Lab 7: Reading Material

Standard Input/Output Redirection

Standard input and output are the "natural" feed (input) and sink (output) streams of a program: User input is read from the keyboard, and the program's output is displayed on monitor. However, in many cases one would like to read input from a file, rather than from keyboard. Similarly, one may wish to store the output in a file, rather than display it on monitor. Both requirements can be met by standard input/output redirection, without changing the code of the original program.

As a convention, input redirection is triggered by adding "<" after the invoked command. For instance, **"cat < my.txt"** tells the shell program to redirect the input of **cat** to file **my.txt**. As a result, cat displays the content of **my.txt** instead of displaying user feed from the keyboard. Output is triggered by adding ">", for instance: **"ls > out.txt"** which stores the list of files in the current directory in **out.txt**. Combining input and output redirection is also possible, e.g. **"cat < my.txt > yours.txt"**, which stores the content of **my.txt** in **yours.txt**.

How does the shell program achieve such an effect? Simply by closing the standard input stream (file decriptor 0) or the standard output stream (file decriptor 1), and opening a new file, which in turn is automatically allocated with the lowest available file-descriptor index. This effectively overrides stdin or stdout.

Introduction to Pipelines

Pipelining is an extremely important concept in system programming. The name "pipeline" itself suggests that pipes are, somehow, involved. But what exactly is a pipeline? This is most easily explained by example, and in our case, by typing **"echo spider-pig | head -c 6"** in the Linux shell.

The output of this command, as you will notice, is the string **spider**. However, if you execute **"echo spider-pig"** and **"head -c 6"** separately, you see something quite different. The first command simply echos **spider-pig**, and the second command asks for user input, and prints the first 6 characters. One can easily deduce that when the two commands are chained using the "|" symbol, the output of **echo** is directly fed into **head**.

And this is where pipes come into play. The two processes communicate using a pipe which they both share: The standard output of **echo** is redirected to the pipe's "write-end", and the standard input of **head** is redirected to the "read-end". A pipeline, then, is simply a chain of processes where the standard output of one process feeds the standard input of the following process. The principle of pipelines easily generalizes to an unlimited number of processes. For instance, the command **"echo spider-pig | cat | head -c 6"** entails a pipeline consisting of three processes (**echo**, **cat** & **head**) and two pipes.

MAN: Relevant Functions and Utilities

hexedit(1)
readelf (1)
dup (2) , dup2 (2)
fork (2), execvp(3)

Lab 7

Lab 7 is built on top of the code infrastructure of Lab 6, 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 7 tasks

Task 0

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 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

Note
Task 2 is independent of the shell we revisited in task 1. You're not allowed to use the LineParser functions in this task. However, you need to declare an array of strings containing all of the arguments and ending with 0 to pass to execvp() just like the one returned by parseCmdLines().

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 3

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 4 - Bonus Task

This task is bonus. You are not required to complete it.

Note

We again revisit our shell program. Namely, the code you write in this task is added to the shell.

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 first implement a set of helper functions. Once completed, you will use these functions in your final implementation.

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.

Task 4a

Implement the following helper functions:

- *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.

You must test the helper functions by yourself before asking for the teaching assistant's acknowledgement.
Failing to do so will trigger an automatic grade deduction.

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.

Task 4b

Finally, extend your shell's capabilities to support an unlimited number of processes.
For example, if the command **"ls | grep .c | tail -n 2"** is entered, you are expected to create three processes and two pipes. Each successive pair of child processes will be connected by one pipe, similarly to the technique presented in Task 2. The first pipe allows the **ls** process to communicate with the **grep** process, and the second pipe allows **grep** to communicate with **tail**.

Submission

Submit a zip file with the following files: task1.c, task2.c task3.c, task4a.c, task4b.c, and a makefile to compile them all.