DEPARTMENT OF INFORMATION TECHNOLOGY

IT 253 Operating Systems Lab

LAB2: 28/02/2023

Evaluation: 10 Marks

Objective

To understand the Inter process communication - Pipes

Inter process communication (IPC) -Pipes

1. Read these Basics for understanding IPC: pipes:

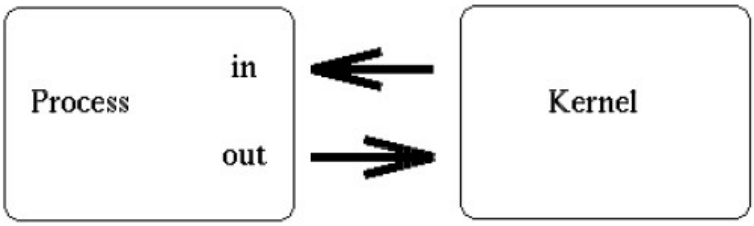
Pipe is a method of connecting the standard output of one process to the standard input of another. Pipes are the eldest of the IPC tools, having been around since the earliest incarnations of the UNIX operating system. They provide a method of one-way communications (half-duplex) between processes.

This feature is widely used, even on the UNIX command line (in the shell).

```
ls | sort | lp
```

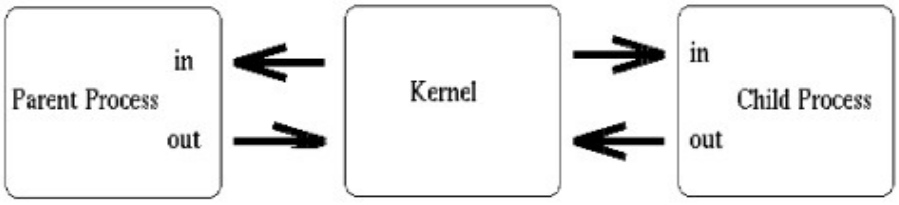
The above sets up a pipeline, taking the output of *ls* as the input of *sort*, and the output of sort as the input of *lp*. The data is running through a half duplex pipe, traveling (visually) left to right through the pipeline.

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. Consider this representation of a process and the kernel after a pipe has been created:

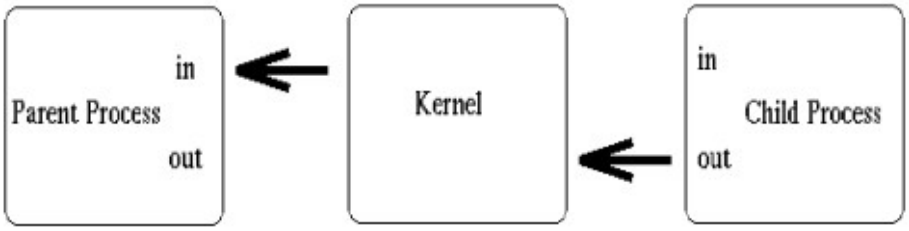


From the above diagram, it is easy to see how the descriptors are connected together. If the process sends data through the pipe (fd0), it has the ability to obtain (read) that information from fd1. However, there is a much larger objective of the simplistic sketch above. While a pipe initially connects a process to itself, data traveling through the pipe moves through the kernel. Under Linux, in particular, pipes are actually represented internally with a valid *inode*. This *inode* resides within the kernel itself, and not within the bounds of any physical file system.

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 multiprocess communication (between parent and child). Consider this updated version of our simple sketch:



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. Our newly revised sketch would appear as such:



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.

Creating Pipes in C: To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

```
SYSTEM CALL: pipe();

  PROTOTYPE: int pipe( int fd[2] );
    RETURNS: 0 on success
             -1 on error: errno = EMFILE (no free descriptors)
                                  EMFILE (system file table is full)
                                  EFAULT (fd array is not valid)

  NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.

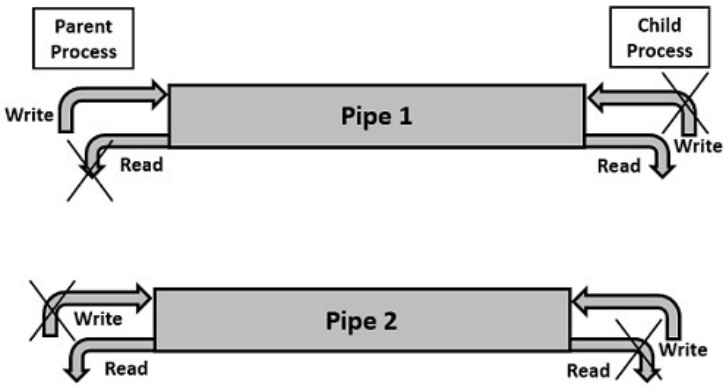
If the parent wants to send data to the child, it should close fd0, and the child should close fd1.

```
if(childpid == 0)

{
        /* Child process */
        close(fd[1]);
}
else
{
        /* Parent process */
        close(fd[0]);
}
```

If the parent wants to receive data from the child, it should close fd1, and the child should close fd0.

```
if(childpid == 0)

{
        /* Child process */
        close(fd[0]);
}
else
{
        /* Parent process */
        close(fd[1]);
}
```

Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with. On a technical note, the EOF will never be returned if the unnecessary ends of the pipe are not explicitly closed.



```
//Child Sends data and parent Receives the same
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    main()
    {
            int     fd[2];
            pid_t   childpid;

            pipe(fd);

            if((childpid = fork()) == -1)
            {
                    perror("fork");
                    exit(1);
            }

            if(childpid == 0)
            {
                    /* Child process closes up input side of pipe */
                    close(fd[0]);
            }
            else
            {
                    /* Parent process closes up output side of pipe */
                    close(fd[1]);
            }
            .
            .
    }
```

As mentioned previously, once the pipeline has been established, the file descriptors may be treated like descriptors to normal files.

2. Execute the following program where a child sends message to parent. Observe the result write your observation along with screenshots of result. [Marks : 2 Marks]

The child process closes fd[0] and parent closes fd[1] as they are not used.

Child process writes to fd[1] and parent process reads from fd[0].

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

3. modify the above program to send the message by parent process to the child process. Observe the result write your observation along with screenshots of result. (Note: Pipes are half duplex.) [Marks : 2 Marks]

4. A client program reads a file name from the standard input and sends it to server program. The server program opens the file if it is present, reads the content of file and sends it back to client program. If file is not present, then the server sends an error message. Implement the client server program using inter process communication: Pipes (client as child process and server as parent process) [Marks : 6 Marks]

[Note: The file which you are uploading should also contain source code of this program]

Use manual for further details.

Create pipe:
Int pipe(int fd[2]);

Pid=fork()

if(pid>0) //parent process
{
//Close unused pipe connections
read/write
//close used pipe connections
}
Else
{
//Close unused pipe connections
read/write
//close used pipe connections
}

Use files for read and write operation.
