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 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 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.

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:
    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

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.

All processes that are run from the same command (connected with pipes) have the same process group (and so, they belong to the same job). This is why you were told to wait for **any process in the process group** to change state and **not only for a process pid** (this replaces step 8 in task2). So both ls and wc -l from the example above have the same process group, and it's usually set according to the pid of the first process (ls in this case). Make the necessary changes in your shell to support job control of commands containing pipes.

Task 4

Here we wish to emulate the internal variables environment of the command shell. Briefly, the internal variables environment is a list of string pairs (name, value), which associates names with values. When variable names appear in the command line with a "$" prefix, they are replaced with their associated value.
For instance, let variable **i** be mapped to value **Hello**. "`echo $i`" will then print "Hello", whereas "`echo i`" will print "i". Similarly, the command "`ls $i`" will be translated to "ls Hello", etc.

The task is divided into four mini-tasks, which define operations relevant to the internal variables environment. Although the environment can be implemented in several ways, **you are strictly required to base your implementation on a linked list of (name, value) string pairs**. Your implementation should be general enough to support unlimited number of variables and unlimited length of names and values.

1. Add an set command to the shell, which associates a given name with a given value. **If the name already exists, the command should override the existing value**.
   Usage: "set x y" creates an environment variable with name x and value y.

2. Add an env command, which prints all current associations in the environment.
   Usage: "env" prints out all environment associations.

3. Activate the internal variables in each executed command line, by replacing each argument that starts with a **$** sign with its proper environment value. **Write an appropriate error message when variables are not found**.

4. Add a delete command, which delete a variable from the environment. **Write an appropriate error message when variables are not found**.
   Usage: "delete x" deletes the variable x from the environment.

5. Adding support of `~` for `cd`. Initiating the command `cd ~` in your shell will result in an `error`. You need to add support for converting ~ to your home directory. Find the environment variable for your home directly, **do not hard code it**.

You will need to propagate the error messages to stderr.

- Deleting an environment variable that does not exist.
- Activating an environment variable that does not exist.

Deliverables:

Tasks 1,2,3 must be completed during the regular lab. Task 4 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 day.

You must submit source files for task 1, task 2, task 3, and task 4 and a makefile that compiles them. The source files must be named task1.c, task2.c, task3.c, task4.c, and makefile