O.S. Lab 1B: Linux Inter-Process Communication

The objective of this lab is to help you internalize a couple of important facts about Linux pipes:

- To send structured data through a pipe, you must *serialize it* to push it from the sending to receiving process as a stream of bytes. For instance, imagine you are sending a struct type: you must turn the struct into a sequence of bytes to send them over the pipe.
- When you send two different instances of serialized data structures over a serial channel such as a Linux pipe, you must be able to distinguish where one datum ends and the next one begins. What you use as the "marker" or "sentinel value" that distinguishes the two instances is up to you to define.

Set Up

Create a directory for **Lab1b** under your directory of work for this class. You will work with a simple program for this pre-lab, which should be added to this directory.

Problem 0: Pre-Lab (3 points)

1. Read the manual pages for the following system calls: **pipe(2)**, **open(2), read(2)**, **write(2)**, **close(2)**.
2. Find the file called **pipe-test.c** and add a copy to your **Lab1b** directory. Make sure the program **compiles** and **executes** correctly.
3. As before, create a **Makefile** to compile **pipe-test.c** *and all of the programs you will create or modify* for this lab. (**Not providing a Makefile that builds all deliverables will assign you a 2 point penalty to the overall assignment.**) Add to your **Lab1** folder to submit later.
4. Finally, create a file called **answers-prelab.txt** to contain the answers to the questions below (remember that you're shooting for **brief**, **clear**, and **accurate** text!)
    a. Explain how it is possible for the child process to have access to the same pipe which was created and later is used by the parent.
    b. What are the similarities and the differences you observe in **working with** Linux files and pipes? (Never mind how differently they might be implemented by the operating system, we are talking about *how* you work with them via the API.)

When you are done with this, add the **Makefile**, **pipe-test.c** and **answers-prelab.txt** files to your **Lab1b** directory for later submission.

Problem 1 [6 points]

Copy the **pipe-test.c** program from the pre-lab to another file named **pipes.c**.

Now, modify your new program so that the parent process writes to the pipe **one character of the message at a time**. Obviously, you must use a loop to send the entire message in this fashion – if you consider that the message is a C "string," you can figure out what sentinel value you can use as the termination condition for the loop.

The message exchanged between the two processes is hard-coded into the program, but you should still test your program by experimenting with texts of different lengths (don't forget to adjust the buffer size!)

When the entire message has been sent, have the sender process close the file descriptor on the write-end of the pipe. **Closing the write end of a pipe will always cause an end-of-file (EOF) character to be sent to the reader on the other side.**

Next, modify the child process so that **it reads one character from the pipe at a time** and **writes each character individually to the standard output**. This will require another loop, but one that must terminate when the EOF character is received from the pipe (see the man page for **read** to understand how this system call reacts to receiving EOF).

Important: When you read the "**RETURN VALUES**" section of the man page for **read** and **write**, you will notice that when something goes really wrong, *"-1 is returned and the global variable **errno** is set to indicate the error."* Similarly to what you did in **Lab1a**, you will write a wrapper for the **pipe** to call **perror** and exit the function in case of error. Use the same prototype from the system call **pipe**, but call the function **Pipe**, that is:

int Pipe(int pipefd[2]);

From now on, your programs should always define a wrapper for system calls and library functions that set the **errno** variable. Use the **Fork** wrapper created in Lab 1a and create three additional wrappers for the system calls used in this problem following the function prototypes given below:

int Read(int fd, void *buf, size_t count);

int Write(int fd, const void *buf, size_t count);

When you are done with this problem, add **pipes.c** to your **Lab1b** directory for later submission.

Problem 2 [4 points]

Copy your **pipes.c** program to another file named **upper.c**. Modify your program so that **it defines two pipes**: one for communication from *parent to child* and a*nother for communication from child to parent*. Make sure to close the correct ends of each pipe. Ultimately, your goal is to have two pipes, one in each direction, so that you have bi-directional communication between the two processes. It helps a lot to use names for the pipes file descriptors that indicate their direction. For instance: *p-to-c* and c-*to-p* tell us the processes that the pipes interconnect and the direction of the flow of information.

This new version of the program must behave as follows.

- The parent sends a message to the child (byte-by-byte as in Problem 1) and when it is done, it enters a loop to read characters from the pipe coming from the child (also byte-by-byte), terminating the loop on the receipt of EOF.
- The child receives the message also byte-by-byte, printing each character to standard out as it arrives. For each character received, the child converts it to uppercase (using **toupper**; make sure to read its man page) and sends it back to the parent using your second pipe. As the parent reads the characters received from the child, it prints them to standard out.

Make sure to reason carefully about when the write ends of the two pipes should be closed, so that your processes can terminate their loops gracefully and reach their termination state when appropriate. Also, **make sure to use the wrappers defined previously (Fork, Pipe, Read, and Write).**

When you are done with this problem, add **upper.c** to your **Lab1b** directory for later submission.

Problem 3 [4 points]

Copy your **upper.c** program to another file named **tokens.c**. Your parent process will work on an *infinite loop,* in which it reads a line from the standard input that will contain various words (or *tokens*) separated by one or more blank spaces. After reading an entire line, the parent process sends it to the child process, which will work to substitute the possibly multiple instances of a space by a single instance of a space. For instance, if the child process receives a C string like:

```
“This     is    a test    of     the alert               system”
```

it sends back to the parent the C string:

“This is a test of the alert system”

The communication between parent and child processes is implemented by one pipe in each direction, as in Problem 2. The child process will now *eliminate the repetitions of blank spaces* instead of converting to uppercase the characters it receives.

Although there are different ways in which you can implement this functionality in the child process, your solution will rely on two library functions: **strtok** and **strcat**. If you read its **man** page, you will see that the **strtok** function *tokenizes* the line it receives. That is, given a line, each invocation of **strtok** returns to the caller the next token it finds, skipping over one or multiple occurrences of a chosen delimiter character (space, in this case). If you repeatedly call **strtok** to extract tokens one at a time, you can progressively build a cleaned-up line using **strcat** to concatenate each new token into an “accumulator” string.

Important: in this new program, your communications between parent and child process must not happen one byte at a time. Instead, the sending process will write on the pipe the complete C string using a single invocation of **write** for all the characters in the string (including the NULL byte).

Do the best you can in reading and interpreting the **man** pages for these two functions to learn how to use them. If you need clarifications on the logic for the parent and child processes, be sure to ask for them!

Note: reading a line of text that possibly contains white spaces in C isn't as straightforward as one might think. (Think of the functions or system calls you already know that you might use to read a line of text. Consider what these would read from the keyboard when the user types a space in between any two words.) It turns out that the **readline** function provided in our Linux system makes things easier. Read the manual page on this function and use it in this problem. One point this particular manual page doesn't raise is that in order to use the **readline** function in a program, one must link it with the library **readline** because it's not part of the GNU standard C library. (Add **-lreadline** to your gcc compilation command.)

Make sure to use the wrappers defined previously (Fork, Pipe, Read, and Write).

When you are done with this problem, add **tokens.c** to your **Lab1b** directory for later submission.

Important!

Extend the **Makefile** you started in the pre-lab to build the programs in problems 1, 2 and 3. Not having a working **Makefile** will incur a 2 point penalty in the overall grade for this week's pre-lab and lab.

When you are done with this problem, add the finalized **Makefile** to your **Lab1b** directory for later submission.

Hand In

As instructed throughout the lab create a **Lab1b** folder and submit there the following files: **Makefile**, **pipe-test.c**, **answers-prelab.txt**, **pipes.c**, **upper.c** and **tokens.c**.

Before turning in your work for grading, create a text file in your **Lab1b** called **submission.txt**. In this file, provide a list to indicate to the grader, problem by problem, if you completed the problem and whether it works to specification.

Finally zip your **Lab1b** folder and submit the compressed file to Brightspace for grading. Total of **17 points**.

Last updated 2.1.2022 by T. O'Neil. This lab was originally developed by Prof. L. Felipe Perrone (Bucknell).

Rubric

1. Using the **Fork** wrapper and writing the three additional ones [10 points]. Implementing the correct functionality for sending and receiving messages byte-by-byte over the pipe [20 points].
2. Using the wrappers defined previously, implementing the correct functionality for sending and receiving messages byte-by-byte over the pipe and using the child process to do the case conversion for the characters in the message [20 points].
3. Using the wrappers defined previously, implementing the correct functionality for sending and receiving messages between processes according to the required protocol, implementing the correct functionality for tokenizing and rebuilding strings. [20 points].
4. Not providing a correct **Makefile** to build all three programs in lab and pre-lab. [up to -10 points].