PIPES FOR INTER-PROCESS COMMUNICATION

Now that we know how to create processes, let us turn our attention to make the processes communicate among themselves. There are many mechanisms through which the processes communicate and in this lab we will discuss one such mechanism: Pipe. A pipe is used for one-way communication of a stream of bytes.

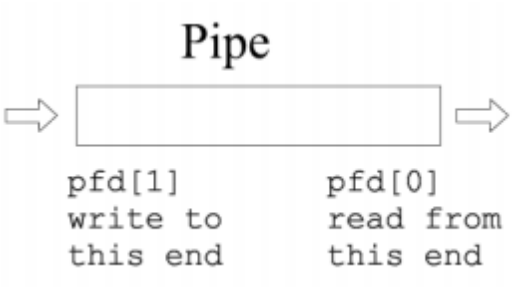
In this lab we will learn how to create pipes and how processes communicate by reading or writing to the pipe and also how to have a two-way communication between processes.

PIPE

Pipes are familiar to most Unix users as a shell facility. For instance, to print a sorted list of who is logged on, you can enter this command line: who | sort | lpr There are three processes here, connected with two pipes. Data flows in one direction only, from who to sort to lpr. It is also possible to set up bidirectional pipelines (from process A to B, and from B back to A) and pipelines in a ring (from A to B to C to A) using system calls. The shell, however, provides no notation for these more elaborate arrangements, so they are unknown to most Unix users. We can begin by showing some simple examples of processes connected by a one-directional pipeline.

pipe System Call :

```
int pfd[2];
int pipe (pfd);  /* Returns 0 on success or -1 on error */
```



I/O with a pipe:

These two file descriptors can be used for block I/O

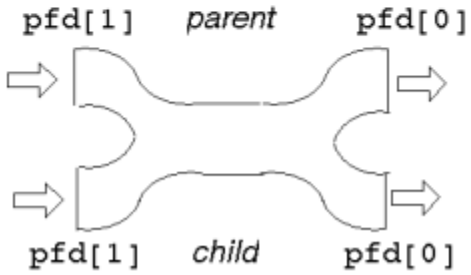
```
write(pfd[1], buf, SIZE);

read(pfd[0], buf, SIZE);
```

Fork and a pipe:

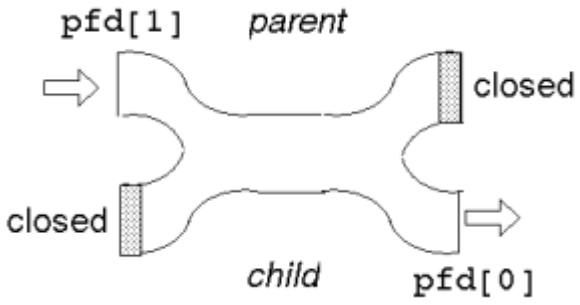
A single process would not use a pipe. They are used when two processes wish to communicate in a one-way fashion. A process splits in two using fork ( ). A pipe opened before the fork becomes shared between the two processes.

Before fork

After fork



This gives two read ends and two write ends. The read end of the pipe will not be closed until both of the read ends are closed, and the write end will not be closed until both the write ends are closed. Both processes can write into the pipe, and either can read from it. Which process will get what is not known? For predictable behavior, one of the processes must close its read end, and the other must close its write end. Then it will become a simple pipeline again.



Suppose the parent wants to write down a pipeline to a child. The parent closes its read end, and writes into the other end. The child closes its write end and reads from the other end. When the processes have ceased communication, the parent closes its write end. This means that the child gets eof on its next read, and it can close its read end. Pipes use the buffer cache just as ordinary files do. Therefore, the benefits of writing and reading pipes in units of a block (usually 512 bytes) are just as great. A single write execution is atomic, so if 512 bytes are written with a single system call, the corresponding read will return with 512 bytes (if it requests that many). It will not return with less than the full block. However, if the writer is not writing complete blocks, but the reader is trying to read complete blocks, the reader may keep getting partial blocks anyhow. This won't happen if the writer is faster than the reader, since then the writer will be able to fill the pipe with a complete block before the reader gets around to reading anything. Still, it's best to buffer writes and reads on pipes, and this is what the Standard I/O Library does automatically.

```c
#include <stdio.h>

#define SIZE 1024

main( )
{
 int pfd[2];
 int nread;
 int pid;
 char buf[SIZE];

 if (pipe(pfd) == -1)
 {
  perror("pipe failed");
  exit(1);
 }
 if ((pid = fork()) < 0)
 {
  perror("fork failed");
  exit(2);
 }
 if (pid == 0)
 {
  /* child */
  close(pfd[1]);
  while ((nread = read(pfd[0], buf, SIZE)) != 0)
   printf("child read %s\n", buf);
  close(pfd[0]);
 }
 else
 {
  /* parent */
  close(pfd[0]);
  strcpy(buf, "hello...");
  /* include null terminator in write */
  write(pfd[1], buf, strlen(buf)+1);
  close(pfd[1]);
 }
}
```

Given that we have two processes, how can we connect them so that one can read from a pipe what the other writes? We can't. Once the processes are created they can't be connected, because there's no way for the process that makes the pipe to pass a file descriptor to the other process. It can pass the file descriptor number, of

course, but that number won't be valid in the other process. But if we make a pipe in one process before creating the other process, it will inherit the pipe file descriptors, and they will be valid in both processes. Thus, two processes communicating over a pipe can be parent and child, or two children, or grandparent and grandchild, and so on, but they must be related, and the pipe must be passed on at birth. In practice, this may be a severe limitation, because if a process dies there's no way to recreate it and reconnect it to its pipes -- the survivors must be killed too, and then the whole family has to be recreated.

In general, then, here is how to connect two processes with a pipe:

- Make the pipe.
- Fork to create the reading child.
- In the child **close the writing end of the pipe**, and do any other preparations that are needed.
- In the child execute the child program.
- In the parent **close the reading end of the pipe**.
- If a second child is to write on the pipe, create it, make the necessary preparations, and execute its program. If the parent is to write, go ahead and write.

Here’s a small program that uses a pipe to allow the parent to read a message from its child:

```c
#include <stdio.h>
#include <string.h>

#define  READ   0
#define  WRITE  1

char*  phrase  =  "This is ICS431 lab time"
main ( )
{
 int     fd [2],  bytesread ;
 char    message [100] ;

 pipe ( fd ) ;                                   /*  child, writer */
 if ( fork ( )  ==  0 )
 {                                               /*  close unused end */
    close ( fd [READ] ) ;
    write  ( fd [WRITE], phrase, strlen (phrase) + 1) ;
    close ( fd [WRITE] ) ;                       /*  close used end  */
 }
 else                                            /*  parent,  reader  */
 {
 close ( fd [WRITE] ) ;                /* close unused end  */
 bytesread = read (fd [READ], message, 100) ;
 printf ("Read %d bytes : %s\n", bytesread, message) ;
 close ( fd [READ] ) ;                           /*  close used end  */
```