# Chapter 6

# IPC: Pipes

Pipes are another IPC technique for pro cesses to communicate with one another. It can also be used by two threads within the same process to communicate.

In the description of le descriptors from Chapter-4, you will recall that there are three les that are open all the time for input and output purposes. These are:

1. The standard input
2. The standard output
3. The standard error

And that they have a le-descriptor of 0, 1, and 2 respectively. Whenever a program needs to display some output (via cout or printf), it will write that output to the standard out put le descriptor. This will in return be dis played on the monitor. Whenever a program needs to take some input from the keyboard, it will take it's input from the standard input le descriptor. Similarly, whenever an error needs to be displayed, that error will be sent to the standard error le descriptor. These les are linked-up internally to peripheral devices such as keyboard, monitor, etc. However, these linkages can be changed to point to something else. Pipes work by doing exactly that! So a pipe will::

- Redirect the standard output of one pro cess to become the standard input of an other

The rest of communication is done using the following rules:

- The pipe will be a bu er region in main memory which will be accessible by only two processes.
- One process will read from the bu er while the other will write to it.
- One process cannot read from the bu er unless and untill the other has written to it.

## 6.1 Pipe On the Shell

Run the following command:

```
pstree
```

As you will notice, the output is too long to t in the screen. Now run the command with:

```
pstree | less
```

Using the up and down arrows you will notice that you can browse through the output which was otherwise not visible in the rst command which we gave. To exit, press q.

The | is the symbol for pipe and as you would have guessed, pstree and less are two processes. In this usage, the standard output of pstree has become the standard input of the less program. Try the following command:

```
pstree | grep bash
```

Again, you will see that the output of above command is much di erent from just pstree command. What the above command should print is only those lines of text from the pstree output in which the keyword bash appears. Hence, pstree and grep are two separate pro cesses. But the standard output of the pstree command has become the standard input of the grep program. The pstree command writes out its output to the grep process. The grep process receives it, searches for keyword, for mats output, and then displays the result.

## 6.2 Pipe System Call

Type, compile, and run the following code:

```
#include <unistd.h>
int main()
{
   int pfd[2];
   pipe(pfd);
}
```

The above code creates a pipe using the pipe() system call. We have passed it the name of the integer array pfd that we have declared earlier on.

Task: Rewrite this code so that you can view the contents of the array using printf ar guments. You should see two numbers. What are these numbers?

Let us extend our code. Type, compile, and run the following:

```
01 #include <unistd.h>
02 int main()
03 {
04 int pid; // for storing fork() return 05 int pfd[2]; // for pipe
file descriptors 06 char aString[20]; // Temporary storage 07
pipe(pfd); // create our pipe
```

When line number 07 completes, we will have the following in our process:

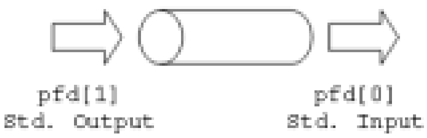
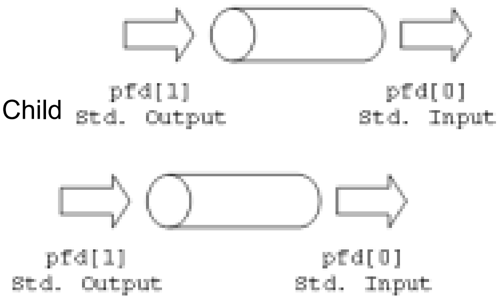



Figure 6.1: Before Fork()

```
08 pid = fork(); // create child process
```

When line number 08 completes, we will have the following in our two processes:

(a) In Child




(b) In Parent

Figure 6.2: After Fork() is made

Continuing with rest of code:

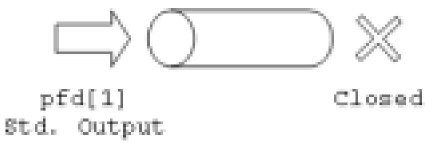
```
09 if (pid == 0) // For child 10 {
11 write(pfd[1], "Hello", 5); // Write onto pipe 12 }
13 else // For parent 14 {
15 read(pfd[0], aString, 5); // Read from pipe 16 }
17 }
```

Just like we open a  le, we read from a  le, we write to a  le, and we close a  le, we will perform the same operations of open(), close(), read() and write() on the pipe.

Task: Rewrite this code so that you can see the contents of aString in the parent be fore and after the read() call. What are the contents? You will notice that  Hello has been mentioned in the chid process. Then how is it possible that we are able to see the term  Hello  in the parent process? The answer is through the pipe mecha nism which we just used.

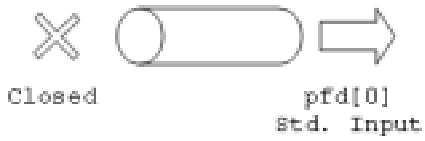
In the code we just saw, since the child is only going to write to a pipe and the parent will only read from the pipe, it makes sense to close the read capabilities for the child and write capa bilities for the parent for that particular pipe. Diagramatically, we want to achieve something like the following:






close(pfd[1]);

(a) In Child




(b) In Parent

Figure 6.3: Closing un-necessary ends of the Pipe

For that, we have to add the following before line 11:

```
close(pfd[0]);
```

and the following before line 15:

```
close(pfd[1]);
```

### 6.2.1 Example

Type, run and execute the code below. It should give output which is equivalent to the command ls | wc (Wc is used for printing three numbers; the number of newlines, the number of words, and the byte count for a  le). Note the usage of pipes.

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
int main()
{
   int pfd[2];
   pipe(pfd);
   if (fork() == 0)
   {
      close(pfd[1]);
      dup2(pfd[0], 0);
      close(pfd[0]);
      execlp("wc", "wc", (char *) 0);
   }
   else
   {
      close(pfd[0]);
      dup2(pfd[1], 1);
      execlp("ls", "ls", (char *) 0);
   }
}
```