Practical-3

Objective: Inter Process Communications using Pipe and FIFO.

Theory/Concepts

Half-duplex UNIX Pipes

A pipe is a method of connecting the standard output of one process to the standard input of another. They provide a method of one-way communications between processes.

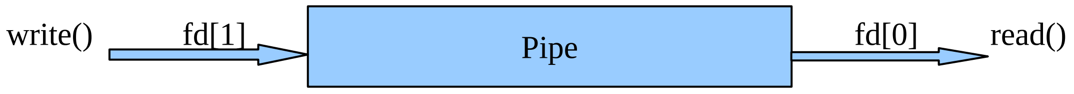
```
$ ls | sort | lp
```

The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp. The data is running through a half duplex pipe, traveling (visually) left to right through the pipeline.

When a process creates a pipe, the kernel sets up two file descriptors for use by the pipe.

One descriptor is used to allow a path of input into the pipe (write), while the other is used to obtain data from the pipe (read).

At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself.



At this point, the pipe is fairly useless.

After all, why go to the trouble of creating a pipe if we are only going to talk to ourself?

At this point, the creating process typically forks a child process.

Since a child process will inherit any open file descriptors from the parent, we now have the basis for interprocess communication (between parent and child).

Above, we see that both processes now have access to the file descriptors which constitute the pipeline.

It is at this stage, that a critical decision must be made. In which direction do we desire data to travel?

Does the child process send information to the parent, or vice-versa?

The two processes mutually agree on this issue, and proceed to “close”' the end of the pipe that they are not concerned with.

For discussion purposes, let's say the child performs some processing, and sends information back through the pipe to the parent.

Construction of the pipeline is now complete!

The only thing left to do is make use of the pipe.

To access a pipe directly, the same system calls that are used for low-level file I/O can be used (recall that pipes are actually represented internally as a valid inode).

To send data to the pipe, we use the write() system call, and to retrieve data from the pipe, we use the read() system call.

Remember, low-level file I/O system calls work with file descriptors!

However, keep in mind that certain system calls, such as lseek(), do not work with descriptors to pipes.

<u>Notes on half-duplex pipes:</u>

- Two way pipes can be created by opening up two pipes, and properly reassigning the file descriptors in the child process.
- The pipe() call must be made BEFORE a call to fork(), or the descriptors will not be inherited by the child! (same for popen()).
- With half-duplex pipes, any connected processes must share a related ancestry. Since the pipe resides within the confines of the kernel, any process that is not in the ancestry for the creator of the pipe has no way of addressing it. This is not the case with named pipes (FIFOS).

<u>Named Pipes (FIFO) : Basic Concepts</u>

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

<u>Creating a FIFO</u>

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception.

The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation.

With mknod, a quick call to the chmod command will be necessary.
FIFO files can be quickly identified in a physical file system by the “p”' indicator seen here in a long directory listing.

To create a FIFO in C, we can make use of the mknod() system call.

Consider a simple example of FIFO creation from C:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file ``/tmp/MYFIFO" is created as a FIFO file. The requested permissions are “0666'”.

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception.

An “open”' system call or library function should be used to physically open up a channel to the pipe.

With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem.

Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO.

In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing.

This action works vice-versa as well. If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

For further details:

http://os.obs.utcluj.ro/OS/Lab/08.Linux%20Pipes.html

PROGRAM 3.1

AIM: W.A.P TO IMPLEMENT HALF-DUPLEX PIPE.

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<string.h>
int main()
{
        int pipe1[2],size,n;
        char buffer[50];
        pipe(pipe1);
        if(fork() == 0)
        {
                close(pipe1[1]);
                n = read(pipe1[0],buffer, sizeof(buffer));
                write(1,"Received message:",18);
                write(1,buffer,n);
        }
        else
        {
                close(pipe1[0]);
                write(1,"Enter the message:\n",22);
                gets(buffer);
                write(pipe1[1],buffer,strlen(buffer));
                wait();
        }
        return 0;
}
```

OUT PUT 3.1

1st EXECUTION:

```
Enter the message:
good morning.
Received message:good morning.
```

2nd EXECUTION:

```
Enter the message:
Hello world.
Received message:Hello world.
```

PROGRAM 3.2

AIM: W.A.P TO IMPLEMENT ONE WAY FIFO.

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

int main()
{
	int n;
	mode_t mode;
	char fifo[10],buffer[50];
	int fd;

	printf("Enter name of fifo :");
	gets(fifo);
	mkfifo(fifo,S_IRUSR | S_IWUSR |S_IROTH | S_IRGRP);

	if(fork() == 0)
	{
		fd = open(fifo,O_RDONLY,0);
		n = read(fd,buffer,sizeof(buffer));
		write(1,"Received message:",18);
		write(1,buffer,n);
	}
	else
	{
		fd = open(fifo,O_WRONLY,0);
		write(1,"Enter the message:\n",22);
		gets(buffer);
		write(fd,buffer,strlen(buffer));
	}
	return 0;
}
```

OUT PUT 3.2

1st EXECUTION:

```
Enter name of fifo :abc
Enter the message:
Hello computer.
Received message:Hello computer.
```

2nd EXECUTION:

```
Enter name of fifo :xyz
Enter the message:
Good evening.
Received message:Good evening.
```

EXERCISES – 3

3.1) Write a C program that will create two child processes, which will use “pipe” for communication. First child will read the message from user and send it to other process via pipe. The second child process will read the message from pipe and display it on its standard output.

3.2) Write a C program that will create two child processes, say Child1 and Child2. which will use two “pipe”s for communication, say Pipe1 and Pipe2. Child1 will read the filename from the standard input and send it to Child2 via Pipe1. Now the Child1 will wait for reply from Child2.

The Child2 will read the filename from Pipe1 and try to open that file. If Child2 is able to open the file than read the file and send the file content to Child1 via Pipe2. Otherwise send the error message to Child1.

Child1 should now read the file content or error message from Pipe2 and show it onto its standard output.