Lab 6

In this lab, you should perform **task 0 before attending the lab session**. This lab may be done in pairs (members of a pair must be in the same lab group, and should inform the TA so as to be placed in the same breakout room)

> 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

Through out the tasks of the lab, you are asked to add debug messages, according to the specifications if given, and as you see fit where no specification is provided.

Task 0a

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"). 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

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 the redirection exists, 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 1

> **Note**
> Task 1 is independent of the shell we revisited in task 0. 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:
    a. Close the standard output.
    b. Duplicate the write-end of the pipe using **dup** (see man).
    c. Close the file descriptor that was duplicated.
    d. 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:
    a. Close the standard input.
    b. Duplicate the read-end of the pipe using **dup**.
    c. Close the file descriptor that was duplicated.
    d. 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:
   a. 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")
   b. Undo the change from the last step. Comment out step 7 in your code. Compile and run your code.
   c. Undo the change from the last step. Comment out step 4 and step 8 in your code. Compile and run your code.

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.

Make sure that I/O redirections work together with pipelines i.e. the following commands
`cat < input.txt | grep a > output.txt`

Task 3

Add a "history" command which prints the list of all the command lines you've typed, in an increasing chronological order. Namely, if the last command typed is "ls", then "ls" is the last command to appear. A printout of N commands should consist of N lines - numbered from #0 (the first) to #N-1 (the last).

You can partially test your code by using this input scenario:

```
ls -l
echo hello world
cd .
history
ls -l
history
```

> Use an array to store your list. You may assume history to contain at most 10 commands, but do not use the number in your code more than once (use #define).

Task 3 b

History list is useful if one wishes to re-use previously typed commands. To use a previously typed command in say, index #0, one should invoke the command "!0". Likewise for 1,2 etc.

Add the "!" command implementation, as explained above. **Make sure the history list adds the reused command line as it was initially entered**. That is - if the command "ls -l" currently occupies index #2 in the log list, then typing "!2" should: (i) Invoke the "ls -l" command, and (ii) Add "ls -l" to the bottom of the log list.

You can partially test your code by using this input scenario:

```
ls -l
echo hello world
cd .
!0
!1
history
```

Remember to **print an error message when a non-existing log index is invoked** (e.g. when you have only 5 history entries, "!5" should trigger an appropriate error message).

Deliverables:

Tasks 1,2 must be completed during the regular lab. Task 3 may be done in a completion lab, but only if you run out of time during the regular lab. The deliverables must be submitted until the end of the lab session.
You must submit source files for task 1, task 2, task 3 and a makefile that compiles them. The source files must be named task1.c, task2.c, task3.c and makefile

Submission instructions

- Create a zip file with the relevant files (only).
- Upload zip file to the submission system.
- Download the zip file from the submission system and extract its content to an empty folder.
- Compile and test the code to make sure that it still works.

Example structure of zip, + is a directory and - is a file:

+ t1
  - makefile
  - task1.c
+ t2
  - makefile
  - task2.c
+ t3
  - makefile
  - task3.c