Experiment Number: 06

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

OBJECTIVE:

1. To study use of pipes Linux.
2. To study inter-process communication in Linux.

THEORY:

The Pipe Call

The lower-level pipe( ) function provides a means of passing data between 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.

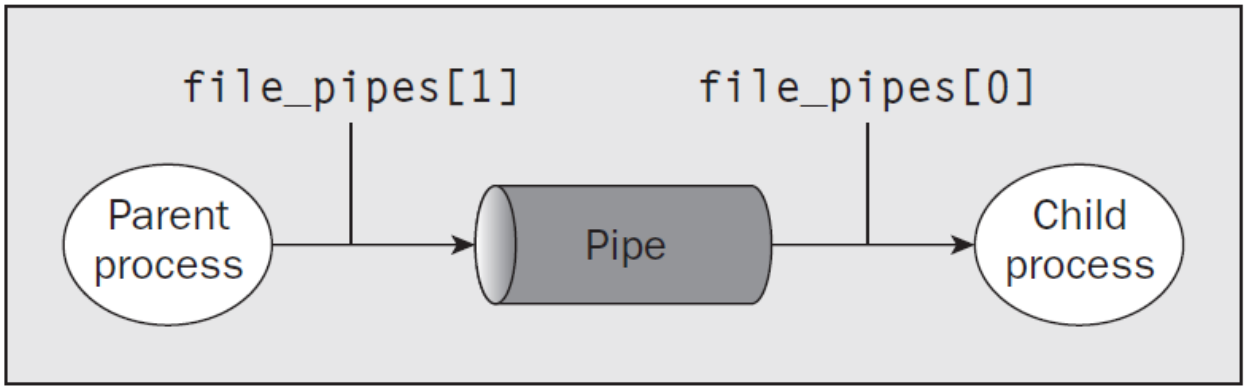
The pipe function has the following prototype:

```
#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.
- ❑ EFAULT: 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_t nbytes);
```

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);
```

ALGORITHM:

PROGRAM (With output and Comments):

CONCLUSION (At least three 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?

REFERENCES:

1. “Beginning Linux Programming” by Neil Mathew and Richard Stones, Wrox Publications.
2. “Operating System Internals and Design Principals” by William Stallings, Pearson Education.

Tushar B Kute
http://tusharkute.com