UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COP4600

Ex7: Pipes

Overview

The aim of this exercise is to familiarize you with pipes in Linux. Pipes are a common form of inter-process communication.

Structure

The assignment is broken into three main parts. In each part, you will learn about a different way to implement pipes. The three parts are as follows:

1. Basic Pipes
2. Named Pipes
3. Pipe System Call

Part 1: Basic Pipes

Basic pipes can be created in the terminal by placing the "|" operator between commands. Using "|" causes the output of the left command to be piped in as the input to the right command. For example, running "cat result.txt | grep -o "COP4600" | wc -l" would return the number of times that the pattern "COP4600" is found in result.txt. The contents of the file are being piped into the grep command which then passed any matches into the wc command. The "wc -l" option returns the number of newlines which, in this case, equals the number of occurrences of the pattern.

For this part, an executable file named "part1.o" is provided via Canvas. When run, the executable will perform basic operations until it encounters a failure. The amount of successful operations completed before failing will vary. Here is a sample output of this program:

```
reptilian@localhost:~/Ex6$ ./part1.o
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation successful.
Operation failed.
reptilian@localhost:~/Ex6$
```

You may need to give part1.o executable permissions using "`sudo chmod +x ./part1.o`".

Your task is to create a C++ program that, when the output of "part1.o" is piped in, determines which operation number caused the failure. For example, your program would print "Program failed on operation 12." if it were run with the information from the above screenshot. For this part you will need to submit a screenshot that shows your code successfully running twice with input provided by "part1.o" through a basic pipe.

Part 2: Named Pipes

Named pipes, also known as FIFOs (first-in first-out), are functionally similar to basic pipes except they are represented as a special file and are, therefore, a non-temporary part of the filesystem. A FIFO can be created using the "mkfifo [name]" command (similar to mkdir). A FIFO does not store the communicated data in the file system and only acts as a reference point for processes to establish a connection. Because of this, when a writer process opens the FIFO, it will **block** until a reader process opens the other end. Utilizing named pipes through the command line, therefore, will likely require multiple terminal windows**.**

For this part, implement the following:

1. Create a named pipe.
2. Modify your part 1 C++ code to read its input from a file (your named pipe). This program should be named "lastname_part2.cpp" where "lastname" is your last name.
3. Run "part1.o" and redirect its output into the named pipe.
4. Compile and run your part 2 program (input **should not** be redirected from the command line).
5. Repeat steps 3 and 4 then take a screenshot that shows the programs running to completion twice. Be sure to include all terminal windows that were used.

Part 3: Pipe System Call

Both piping methods used above have required some form of post-compile time actions (i.e. redirecting output or creating a named pipe). In many cases, this may be a drawback. To set up a pipe entirely within the process' code, the pipe system call can be used.

For this part you will use the "fork()" and "pipe()" system calls to implement the following:

1. Create a new C++ program named "<lastname>_part3.cpp" filling in with your last name. The program should accept 5 command-line arguments (all integers).
2. The program must create **two** new child processes and utilize **four** pipes.
3. The parent process will send the 5 integers to the first child process using a pipe. This child will sort the integers and send the result to the parent process and the second child process using one pipe for each. The parent will print the result.
4. After receiving the sorted list from the first child process, the second child process will identify the median value of the list and send the result to the parent using a pipe. The parent will print the result.
5. Take a screenshot of the program running with the following integers passed in as command-line arguments (in the specified order): 42, 15, 8, 16, and 23.

Submissions

You will submit the following at the end of this exercise on Canvas:

- Screenshot of the output from running your part 1 program with a basic pipe ***twice*** (`part1.png`)
- Screenshot of the output from running your part 2 program with a named pipe ***twice*** (`part2.png`)
- C++ source file for your part 2 program (`lastname_part2.cpp`)
- Screenshot of the output from running your part 3 program with the given values (`part3.png`)
- C++ source file for your part 3 program (`lastname_part3.cpp`)

All parts of the exercise must be completed in Reptilian.

Helpful Links

https://www.linuxjournal.com/article/2156
https://www.tldp.org/LDP/tlk/ipc/ipc.html
http://www.cplusplus.com/doc/tutorial/files/
http://www.gnu.org/software/libc/manual/html_node/Creating-a-Pipe.html
http://man7.org/linux/man-pages/man2/pipe.2.html