**Experiment Number: 07**

**TITLE:** Inter-process Communication(IPC) in Linux using:

a. **Pipes:** Full duplex communication between parent and child processes. Parent process writes a pathname of a file (the contents of the file are desired) on one pipe to be read by child process and child process writes the contents of the file on second pipe to be read by parent process and displays on standard output.

b. **FIFOs:** Full duplex communication between two independent processes. First process accepts sentences and writes on one pipe to be read by second process and second process counts number of characters, number of words and number of lines in accepted sentences, writes this output in a text file and writes the contents of the file on second pipe to be read by first process and displays on standard output.

# OBJECTIVE:

1. To study use of pipes, FIFO’s.
2. To study inter-process communication in Linux.

**THEORY:**

**The Pipe Call**

The lower-level pipe( ) function provides a means of passing data between 2 two programs, without the overhead of invoking a shell to interpret the requested command. It also gives us more control over the reading and writing of data.

The pipe function has the following prototype:

**#include <unistd.h>**

**int pipe(int file\_descriptor[2]);**

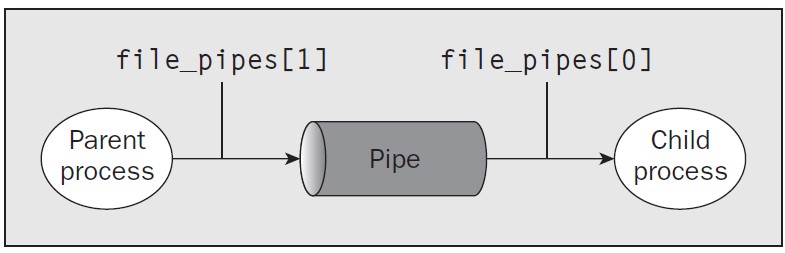
It is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors and returns a zero. On failure, it returns -1 and sets errno to indicate the reason for failure. Errors defined in the Linux manual page for pipe (in section 2 of the manual) are

❑ EMFILE: Too many file descriptors are in use by the process.

❑ ENFILE: The system file table is full.

❑ DEFAULT: The file descriptor is not valid.

The two file descriptors returned are connected in a special way. Any data written to file\_descriptor[1]can be read back from file\_descriptor[0]. The data is processed in a first in, first out basis. This means that if we write the bytes 1, 2, 3 to file\_descriptor[1], reading fromfile\_descriptor[0] will produce 1, 2, 3. The illustration is given as below:



Each running program, called a process, has a number of file descriptors associated with it. These are small integers that we can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:

❑ 0: Standard input

❑ 1: Standard output

❑ 2: Standard error

We can associate other file descriptors with files and devices by using the open system call. The file descriptors that are automatically opened, however, already allow you to create some simple programs using write.

**The write( ) system call**

The write system call arranges for the first *nbytes* bytes from *buf* to be written to the file associated with the file descriptor *fildes*. It returns the number of bytes actually written. This may be less than *nbytes* if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size. If the function returns 0, it means no data was written; if it returns –1, there has been an error in the write call, and the error will be specified in the *errno* global variable.

**#include <unistd.h>**

**size\_t write(int fildes, const void \*buf, size\_tnbytes);**

**The read( ) system call**

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes actually read, which may be less than the number requested. If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return –1.

**#include <unistd.h>**

**size\_t read(int fildes, void \*buf, size\_t nbytes);**

**Pipes and FIFOs**

A pipe is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

A FIFO special file is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.

A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn’t have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn’t have a reading process is treated as an error condition; it generates a SIGPIPE signal, and fails with error code EPIPE if the signal is handled or blocked.

Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

**Pipe to a Subprocess**

A common use of pipes is to send data to or receive data from a program being run as a subprocess. One way of doing this is by using a combination of pipe (to create the pipe), fork (to create the subprocess), dup2 (to force the subprocess to use the pipe as its standard input or output channel), and exec (to execute the new program). Or, you can use popen and pclose.

The advantage of using popen and pclose is that the interface is much simpler and easier to use. But it doesn’t offer as much flexibility as using the low-level functions directly.

Function: **FILE \* popen (const char \*command, const char \*mode)**

The popen function is closely related to the system function; see Running a Command. It executes the shell command command as a subprocess. However, instead of waiting for the command to complete, it creates a pipe to the subprocess and returns a stream that corresponds to that pipe.

If you specify a mode argument of "r", you can read from the stream to retrieve data from the standard output channel of the subprocess. The subprocess inherits its standard input channel from the parent process.

Similarly, if you specify a mode argument of "w", you can write to the stream to send data to the standard input channel of the subprocess. The subprocess inherits its standard output channel from the parent process.

In the event of an error popen returns a null pointer. This might happen if the pipe or stream cannot be created, if the subprocess cannot be forked, or if the program cannot be executed.

Function: **int pclose (FILE \*stream)**

The pclose function is used to close a stream created by popen. It waits for the child process to terminate and returns its status value, as for the system function.

**FIFO Special 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 file system 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.

The mkfifo function is declared in the header file sys/stat.h.

Function: **int mkfifo (const char \*filename, mode\_t mode)**

The mkfifo function makes a FIFO special file with name filename. The mode argument is used to set the file’s permissions; see Setting Permissions.

The normal, successful return value from mkfifo is 0. In the case of an error, -1 is returned. In addition to the usual file name errors (see File Name Errors), the following errno error conditions are defined for this function:

EEXIST The named file already exists.

ENOSPC The directory or file system cannot be extended.

EROFS The directory that would contain the file resides on a read-only file system.

**CONCLUSION(Atleastthree points):**

**ASSIGNMENTS**:

1. Explain the use of | operator with the example of multiple commands. (At least three examples with practical demonstration is expected).

2. What is difference between pipe and shared memory implementation in Linux IPC?

3. What is difference between pipe and FIFO? Explain at least three points.

4. What are the advantages of FIFO over pipe?

5. Explain the situation where FIFO is appropriate structure used over pipe.

6. State the difference between named and unnamed pipes.

7. Enlist the system calls related to Signals

Program

1. Pipes

#include<stdio.h>

#include<unistd.h>

#include<string.h>

#include<stdlib.h>

#include<sys/types.h>

int main(void)

{

int fd1[2], nbytes=1,fd2[2],a=0;

pid\_t pid;

char string[80];

char readbuffer[80];

char ch='a',ch1='\n';

FILE \*fp;

pipe(fd1);//PIPE CREATED

pipe(fd2);//PIPE CREATED

/\*Error in fork\*/

if((pid = fork()) == -1)

{

perror("fork");

exit(1);

}

//Child Process

if(pid == 0)

{

close(fd1[1]); /\*closing write end of Pipe 1\*/

read(fd1[0], readbuffer, sizeof(readbuffer)); /\*reading filename through Pipe 1\*/

printf("\nFilename '%s' is being read by Child Process through Pipe 1...\n",readbuffer);

fp=fopen(readbuffer,"r");

close(fd1[0]); /\*closing read end of Pipe 1\*/

close(fd2[0]); /\*closing read end of Pipe 2\*/

printf("\nContents of %s are being sent to Parent Process through Pipe 2...\n",readbuffer);

while(a!=-1)

{

a=fscanf(fp,"%c",&ch);

write(fd2[1], &ch, sizeof(ch)); /\*writing contents of file on Pipe 2\*/

}

close(fd2[1]); /\*closing write end of Pipe 2\*/

exit(0);

}

//Parent process

else

{

close(fd1[0]); /\*closing read end of Pipe 1\*/

printf("IN PARENT PROCESS\n" );

printf("\nEnter name of file:");

scanf("%s",string);

printf("Filename is being sent by Parent Process to Child Process through Pipe 1...\n");

write(fd1[1], string, (strlen(string)+1)); /\*writing filename on Pipe 1\*/

wait();

close(fd1[1]); /\*closing write end of Pipe 1\*/

close(fd2[1]); /\*closing write end of Pipe 2\*/

printf("\nContents of %s are being received by Parent Process through Pipe 2...\n\n",string);

printf("IN PARENT PROCESS\n" );

printf("\nReceived Message:\n");

while(nbytes!=0)

{

printf("%c",ch1);

nbytes = read(fd2[0], &ch1, sizeof(ch1)); /\*reading contents of file from Pipe 2\*/

}

close(fd2[0]); /\*closing read end of Pipe 2\*/

}

return(0);

}

1. FIFO

Process 1 :

#include<stdio.h>

#include<fcntl.h>

#include<sys/stat.h>

#include<unistd.h>

#include<string.h>

#define MAX\_BUF 80

int main(){

int fd,i = 0;

char \*myfifo="myfifo";

char str[MAX\_BUF]="";

char ch;

mkfifo(myfifo,0666);

printf("\nEnter a string : ");

fgets(str,MAX\_BUF,stdin);

fd=open(myfifo,O\_WRONLY);

write(fd,str,strlen(str));

close(fd);

unlink(myfifo);

return 0;

}

Process 2:

#include<stdio.h>

#include<fcntl.h>

#include<sys/stat.h>

#include<unistd.h>

#include<string.h>

#define MAX\_BUF 1024

int main(){

int fd,k,i, c=0,c1=0;

char \*myfifo="myfifo";

char str[MAX\_BUF]="";

mkfifo(myfifo,0666);

fd=open(myfifo,O\_RDONLY);

read(fd,str,sizeof(str));

printf("\nReceived = %s\n",str);

k=strlen(str)-1;

printf("length = %d\n",k);

for(i=0;str[i]!='\0';i++)

if(str[i]==' ')

c++;

for(i=0;str[i]!='\0';i++)

if(str[i]=='\n')

c1++;

printf("\n\nTotal lines are %d\n",c1);

printf("\n\nTotal words are %d\n ",c+1);

close(fd);

return 0; }