Programming Assignment (PA) - 1
(Shell Command Execution Simulation in C)

CS307 - Operating Systems

22 March 2023
DEADLINE: 03 April 2023, 23:55

Pipes help combine two or more commands and are used as input/output concepts in a command. the output of one command acts as input to another command, and this command's output may act as input to the next command and so on.This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen.
Pipes are unidirectional i.e data flows from left to right through the pipeline.

1 Motivation

As Computer Scientists/Engineers, you will constantly learn about new tools. While Google is always helpful, if you are working with UNIX systems, there is another great utility. The `man` command, which stands for *manual files* have information about many UNIX commands, including their options and arguments. Whenever you feel unsure about a command, manual pages provides you a quick tour of learning/remembering.

However, `man` pages can be a bit overwhelming. There could be tens of options for a simple command with long and precise explanations. If you are interested in only one of these options, it can be difficult to locate that option in the `man` page. Here, *grep* comes to our rescue. *grep* allows us to get the specific lines that we are interested in. In general, grep command is used for finding text fragments in text files matching to provided input patterns.

Your task requires combining `man` and `grep` commands using UNIX pipes. In `Unix`, commands can be combined by directing the output of first command (which is normally printed to the console) to the input of the second command (which normally expects an input from the console). The user provides piped commands in the following form to the shell:

$$< command_1 > | < command_2 >$$

In this setting, $command_1$ is executed first as a process. When it terminates, its output is directed to its child process as the input. Then, the child process executes $command_2$ using this input.

For this assignment, you are expected to write a C program that simulates piped execution of `man` and `grep` commands on the shell. For this purpose you will use *fork*, *wait* and *exec* system calls we have seen in the lectures. In addition, you will need to learn and use *pipe* system call yourselves for piping the commands. Details about the program, command structure and system calls are provided in the following sections.

2 Problem Description

For this programming assignment, you are asked to implement execution of a shell command admitting the following pattern:

$$\texttt{man} < \textit{command} > \texttt{ | grep} < \textit{option} > \texttt{ > output.txt} \tag{1}$$

Here, *command* and *option* are variables you will determine as arguments to the `man` and `grep` commands, respectively. Once they are fixed and the above command (1) is executed on the shell, it prints lines in the manual page of the *command* that contain the string *option* to the `output.txt`. You have to be careful about how to interpret the *option* variable. It is not an option for the `grep` command. It should be an option of the *command* that `grep` accepts as the search pattern argument. Also, you must determine the number of lines the `grep` command will pick when it finds a matching string. For most options, the explanation in the man page spans multiple lines. So, you must obtain all of them.

Rules and explanations on how to choose values for *command* and *option* variables are explained in Section 3. After you fix these values, the piped command you will simulate will be determined. You are expected to implement your simulation in a single C file called `pipeSim.c`. Guides and details about the implementation are presented in Section 4.

3 Command Formation & Argument Selection

Before implementing your simulation, you have to pick values for *command* and *option* variables.

Once you fix these variables, your shell command will be determined. In order to inform us about the command you are simulating, you must write the exact command in a file called `command.txt`. This file will contain a single line which is the command. The command will be approximately in the form of (1).

Your implementation might require adding more options to `man` or `grep` commands. For instance, if you enforce `grep` to take the next three lines after it finds a matching string, you must reflect it to the command in the `command.txt` as well.

While evaluating your submission, we will first execute the command in your `command.txt` in BASH, then compile and run your simulation program in BASH and lastly compare the results in both `output.txt` files.

While deciding on variables of the piped command, you should first fix the *command* variable. You can select one item from the list below. They are some of the most useful command-line utilities for UNIX systems. You can look at their `man` page and use Google to determine which is the most interesting one for you. After fixing the *command*, you will select an *option* of the *command*. All the options of the *command* can be found under the `man` page of *command*.

- ls
- rm
- touch
- find
- grep
- df
- diff
- ping
- wget
- top

Example: Here is an example where we selected a command that is not in the list

selected command: `kill`
selected flag (option) for `kill`***:*** `-L`
command.txt:

```
1 man kill --extraOptions | grep "-L" --extraOptions > output.txt
```

Here, *−extraOptions* are again variables, you might want to use to make sure that your implementation exactly simulates the piped command.

In your submission, we expect to find a `pdf` file called `report.pdf`. In this file, you will explain what your command and option does, and why they attracted your attention. Here is an example:

```
1 kill command kills (terminates) a process via sending
2 various signals to the Operating System. –L flag is
3 for listing the available signals in a nice table.
4 I picked this command and option because ...
```

3.1 Niche Problems & Corner Cases

When you execute `man kill | grep "-L"`, it will not successfully *grep* "-L" from *kill* command's *man* page. The reason is, the character {-} is a **special character**, so the shell does not interpret this as a part of the search query. You have to solve this problem yourselves.

Again, let's focus on the given example command `man kill | grep "-L"` for demonstration purposes. After you solve the special character problem, you will see that only the first line of the definition of flag `-L` will be displayed in the console as the output. The problem here is, actual definition was consisting of 2 lines.

actual definition:
-L, –table
List signal names in a nice table.
grep result:
-L, –table

Basically, the second line of the definition was ignored, since the second line was not including the string *-L*. We expect you to find a solution to this, and print all the necessary lines to output full definition. You do not have to automatically infer the number of lines you need. Once you decide on the *command* and *option*, you can look at the `man` page of the *command*, see how many lines the *option* description occupies and directly use this number in `command.txt` and your implementation.

4 C Program Implementation

In `pipeSim.c` file, you will implement the simulation of the piped command. In Chapter 5 of the OSTEP book, `p3.c` and `p4.c` are example command execution simulations of the shell. You should fully understand these programs and the chapter to understand the nature of the shell. Your implementation must agree with the structure of these programs.

Difference of your implementation from the examples in the book is that your implementation will simulate the execution of two commands that are connected. Similar to examples, each command must execute in a separate process. These processes must be descending[1] from the initial process which will simulate the shell. The initial process should wait until the piped command finishes its execution as the shell would do. The shell process should not contain the bug we have seen in the lecture. It should not print to the console until all the processes descending from it terminate.

In order to implement your simulation as described, you must use *fork*, *wait* and *exec* system calls as in the examples. You should describe the process hierarchy generated by your program in your report (`report.pdf`).

In addition to these system calls, you have to utilize *pipe* system call for interprocess communication. Basically, *pipe* takes two file descriptors as input and connects the parent and child processes using these descriptors. The parent process' descriptor is in the writing mode and the child's descriptor is in reading mode. Using these descriptors, the parent can send its output to the child as the input. You can get more information about *pipe*s on this page.

We want to ensure that your program allows correct interleaving of the processes. By correct interleaving we mean that the initial process that represents the shell should wait until all of subprocesses that execute commands`man` and `grep` finish. Therefore, each process should print their process IDs to the **console (not to any file)**. Only the shell process should print second time after the piped command execution finishes. Below is an example output in the console with correct print ordering:

[1] They are children or grand children or grand grand children or so on.

I'm SHELL process, with PID: 38810 - Main command is: <insert piped command here>
I'm MAN process, with PID: 38811 - My command is: <insert man command here>
I'm GREP process, with PID: 38812 - My command is: <insert grep command here>
I'm SHELL process, with PID:38810 - execution is completed, you can find the results in output.txt

5 Submission

You are expected to submit a zip file named *<YourSUUserName>_PA1.zip* until 1 November 2022, Tuesday, 23:55.

The content of the zip file is as follows:

- `command.txt`: The piped command your program simulates. The command here will be used for evaluation.
- `report.pdf`: Your report that explains your choices on variables and process hierarchy of your program.
- `pipeSim.c`: Your C implementation.
- You don't need to submit the generated output files (`output.txt`).
- Make sure that you are submitting your zip file in correct form. If your submission doesn't obey the submission criteria, we will deduce 10 points (both for wrong naming and multiple file submissions).

6 Grading

Some parts of the grading will be automated. If automated tests fail, your code will not be manually inspected for partial points. Some students might be randomly called for an oral exam to explain their implementations and reports.

Submissions will be graded over 100 points:

1. **Compilation (10 pts):** Your program compiles, runs and terminates without an error.
2. **Process Hierarchy (20 pts):** Process hierarchy described in your report represents correct shell simulation. Your description in the report must match with the reality (your C implementation).

3. **Successful `man` (15 pts):** Your `command.txt` contains just a `man` command without `grep` and the output of executing this command matches with the output of your implementation.
4. **Successful piped command (40 pts):** Your `command.txt` contains a piped command (both `man` and `grep`) and the output of executing this command matches with the output of your implementation.
5. **Correct interleaving (20 pts):** When your implementation is executed, lines printed to the console have the same order with the example execution presented at the end of Section 4.
6. **Correct `grep` option (5 pts):** Your implementation and the command in `command.txt` solves the special character problem described in Section 3.1.
7. **Correct number of lines (5 pts):** Your implementation and the command prints all the description lines for the given *command*, *option* pair as described in Section 3.1.

The first criterion is a precondition for all the other criteria. Similarly, the fourth item is a precondition for items 5, 6 and 7. It means that you will not get any credits from these items if you cannot get full credits from item 4. Items 3 and 4 are mutually exclusive. You can get points from only one.

If you are submitting your file in wrong format, you will lose 10 points.

Important Note: In your implementation, you are not allowed to use `sleep` system call. If your code contains sleep statements, we will comment them out during the evaluation.