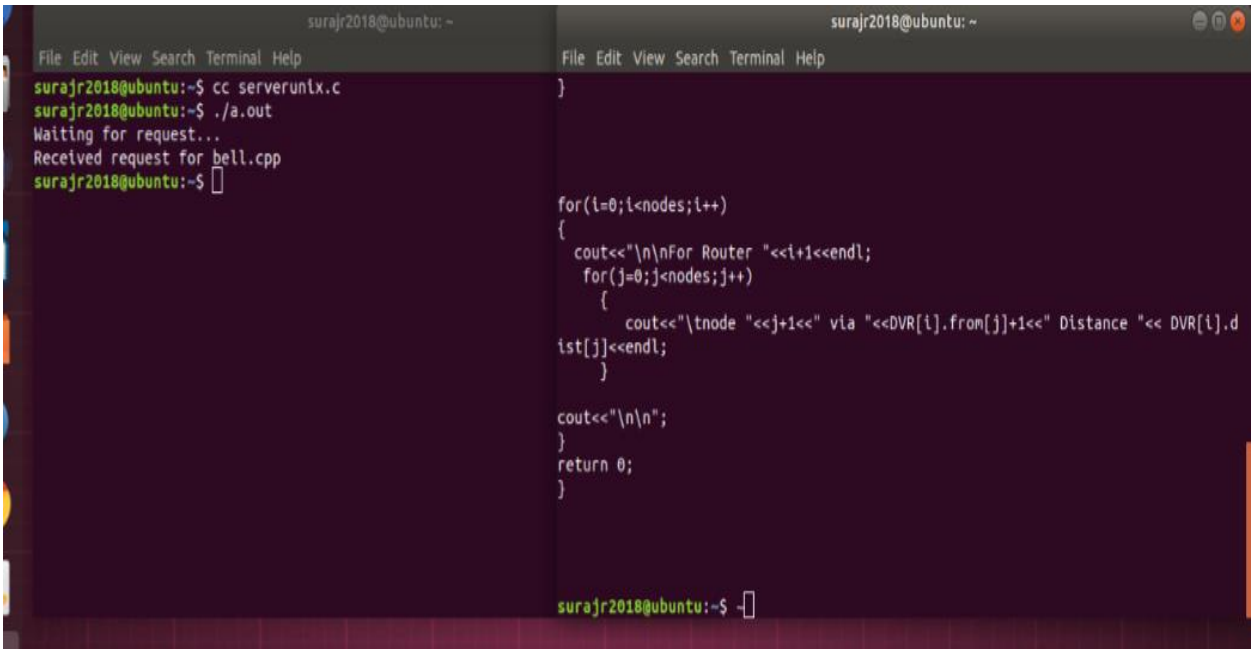
```
#include <stdio.h>
```

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include<string.h>
int main()
{
   char fname[50], buffer[1025],ch[2],chatbuff[1024],servbuff[1024];
   int req, res, n, file,choi,flag=1,m,u;
   mkfifo("req.fifo", 0777);
   mkfifo("res.fifo", 0777);
   mkfifo("choice.fifo",0777);
   req = open("req.fifo", O_RDONLY);
   res = open("res.fifo", O_WRONLY);
   printf("Waiting for request...\n");
   read(req, fname, sizeof(fname));
   printf("Received request for %s\n", fname);
   file = open(fname, O_RDONLY);
   if (file < 0)
      write(res, "File not found\n", 15);
   else {
      while((n = read(file, buffer, sizeof(buffer))) > 0) {
         write(res, buffer, n);
      }
   }
   close(choice);
   close(req)
  close(res);
   unlink("req.fifo");
   unlink("res.fifo");
return 0;
}
```

4.1.3. OUTPUT:

For compilation, first compile the server program and then compile the client program in two different terminals .





4.2. CHAT PROGRAM:

4.2.1. USER 1:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define FILLED 0
#define Ready 1
#define NotReady -1

struct memory {
   char buff[100];
   int status, pid1, pid2;
}; struct memory* shmptr;

void handler(int signum)
{
  if (signum == SIGUSR1) {
      printf("Received User2: ");
      puts(shmptr->buff);
   }
}

int main()
{
 int pid = getpid();
 int shmid;
 int key = 12345;
 shmid = shmget(key, sizeof(struct memory), IPC_CREAT | 0666);
 shmptr = (struct memory*)shmat(shmid, NULL, 0);
 shmptr->pid1 = pid;
 shmptr->status = NotReady;
 signal(SIGUSR1, handler);
```

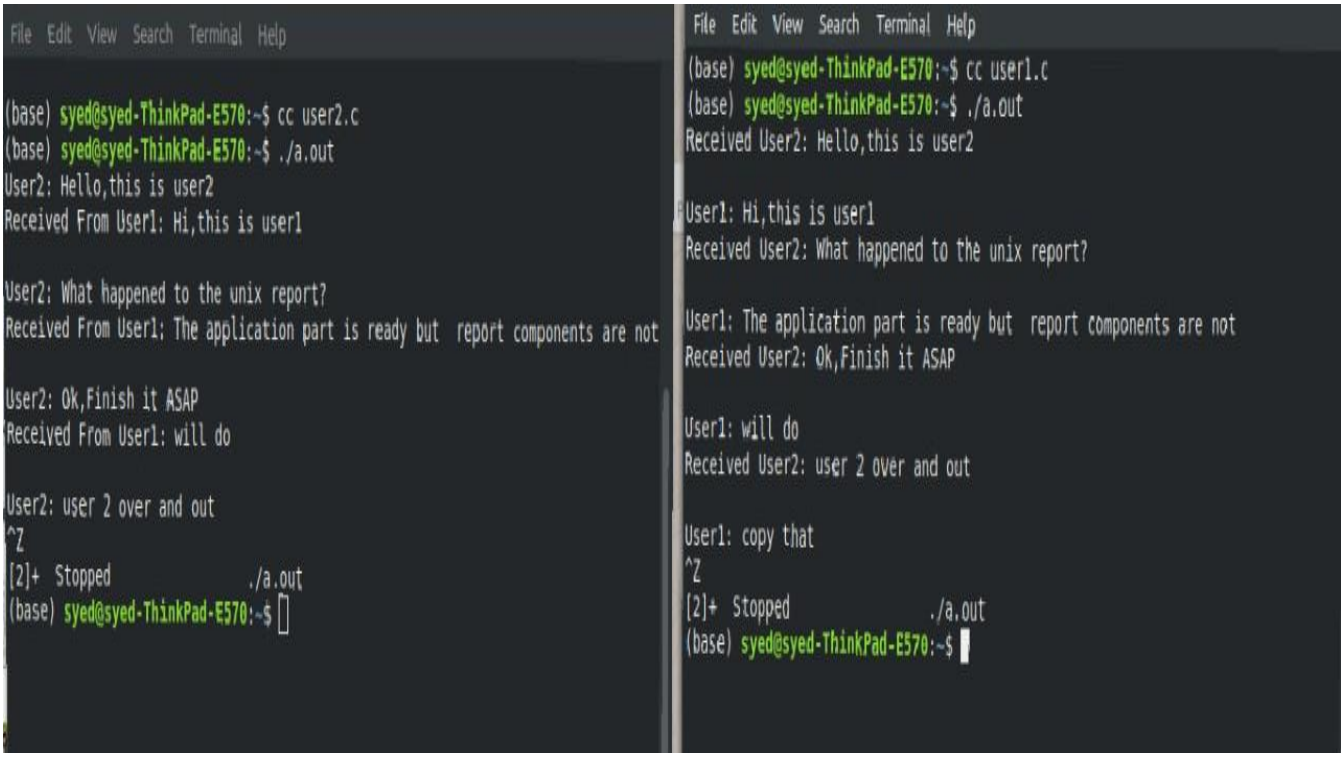
```
  while (1) {
      while (shmptr->status != Ready)
      continue;
      sleep(1);
      printf("User1: ");
      fgets(shmptr->buff, 100, stdin);
  shmptr->status = FILLED;
  kill(shmptr->pid2, SIGUSR2);
  }
shmdt((void*)shmptr);
shmctl(shmid, IPC_RMID, NULL);
return 0;
}
```

4.2.2. USER 2:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define FILLED 0
#define Ready 1
#define NotReady -1

struct memory {
    char buff[100];
    int status, pid1, pid2;
}; struct memory* shmptr;

void handler(int signum)
{
   if (signum == SIGUSR2) {
        printf("Received From User1: ");
        puts(shmptr->buff);
    }
}
```

```
int main()
{
  int pid = getpid();
  int shmid;
  int key = 12345;
  shmid = shmget(key, sizeof(struct memory), IPC_CREAT | 0666);
  shmptr = (struct memory*)shmat(shmid, NULL, 0);
  shmptr->pid2 = pid;
  shmptr->status = NotReady;
  signal(SIGUSR2, handler);
  while (1) {
        sleep(1);
        printf("User2: ");
        fgets(shmptr->buff, 100, stdin);
        shmptr->status = Ready;
        kill(shmptr->pid1, SIGUSR1);
  while (shmptr->status == Ready)
            continue;
    }
  shmdt((void*)shmptr);
    return 0;
}
```

4.2.3. OUTPUT:


File Edit View Search Terminal Help
(base) syed@syed-ThinkPad-E570:~$ cc user2.c
(base) syed@syed-ThinkPad-E570:~$ ./a.out
User2: Hello,this is user2
Received From User1: Hi,this is user1
User2: What happened to the unix report?
Received From User1: The application part is ready but report components are not
User2: Ok,Finish it ASAP
Received From User1: will do
User2: user 2 over and out
^Z
[2]+ Stopped ./a.out
(base) syed@syed-ThinkPad-E570:~$
File Edit View Search Terminal Help
(base) syed@syed-ThinkPad-E570:~$ cc user1.c
(base) syed@syed-ThinkPad-E570:~$ ./a.out
Received User2: Hello,this is user2
User1: Hi,this is user1
Received User2: What happened to the unix report?
User1: The application part is ready but report components are not
Received User2: Ok,Finish it ASAP
User1: will do
Received User2: user 2 over and out
User1: copy that
^Z
[2]+ Stopped ./a.out
(base) syed@syed-ThinkPad-E570:~$

5. Conclusion :

Hence we successfully demonstrated the working mechanism of the interprocess communication using the fifo files,TCP/IP sockets and a chatting process using client server paradigm.

6. References:

1.Richard stevens, Stephen A Rago, Third edition, Advanced Programming in the UNIX Environment
2.Rochkind, Second Edition, Advanced UNIX Programming
3. https://www.tldp.org/LDP/tlk/ipc/ipc.html
4. https://notes.shichao.io/apue/ch15/#popen-and-pclose-functions
5. https://www.techopedia.com/definition/3818/inter-process-communication-ipc
6. https://www.softprayog.in/programming/interprocess-communication-using-fifos-in-linux