OPERATING SYSTEMS LAB

LAB MANUAL # 10

Instructor: Muhammad Abdullah Orakzai

Semester Fall 2021

Course Code: CL2006

Fast National University of Computer and Emerging Sciences, Peshawar

Department of Computer Science

Contents

Inter Process Communications (Pipes)

IPC: Pipes

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

In the description of file descriptors from lab # 09, you will recall that there are three files that are open all the time for input and output purposes. These are:

1. The standard input ------> 0
2. The standard output ------> 1
3. The standard error ------> 2

And that they have a file-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 output file descriptor (**1**). This will in return be displayed on the monitor. Whenever a program needs to take some input from the keyboard, it will take it's input from the standard input file descriptor (**0**). Similarly, whenever an error needs to be displayed, that error will be sent to the standard error file descriptor (**2**). These files 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 process to become the standard input of another

The rest of communication is done using the following rules:

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

Pipe on the Shell

Run the following command:

```
pstree
```

As you will notice, the output is too long to fit 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 first 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 different 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 (**Global Regular Expression Print**) are two separate processes. 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, formats output, and then displays the result.

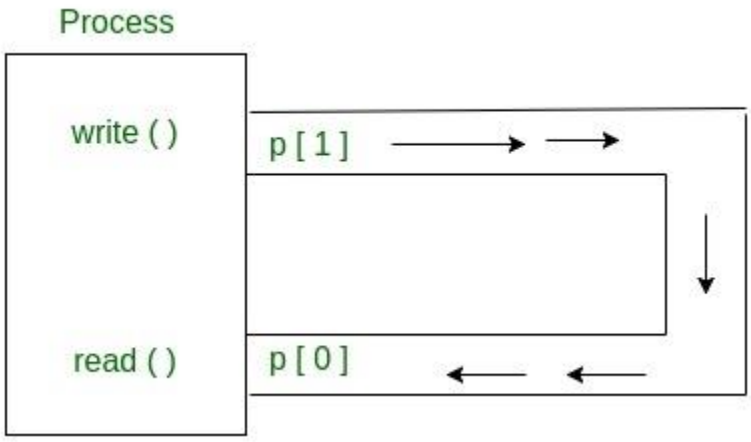
pipe() System call

Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. It is possible to have a series of processes arranged in a pipeline, with a pipe between each pair of processes in the series. In **UNIX Operating System**, Pipes are useful for communication between related processes (**inter-process communication**).

Implementation: A pipe can be implemented as a 10k buffer in main memory with 2 pointers, one for the **FROM** process and one for **TO** process. One process cannot read from the buffer until another has written to it.

- pipe() is a system call that facilitates **inter-process communication**.

- Pipe is one-way communication only i.e. we can use a pipe such that One process writes to the pipe, and the other process reads from the pipe. It opens a **pipe**, which is an area of main memory that is treated as a ***"virtual file"***.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this "**virtual file**" or **pipe** and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe. Recall that the open system call allocates only one position in the open file table.



Syntax in C language:

```
int pipe(int fds[2]);
```

or

```
int fds[2];
int result = pipe(fds);
```

Parameters:

`fd[0] will be the fd(file descriptor) for the read end of pipe.`

`fd[1] will be the fd for the write end of pipe.`

Returns:

```
0 on Success.
-1 on error.
```

Pipes behave **FIFO**(First in First out), Pipe behave like a **queue** data structure. Size of read and write don't have to match here. We can write **512** bytes at a time but we can read only 1 byte at a time in a pipe.

Examples of Pipe System Call

Example # 01

With error checking

pipe.c

```
#include<unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int pip[2];
        int result;
        result = pipe(pip);
        printf("Success Status: %d\n", result);
        if (result == -1)
        {
        perror("pipe");
        exit(1);
        }
        return 0;
}
```

Type, compile, and run the following code:

Example # 02

pipe1.c

```
#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 arguments. You should see two numbers. What are these numbers?

```
#include<unistd.h>

#include <stdio.h>

#include <stdlib.h>

int main()

{

        int pfd[2];

        pipe(pfd);

        printf("%d\n",pfd[0]);

        printf("%d\n",pfd[1]);

        return 0;

}
```

Example # 03

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

pipe2.c

```
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:

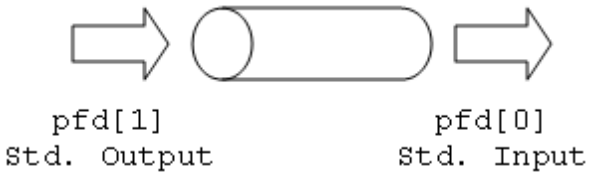
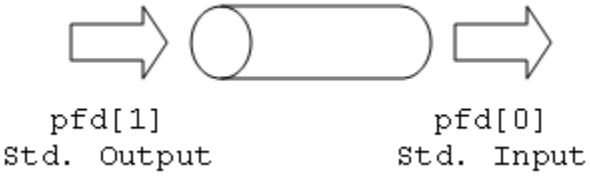


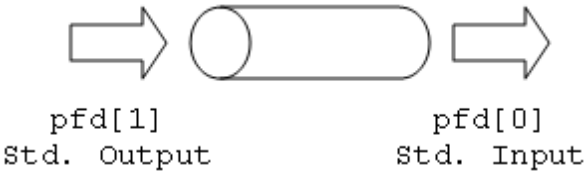
Figure 10.1: Before fork()

```
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 10.2: 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  }
```

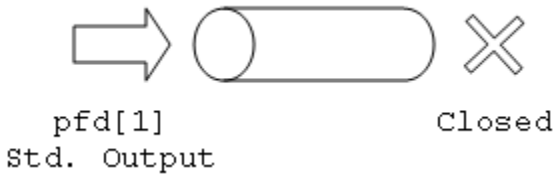
// Data is read from pipe using file descriptor and saved into a buffer namely sString after reading.

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

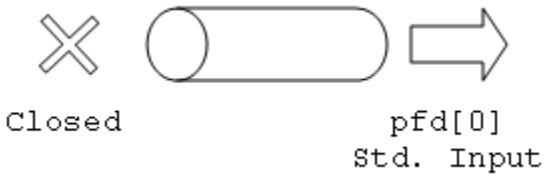
Task: Rewrite this code so that you can see the contents of aString in the parent before and after the read() call. What are the contents? You will notice that "**Hello**" has been mentioned in the child process. Then how is it possible that we are able to see the term "**Hello**" in the parent process? The answer is through the pipe mechanism 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 capabilities for the parent for that particular pipe.

Diagrammatically, we want to achieve something like the following:



(a) In Child



(b) In Parent

Figure 10.3: 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]);
```

Example # 04

pipe3.c

```
#include <unistd.h>
#include<stdio.h>
int main()
{
	int pid, pip[2];
	char instring[20];
	pipe(pip);
	pid = fork();
	if (pid == 0)  /* child : sends message to parent*/
	{
		/* send 7 characters in the string, including end-of-string */
		write(pip[1], "Hi Mom!", 7);
	}
	else	/* parent : receives message from child */
	{
		/* read from the pipe */
		read(pip[0], instring, 7);
		printf("Message of child: %s\n", instring);
	}
}
```

Example # 05

pipe4.c

```
#include<stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
  int status, pid, pipefds[2];
  char instring[20];
/* Create the pipe and return 2 file descriptors in the pipefds array */
/* This operation is done before the fork so that both processes will know about the same pipe, which will allow them to communicate.*/
   status = pipe(pipefds);
   if (status == -1)
   {
        perror("Trouble");
        exit(1);
   }
 /* create child process; both processes continue from here */
   pid = fork();
   if (pid == -1)
   {
        perror("Trouble");
        exit(2);
   }
   else if (pid == 0)    /* child : sends message to parent*/
        {
           /* close unused end of pipe */
        /* because this process only needs to write */
```

```
            close(pipefds[0]);

          /* send 7 characters in the string, including end-of-string */
            printf("About to send a message:\n");
          write(pipefds[1], "Hi Mom!", 7);
            close(pipefds[1]);
            exit(0);
          }
    else   /* parent : receives message from child */
    {
          /* close unused end of pipe */
          /* because this process only needs to read */
          close(pipefds[1]);
          /* read from the pipe */
       read(pipefds[0], instring, 7);
          printf("Just received a message that says: %s\n",instring);
          close(pipefds[0]);
          exit(0);
     }
}
```

Example # 06

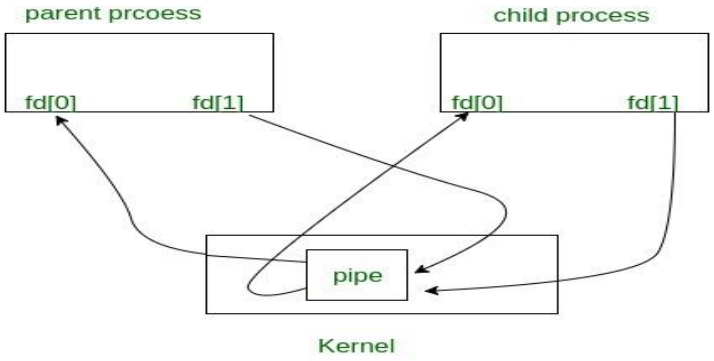
pipe5.c

```
// C program to illustrate pipe system call in C
#include <stdio.h>
#include <unistd.h>
#include<stdlib.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
```

```
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";
int main()
{
        char inbuf[MSGSIZE];
        int p[2], i;

        if (pipe(p) < 0)
                exit(1);

        /* continued */
        /* write pipe */
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);

        for (i = 0; i < 3; i++) {
                /* read pipe */
                read(p[0], inbuf, MSGSIZE);
                printf("% s\n", inbuf);
        }
        return 0;
}
```

Parent and child sharing a pipe

When we use **fork** in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



Example # 07

pipe6.c

```
// C program to illustrate pipe system call in C shared by Parent and Child
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
        char inbuf[MSGSIZE];
        int p[2], pid, nbytes;

        if (pipe(p) < 0)
                exit(1);
```

```
        /* continued */
        if ((pid = fork()) > 0) {
                write(p[1], msg1, MSGSIZE);
                write(p[1], msg2, MSGSIZE);
                write(p[1], msg3, MSGSIZE);

                // Adding this line will
                // not hang the program
                // close(p[1]);
                wait(NULL);
        }

        else {
                // Adding this line will not hang the program
                // close(p[1]);
                while ((nbytes = read(p[0], inbuf, MSGSIZE)) > 0)
                        printf("% s\n", inbuf);
                if (nbytes != 0)
                        exit(2);
                printf("Finished reading\n");
        }
        return 0;
}
```

Here, In this code After finishing reading/writing, both parent and child block instead of terminating the process and that's why program hangs. This happens because read system call gets as much data it requests or as much data as the pipe has, whichever is less.

If pipe is empty and we call read system call then Reads on the pipe will return **EOF (return value 0)** if no process has the write end open.

If some other process has the pipe open for writing, read will block in anticipation of new data so this code output hangs because here write ends parent process and also child process doesn't close.

For more details about parent and child sharing pipe, please refer C program to demonstrate fork() and pipe().

Example # 08

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 file).

Note the usage of pipes.

```
/*
   Demonstrate the use of a pipe to connect two filters. We use
fork() to create two children. The first one execs ls(1), which
writes to the pipe, the second execs wc(1) to read from the pipe.
*/
```

pipe7.c

```
#include<unistd.h>
#include<string.h>
#include<stdio.h>
/* This program constructs "ls | wc ", through a pipe:
 *
 *   stdin ---> ls ---> [pipe] ---> wc ---> stdout
 *
 * No arguments -- just run it.
 */
int main()
{
        int pid;
        int pfd[2];   //Pipe descriptor
        pipe(pfd);
        pid=fork();
```

```
if(pid==0)
{
   /* Don't need the write end of the pipe in this process */
    close(pfd[1]);

 /* Copy the read end of the pipe (pfds[0]) into stdin's  file descriptor (file descriptor 0). The old stdin is closed automatically first. */
    dup2(pfd[0], 0);
    close(pfds[0]);
   //execlp("wc", "wc",(char *) 0);/* exec 'wc' to read from pipe */
    //execlp("wc", "wc", (char *) NULL);
     execlp("wc","wc", NULL);
}
else
{
    /* Don't need the read end of the pipe in this process */
    close(pfd[0]);
   /* Copy the write end of the pipe (pfds[1]) into stdout's file descriptor (file descriptor 1). The old stdout  is closed automatically first. */
    dup2(pfd[1], 1);
    close(pfd[1]);
    //execlp("ls", "ls", (char *) 0);   /* Writes to pipe */
    execlp("ls", "ls", NULL);
}
}
```

dup( ) and dup2( ) Routines

Most programs are designed to read from standard input and write to standard output. How do communicate with such a program via a pipe? We use dup( ) or dup2( ).

The dup(oldfd) routine duplicates an existing file descriptor. The new file descriptor returned by dup( ) is guaranteed to be the lowest numbered available. **dup2(oldfd, newfd)** duplicates oldfd over newfd, closing newfd first, if necessary.

dup2 - duplicate an open file descriptor

Enter ***man dup2*** command in terminal for more detail about dup2.

SYNOPSIS

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

OR

```
#include <unistd.h>

int dup(int oldfd);

int dup2(int oldfd, int newfd);
```

The **dup2(pfd[0], 0);** code copies the read end of the pipe to the child's standard input. It's easy enough to remember which is which: stdin is file descriptor 0 and pipe descriptor 0 of the pair is the input descriptor (read end of the pipe), while stdout is file descriptor 1 and pipe descriptor 1 of the pair is the output descriptor (write end of the pipe).

execlp()

Execute a file

```
#include <unistd.h>
int execlp( const char * file,
        const char * arg0,
        const char * arg1,
        …
        const char * argn,
        NULL );
```

This function is declared in `<process.h>`, which `<unistd.h>` includes.

Arguments

file

Used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable **PATH**.

arg0, …, argn

Pointers to NULL-terminated character strings. These strings constitute the argument list available to the new process image. You must terminate the list with a NULL pointer. The *arg0* argument must point to a filename that's associated with the process being started and cannot be NULL.

The list of arguments must be terminated by a null pointer, and, since these are variadic functions, this pointer must be cast (char *) NULL.

References

https://www.geeksforgeeks.org/pipe-system-call/

http://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/pipes/pipes.html#:~:text=Pipes,of%20processes%20in%20the%20series.

https://www.geeksforgeeks.org/c-program-demonstrate-fork-and-pipe/

https://www.geeksforgeeks.org/piping-in-unix-or-linux/