**BRIEF INTRODUCTION TO INTERPROCESS COMMUNICATIONS**

We will read about inter-process communication between processes. There are many mechanisms that are available for a system programmer to use. The mechanisms range from data transfer from one process to another ( via pipes, FIFOs, message queues, shared memory ), signaling a process to handle or ignore an event, and lastly synchronization (using semaphores, mutexes, condition variables ). Pipes are simple, but are less desirable because they are half-duplex ( you can read or write ) compared to shared memory. Shared memory is fast because a process can create a shared memory for all processes involved to read or write.

**PIPES**

We use a pipe to allow communication between two processes. A pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child process. A pipe is half duplex, i.e. data flows in only one direction from parent process to child process or child to parent process.

Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe. For example, consider this shell command

**ls –l | wc –l**

would create two processes : one for the ls –l and the other for wc –l . A pipe for the output of the first process is created and the data is piped to the second. The pipe in the second is created for reading the input data. When both are done completing the data transfer, the pipes are destroyed and then the processes.

General characteristics of pipe are:

**A pipe is a byte stream** - there is no concept of messages or message boundaries when using a pipe. The process reading from a pipe can read blocks of data of any size, regardless of the size of blocks written by the writing process. Furthermore, the data passes through the pipe sequentially— bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data

**Reading from a pipe** - Attempts to read from a pipe that is currently empty block until at least one byte has been written to the pipe. If the write end of a pipe is closed, then a process reading from the pipe will see end-of-file ( i.e., read ( ) returns 0) once it has read all remaining data in the pipe.

**Half-Duplex**- Pipes are unidirectional Data can travel only in one direction through a pipe. One end of the pipe is used for writing, and the other end is used for reading.

**Pipes have a limited capacity** - Amount of Data can be written - PIPE\_BUF is a constant that varies across UNIX implementations; for example, it is 512 bytes on FreeBSD 6.0, 4096 bytes on Tru64 5.1, and 5120 bytes on Solaris 8. On Linux, PIPE\_BUF has the value 4096. When a process is writing data to a pipe, another process should be reading it because there is a limit of size. If the size is exceeded, the process that is writing is blocked. When there is no data on the pipe, if a process is reading it, it will get end of file.

Creating and Using Pipes programmatically :

The pipe ( ) system call creates a new pipe

#include <unistd.h>

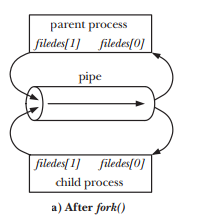
int pipefd [ 2 ] ; // step1 : create an array of two cells

int pipe ( pipefd ) ; // step2 : pass the array to the pipe function , which returns

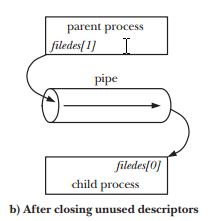
A successful call to pipe ( ) returns two open file descriptors in the array pipefd : one for the read end of the pipe (pipefd [ 0 ] ) and one for the write end (pipefd [ 1 ] ) .

As with any file descriptor, we can use the read ( ) and write ( ) system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end.

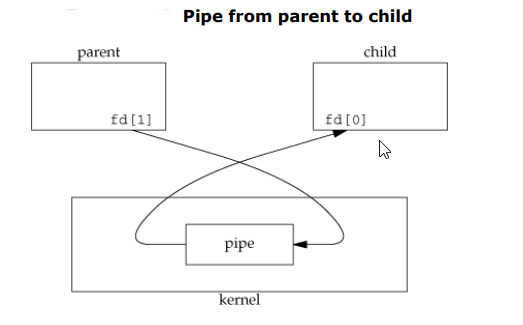
To connect two processes using a pipe, we follow the pipe() call with a call to fork(). During a fork(), the child process inherits copies of its parent’s file descriptors . See the figure .



To start communication between two processes (say parent and child), immediately after the fork ( ) by a parent, one process closes its descriptor for the write end of the pipe, and the other closes its descriptor for the read end. For example, if the parent is to send data to the child, then it would close its read descriptor for the pipe, fd [ 0 ], while the child would close its write descriptor for the pipe, fd [ 1 ] . Otherwise, if both parent and child write into the pipe or read from it, we may not know which process wrote into the pipe or read from the pipe, causing race conditions.



Pipes are generally done in kernel space, so reading or writing involves system call. So we have this diagram



Here is the sample program

#include <sys/wait.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define MAX\_LINE 32

int main(int argc, char \*argv[])

{

int pipefd[2];

pid\_t cpid;

char buf[ MAX\_LINE ];

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

cpid = fork();

if (cpid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (cpid == 0) { /\* Child reads from pipe \*/

close(pipefd[1]); /\* Close unused write end \*/

int num = read(pipefd[0], &buf, MAX\_LINE) ;

printf ( " Reading in child process\n");

printf( "%s \n", buf );

close(pipefd[0]);

} else { /\* Parent writes argv[1] to pipe \*/

close(pipefd[0]); /\* Close unused read end \*/

write(pipefd[1], "Hello World", 12);

close(pipefd[1]); /\* Reader will see EOF \*/

wait(NULL); /\* Wait for child \*/

}

exit( EXIT\_SUCCESS);

}

In the previous example, we called read and write directly on the pipe descriptors. The parent process closes the file descriptor pipefd [ 0 ] , and then writes “hello world” into the file descriptor pipefd [ 1 ]. The child process closes the file descriptor pipefd [ 1 ] and reads the data from pipefd [ 0 ] and then writes the data to the standard out .