Lab 6

Lab 6 is built on top of the code infrastructure of Lab 5, i.e., the "shell". Naturally, you are expected to use the code you wrote for the previous lab.

Motivation

In this lab you will enrich the set of capabilities of your shell by implementing **input/output redirection** and **pipelines** (see the reading material). Your shell will then be able to execute non-trivial commands such as **"tail -n 2 in.txt| cat > out.txt"**, demonstrating the power of these simple concepts.

Lab 6 tasks

Task 0a

Pipes

A pipe is a pair of input stream/output stream, such that one stream feeds the other stream directly. All data that is written to one side (the "write end") can be read from the other side (the "read end"). This sort of feed becomes pretty useful when one wishes to communicate between processes.

Your task: Implement a simple program called **mypipe**, which creates a child process that sends the message "hello" to its parent process. The parent then prints the incoming message and terminates. Use the **pipe** system call (see man) to create the pipe.

Task 0b

Here we wish to explore the implementation of a pipeline. In order to achieve such a pipeline, one has to create pipes and properly redirect the standard outputs and standard inputs of the processes.
Please refer to the 'Introduction to Pipelines' section in the reading material.

Your task: Write a short program called **mypipeline** which creates a pipeline of 2 child processes. Essentially, you will implement the shell call **"ls -l | tail -n 2"**.
(A question: what does "ls -l" do, what does "tail -n 2" do, and what should their combination produce?)

Follow the given steps as closely as possible to avoid synchronization problems:

1. Create a pipe.

2. Fork to a child process (child1).

3. On the child1 process:
    1. Close the standard output.
    2. Duplicate the write-end of the pipe using **dup** (see man).
    3. Close the file descriptor that was duplicated.
    4. Execute "ls -l".
4. **On the parent process: Close the write end of the pipe.**
5. Fork again to a child process (child2).
6. On the child2 process:
    1. Close the standard input.
    2. Duplicate the read-end of the pipe using **dup**.
    3. Close the file descriptor that was duplicated.
    4. Execute "tail -n 2".
7. **On the parent process: Close the read end of the pipe.**
8. Now wait for the child processes to terminate, in the same order of their execution.

Mandatory Requirements

1. Compile and run the code and make sure it does what it's supposed to do.
2. Your program must print the following debugging messages if the argument -d is provided. All debugging messages must be sent to stderr! These are the messages that should be added:
    - On the parent process:
        - Before forking, "(parent_process>forking…)"

        - After forking, "(parent_process>created process with id:)"
        - Before closing the write end of the pipe, "(parent_process>closing the write end of the pipe…)"
        - Before closing the read end of the pipe, "(parent_process>closing the read end of the pipe…)"
        - Before waiting for child processes to terminate, "(parent_process>waiting for child processes to terminate…)"
        - Before exiting, "(parent_process>exiting…)"
    - On the 1st child process:
        - "(child1>redirecting stdout to the write end of the pipe…)"
        - "(child1>going to execute cmd: …)"
    - On the 2nd child process:
        - "(child2>redirecting stdin to the read end of the pipe…)"
        - "(child2>going to execute cmd: …)"

3. How does the following affect your program:
    - Comment out step 4 in your code (i.e. on the parent process:**do not** Close the write end of the pipe). Compile and run your code. (Also: see "man 7 pipe")

- Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.

- Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

Task 1

Redirection

Add standard input/output redirection capabilities to **your** shell (e.g., **"cat < in.txt > out.txt"**). Guidelines on I/O redirection can be found in the reading material.

Notes:

- The **inputRedirect** and **outputRedirect** fields in cmdLine do the parsing work for you. They hold the redirection file names if exist, NULL otherwise.

- Remember to redirect input/output only in the child process. We do not want to redirect the I/O of the shell itself (parent process).

Task 2

Go back to your shell and add support to a **single pipe**. Your shell must be able now to run commands like: `ls|wc -l` which basically counts the number of files/directories under the current working dir. The most important thing to remember about pipes is that the write-end of the pipe needs to be closed in all processes, otherwise the read-end of the pipe will not receive EOF, unless the main process terminates.

Task 3

Having learned how to create a pipeline, we now wish to implement a pipeline in our own shell. In this task you will extend your shell's capabilities to support pipelines that consist of an unlimited number of processes. To achieve this goal, you will use a set of helper functions.

Notes:

- The line parser automatically generates a list of cmdLine structures to accommodate pipelines. For instance, when parsing the command **"ls | grep .c"**, two chained cmdLine structures are created, representing **ls** and **grep** respectively.
- Your shell must support all previous features, including input/output redirection. It is important to note that commands utilizing both I/O redirection and pipelines are indeed quite common (e.g. **"cat < in.txt | tail -n 2 > out.txt"**).
- As in previous tasks, you must keep your program free of memory leaks.

The following helper functions will help you implement full support for multiple pipes and can be found on this Link

- *int **createPipes(int nPipes)*
  This function receives the number of required pipes and returns an array of pipes.
- *void releasePipes(int **pipes, int nPipes)*
  This function receives an array of pipes and an integer indicating the size of the array. The function releases all memory dedicated to the pipes.
- *int *leftPipe(int **pipes, cmdLine *pCmdLine)*
  This function receives an array of pipes and a pointer to a cmdLine structure. It returns the pipe which feeds the process associated with the command. That is, the pipe that appears to the left of the process name in the command line.
  For example, the left pipe of process **tee** in pipeline **"cat | tee | more"** is the first pipe. If the command does not have a left pipe (as with **cat** in our example), the function returns NULL.
- *int *rightPipe(int **pipes, cmdLine *pCmdLine)*
  This function receives an array of pipes and a pointer to a cmdLine structure.

It returns the pipe which is the sink of the associated command. That is, the pipe that appears to the right of the process name in the command line.
For example, the right pipe of process **tee** in pipeline **"cat | tee | more"** is the second pipe. If the command does not have a right pipe (as with **more** in our example), the function returns NULL.

In order to implement the functions `leftPipe` and `rightPipe` it is advisable that you will use the `idx` field in the `cmdLine` struct.

Here is a recommended test:

1. Parse **"ls | tee | cat | more"**.
2. Create an appropriate array of pipes.
3. Validate that both the left pipe of **ls** and the right pipe of **more** are NULL.
4. Print the file descriptors of both the left & right pipes of **tee** (4 file descriptors in total).
5. Release the pipes.

Submission

Submit a zip file with the following files: task1.c, task2.c task3.c, and a makefile to compile them all.