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. As a final task, we will also add a basic history mechanism.

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.

History

Check out the "history" mechanism in the Linux shell. For example, see this link and try it out in a Linux shell.

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 a first child process (child1).
3. In 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. **In the parent process: Close the write end of the pipe.**
5. Fork a second child process (child2).
6. In 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. **In 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 is 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:

    - In 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…)"
    - In the 1st child process:
        - "(child1>redirecting stdout to the write end of the pipe…)"
        - "(child1>going to execute cmd: …)"
    - In 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:
    1. 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")
    2. Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.
    3. Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

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. In this task you will extend your shell's capabilities to support pipelines that consist of just one pipe and 2 child processes.

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.

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 4

Here you will add a history mechanism to your shell. The history mechanism works as follows. Your shell should keep HISTLEN previous command lines in a queue, where HISTLEN is a constant with a value of 20 as a default. The history list is maintained in an array of size HISTLEN of pointers to (copies of) previous commands. Note that you need to allocate space for these copies. Also note that you should keep the UNPARSED command lines in the history list, and NOT the parsed version.

When a new command line is entered after the history list is full (already has HISTLEN entries), delete the oldest entry and insert the new one. You should implement the history as a circular queue, using "newest" and "oldest" indices.

The user can now perform the following functions as a shell command (not a process!):

- "history": print the history list (number of entry in the array and the appropriate command line), for all valid entries.
- "!!": retrieve the last command line (non-history, for clarification please refer to lab reading material) CL, enter CL again into the queue, and execute it (needs to be parsed again!).
- "!n": With n a number between 1 and HISTLEN, as in "!!" except with CL being the command line at history index n. If n is an invalid number (out of range, or no such entry yet) print an error message to stdout and ignore this command line.

Note that your shell should support history on top of all the other features: pipes, redirection, etc. This should not be hard if your code is well-designed.

Submission

Submit a zip file (named either [student-id-num].zip [student1-id-num_student2-id-num.zip] in case of pair submission) with the following files: task1.c, task2.c task3.c, task4a.c, task4b.c, and a makefile to compile them all.

Task 4 may be done in a completion lab without grade penalty if you run out of time during the regular lab.