Lab 6

Lab 6 in 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.

Deliverables
You should read and understand the reading material and do task 0 before attending the lab. To be eligible for a full grade, you must complete tasks 1 and 2 during the regular lab, and task 3 during the completion lab (or the regular lab).

Lab 6 tasks

Task 0

Pipes

A pipe is a pair of input stream/output streams, 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"), and vise versa. This sort of feed becomes pretty useful when one wishes to communicate between processes.

Your task here is to 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 (man) to create the pipe.

In this lab, we will be implementing a **pipeline** (see reading material), which consists of one or more pipes, similar to the one you've just learned to use. However, pipelines also require redirection of the standard input/output streams, that you will implement in the next task.

Task 1

Redirection

Add a new functionality to the shell we produced in Lab 5, **standard input/output redirection**, as explained 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.

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.

Here we wish to explore the implementation of a pipeline. Reminder: A pipeline is a chain of processes piped by their standard streams, such that each process has its standard input piped with the standard output of its predecessor. If we look at the pipeline x | y | z, we can infer that:

1. Process x's standard output is piped with process y's standard input
2. Process y's standard output is piped with process z's standard input

In order to achieve such a pipepline, one has to create pipes and redirect the standard outputs and standard inputs of the processes. In the shell syntax, the command "x | y | z" does exactly that (we will implement this in task 3).

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

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:

- Close the standard output
- Duplicate the write-end of the pipe using **dup** (see man)
- Close the file descriptor that was duplicated
- 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:
   - Close the standard input
   - Duplicate the read-end of the pipe using **dup**
   - Close the file descriptor that was duplicated
   - Execute "wc -l"
7. On the parent process: close the pipe input.
8. Now wait for the child processes to terminate, in the same order of their execution.

Task 3

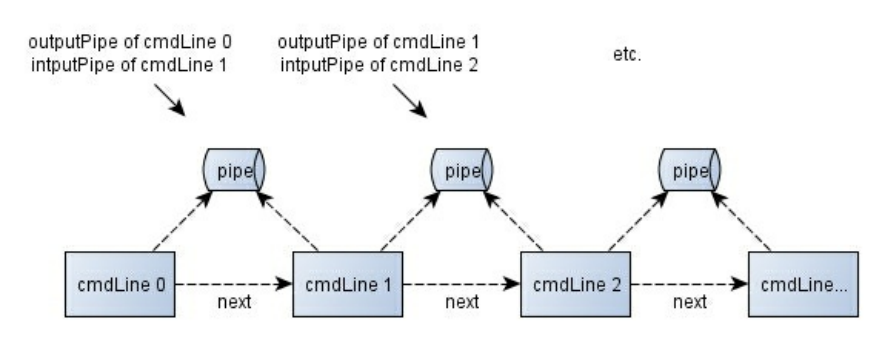
Note
We again revisit our shell program. Namely, the code you write in this task is added to the shell.

Having learned how to create a pipeline, we now wish to implement a pipeline in our own shell. Note that the **parseCmdLines()** function automatically creates a chain of cmdLine once given the appropriate input. Moreover, **parseCmdLines()** already allocates memory for the pipes, although you will still be required to create the pipes in the designated places.

For instance, the command **"ls | cat | grep .c"** will automatically create 3 chained cmdLine structures that represent **ls**, **cat** and **grep .c** respectively, and two memory units, designated for the creation of two pipes.

- The outputPipe of **ls** and the inputPipe of **cat** inputPipe will automatically point to the same address.
- The outputPipe of **cat** and the intputPipe of **grep .c** will automatically point to the same address.

As depicted by the following diagram:



Here is a scheme of such an implementation. It is recommended that you follow it closely.

1. Write a **createPipes(cmdLine*)** function that creates a pipe for each pair of adjacent processes, in the designated places.
   Remember: one command's output pipe is the next command's input pipe.
2. Invoke createPipes() right after invoking parseCmdLines().
3. Implement the same redirection framework as in Task 2.
   Important: remember that the parent process must close the write-end and read-end of the pipe, once used by the child process.

4. Extend the **execute(cmdLine* pCmdLine)** function to execute all the chained entries, instead of just the first one.
5. Now test your shell with "ls | cat | grep .c | wc -l".