Programming Asssignment (PA) - 2
(Synchronizing the CLI Simulator)

CS307 - Operating Systems

13 April 2023
DEADLINE: 26 April 2023, 23:55

1 Introduction & Problem Description

In your first programming assignment you simulated a shell program which executed two commands with a piping system. Now assume a more generalized version of this program where your shell program is reading commands line by line from a file called commands.txt and executing them.

In this Programming Assignment, on top of building this generalized shell program, you are also expected to fix a concurrency problem that might occur in this generalized version of the shell when multiple commands try to print lines to the console concurrently.

In order to understand the problem, consider the following input `"commands.txt"` file to your shell simulator:

```
1 ls -l &
2 wc output1.txt &
```

While your shell simulator is processing this file, the shell process creates two children command processes that execute `ls` and `wc` commands. Since both commands are background jobs due to the `&` sign at the end, they can execute concurrently (co-exist and run at the same time). Therefore, lines printed by these processes can intervene with each other and produce garbled console output. For instance, after the shell process terminates, you might have the following end result at your console:

```
1 total 40
2 drwxrwxr-x 2 deniz deniz  4096 Mar 16 21:47 .
3 drwxr-xr-x 3 deniz deniz  4096 Mar 12 10:38 ..
4 -rwxrwxr-x 1 deniz deniz 16792 Mar 16 19:25 a.out
5  211  957 8048 output1.txt
6 -rw-rw-r-- 1 deniz deniz    73 Mar 16 16:22 commands.txt
7 -rw-rw-r-- 1 deniz deniz  5163 Mar 16 19:25 hw1.c
8 -rwx------ 1 deniz deniz  8048 Mar 16 21:47 output1.txt
```

In this example output, line 5 is the output of the `wc` command and rest of

the lines are printed by the `ls` command. As you can see, this output is undesirable because it is difficult to identify and follow lines of a particular command. This undesirable result occurs because the `ls` and `wc` commands are executed concurrently. The OS first schedules the `ls` command process. It prints the first 4 lines. Then, a context switch happens and the `wc` command process prints Line 5 and terminates. Finally, `ls` process is scheduled again and it prints the remaining lines (6 to 8).

In the scope of PA2, you are expected to build a shell simulator which reads line by line and executes commands from commands.txt and prints its output in a way that lines printed by a command cannot be interrupted. Considering the previous `"commands.txt"` example, after your modifications, there are only two possible console output you might observe:

```
1  211  957 8048 output1.txt
2 total 40
3 drwxrwxr-x 2 deniz deniz  4096 Mar 16 21:47 .
4 drwxr-xr-x 3 deniz deniz  4096 Mar 12 10:38 ..
5 -rwxrwxr-x 1 deniz deniz 16792 Mar 16 19:25 a.out
6 -rw-rw-r-- 1 deniz deniz    73 Mar 16 16:22 commands.txt
7 -rw-rw-r-- 1 deniz deniz  5163 Mar 16 19:25 hw1.c
8 -rwx------ 1 deniz deniz  8048 Mar 16 21:47 output1.txt
```

```
1 total 40
2 drwxrwxr-x 2 deniz deniz  4096 Mar 16 21:47 .
3 drwxr-xr-x 3 deniz deniz  4096 Mar 12 10:38 ..
4 -rwxrwxr-x 1 deniz deniz 16792 Mar 16 19:25 a.out
5 -rw-rw-r-- 1 deniz deniz    73 Mar 16 16:22 commands.txt
6 -rw-rw-r-- 1 deniz deniz  5163 Mar 16 19:25 hw1.c
7 -rwx------ 1 deniz deniz  8048 Mar 16 21:47 output1.txt
8  211  957 8048 output1.txt
```

Note that this concurrency problem does not only occur among background jobs. A background job might be concurrent with a non-background job as well and produce a garbled output. For instance, if we remove the `&` at the end of Line 2 in the previous `"commands.txt"`, we could observe the same garbled output shown before. You must solve the problem for this case as

well.

Moreover, the same problem might occur for file streams other than the console. Consider the following small modification on the `"commands.txt"`:

```
1 ls -l > foo.txt &
2 wc output1.txt > foo.txt &
```

In this case, after executing your shell simulator assuming you haven't implemented concurrency when printing the output lines, the same garbled output we have seen in the console might have been observed in `"foo.txt"`. However, for the scope of this PA, we ignore the concurrency problems in file redirections. **You do not have to synchronize the outputs that will be printed to a file instead of the console**.

So, it could be said that there are mainly two parts for this programming assignment. First you have to build a shell simulator which reads line by line from a file called commands.txt, executes the commands inside commands.txt and prints its output accordingly. Second, and more importantly, we want you to write this shell program in a way that it obeys concurrency and that the outputs of these multiple commands do not intervene each other.

1.1 Generalized Shell Process

1.1.1 Brief Overview

You are expected to implement a C/C++(You choose) program that provides a subset of functionalities of a typical UNIX Shell. However, your shell does not take its input from the console. The commands are provided in a file called `"commands.txt"`. In this file, each line corresponds to a single shell command simulating a command entered by the user through the console. For each command, you have to first decode (parse) it and then create a new child process (using the `fork` system call) and execute this command on the child process (using the `execvp` command). The child processes must terminate as soon as the associated command finishes its execution. The

parent process (shell process) must terminate after all the commands in the `commands.txt` terminates. If the command does not contain an & at the end, the shell process must wait until the command terminates.

In order to give more detailed information about your implementation, we first need to provide the format of commands you should expect in `"commands.txt"` file.

1.1.2 Command Format

Each line in `"commands.txt"` is a shell command obeying the following format:

$$cmd\_name\ [input]\ [option]\ [> \mid <\ file\_name]\ [\&]$$

In this format, parts inside brackets ("[ ]") are optional and "|" corresponds to a logical `or` operation. For instance, the expression $[> \mid < file\_name]$ should be interpreted as $> file\_name$, $< file\_name$ or nothing. Then,

- *cmd_name* is one of the following commands: `ls`, `man`, `grep`, `wait`, `wc`.

  IMPORTANT NOTE: Please note that `wait` command that you will be implementing is different than the $wait(NULL)$ system call provided by the standard C/C++ libraries. The `wait` command/program you will implement waits for all background processes to finish. However, $wait(NULL)$ system call blocks the caller process until **one** of its children terminates. You have to implement the former (command/program) version.

- *input* is an optional input to the command. Possible inputs can be command names: ls, cd, man, grep, wait, wc etc. In the scope of this project, each command can take at most one input. For `grep` command, you can assume that only string pattern can be given as an input. You can also assume that search string is a single word(i.e. it doesn't contain white spaces.). The input file can be provided through redirectioning. For `wait` command, you can assume that it has no input since there is no way to know the `pid` of a background job before running the program. Only `wait` command does not take any inputs.

- *option* starts with a dash (-) symbol, followed by a single letter. After that there can be an optional string following a single white space. For instance, "`grep sample -i < cs.txt`" and "`ls -l`" are commands with valid options for this assignment. However, you should not expect commands like "`ls -la`", "`ls -l -a`" although they are perfectly valid commands for any other shell. Only `wait` command does not have any options.
- $file\_name$ is a valid path to an existing file. Note that the format allows at most one redirectioning per command. Either the input or the output stream can be redirected to a file, but not both. Only `wait` command cannot be combined with file redirectioning.
- Optional "&" operator at the end enforces shell to run this command at the background. Only `wait` command cannot be run in the background

Below are some sample input files with commands obeying the conditions above:

```
1 ls -a
2 man -f grep
3 grep sample -i < cs.txt
```

```
1 ls -l > out1.txt &
2 man cat
3 grep rwx < out1.txt &
4 wait
```

1.1.3 Parsing

You can safely assume that each line in `"commands.txt"` obeys the format explained in Section 1.1.2. In `"commands.txt"`, there will not be any blank lines. Also, for every command line, there will be exactly one blank space between every token. However, you still need to parse each line to extract commands, input, options, redirectioning and background operator information. For parsing, you can use the calls `strtok()` and `strchr()` if you plan

to submit a .c file, however if you plan to code in C++ you could use file streams. Check out this page to find out different string functions.

After parsing the command, the shell process must print the obtained information to a file called `"parse.txt"` in the following format:

First you need to output 10 dash(-) symbols to indicate that the main thread of the shell process is processing a new command from `"commands.txt"`. After that, you need to give information about the command, its input, its option, whether redirection is used or not and whether the command will work as a background job. Every information should be written in a new line. In the end, you need to output 10 dash(-) symbols again.

You only need to write the command name -without input and option- in the `"Command"` output. For `"Input"` and `"Option"` output, if there is no input or option given, you should not output anything specific. If there are not any redirection symbols in the command, you need to put a dash(-) symbol in the `"Redirection"` output. If there exists a redirection symbol, you need to output that symbol. `"Background Job"` part can only get two outputs: 'y' and 'n'. If there exists an ampersand(&) in the command, you need to output 'y'; otherwise 'n'.

For example, if the command is `"grep sample -i < input.txt &"`, then your shell process must print the following to the console:

```
----------
Command: grep
Inputs: sample
Options: -i
Redirection: <
Background Job: y
----------
```

You can check the sample runs(5) for more information.

Warning: Write operations to a file or to the console might not immediately take effect. The OS might wait them for a while and do batch processing for efficiency. We will talk about this in more detail during the *Persistence* section of the course. In order to print the formatted command information

to the console immediately, we recommend you to use `fsync()` or `fflush()` system calls after the print statements so that the effects of these print statements immediately becomes visible on the console. Otherwise, the OS might reorder some lines. Even if you use `fsync()` or `fflush()` after print statements, there might be some background jobs running concurrently with the shell process. Hence, their print statements might interleave with shell process' print statements and the console output might be garbled. This is OK for the scope of this PA. We will consider these possibilities while grading your PAs.

1.1.4 Implementing Command Executions

After parsing, your program (shell process) should *fork* a new child process called command process. If the command does not contain any redirectioning or background operator part, the command process puts parsed components of the command in an array. This array becomes the input for the `execvp` that is executed next by the command process. See the program `p3.c` in your textbook (Section 5.3 of OSTEP) which does a similar task.

There are corner cases you need to consider while giving arguments to `execvp`. Options start with a dash character (-). The dash character is a **special character** and must be **escaped** by using some other special characters. It is your task to identify and find ways to escape these characters. In the report you will submit, please write what are the special characters you identified and how you escaped them.

IMPORTANT NOTE: You are **not** allowed to use **system()** library function to create processes and run commands.

1.1.5 Implementing Redirectioning

If the command contains a redirectioning part like `"> file.txt"` or `"< file.txt`, your command process must redirect the output (resp., input) to the `"file.txt"` file. Note that you cannot do it by changing the command program like opening a file and then replacing every `printf` or `scanf`

statement with a `write` or `read` operation to a file. The only clean way you can do this is to first close the standard output (resp., input) and then open `"file.txt"` using the `open` system call. If you do these two steps consecutively, UNIX based OS will associate the `"file.txt"` with the file descriptor of the standard output (resp., standard input) of the console. See the program `p4.c` in your textbook (Section 5.4 of OSTEP) which performs a similar task. In your report, please explain how your program implements the redirectioning in a clear way.

Commands your program will execute might contain at most one redirectioning part. Only the input or the output could be redirected but not the both.

1.1.6 Implementing Background Jobs

If there is no & at the end of the command, then the shell process waits for the child process executing the current command to terminate. Otherwise it continues and fetches the next line from `"commands.txt"`.

If the current command is `wait`, the shell process does not create a child process for this command. It waits for **all** background jobs to terminate. In order to achieve this task, shell process should keep track of all the child processes that are executing the background commands. In your report, you are expected to explain your program does this bookkeeping.

When all commands inside `"commands.txt"` file is processed by the shell process, it must also wait for all the continuing background jobs to finish i.e., all the children processes to terminate.

1.2 Handling Concurrency - Print Synchronization

We consider the interleaving of the command outputs as a concurrency problem. Multiple programs try to print lines to the console at the same time and due to context switches their outputs interleave.

In the lectures, we solved this problem and established synchronization using

mutexes (locks). Basically, a thread might acquire a mutex before it starts printing and release the mutex after it is done with printing. Wrapping all of the `printf` statements with `lock()` and `unlock()` and using `fflush` or `fsync` system calls before `unlock()` ensures that lines printed by this thread cannot be interleaved by other threads' lines.

However, we cannot directly use mutexes for our problem. As we have seen in the lectures, mutexes are implemented based on shared variables. They can be efficiently used for ensuring synchronization among threads of the same process since they share the heap and the address space. However, mutexes can not provide synchronization among processes since processes do not share any state and each process has its own unique address space.

Unfortunately, our shell simulator creates a new process for each command. Hence, print statements that interleave come from different processes, not threads. Consequently, we cannot use mutexes immediately to solve our problem. The solution we suggest is to transform this interprocess concurrency problem into an intra-process or inter-thread concurrency problem so that we can use mutexes for synchronization.

Basically, our solution is to direct every console output of command processes to the parent shell process so that only one process becomes responsible for printing lines to the console. Then, we can use mutexes in the shell process for synchronizing the print statements.

Note that, after adding this new task (printing output of commands to the console) to the shell process, the shell process cannot stay single-threaded any more. It has to continue processing new lines from `"commands.txt"` by *forking* new processes for them while getting lines from older background processes and printing them to the console in a synchronized manner. We need to do this for efficiency and still keeping the background jobs meaningful. Otherwise, if the shell process stays single-threaded, then it can deal with only one process at a time. If a background job has some console output, then the shell process can only handle it and cannot create and run new concurrent commands.

Considering these factors, your program structure must be like this:

- The shell process fetches a new line (command) from `"commands.txt"`.
- If the command contains an output redirectioning part (like `"> foo.txt"`), shell process forks a new process that manipulates standard stream file handlers and at the end calls `execvp` with the correct command name and arguments. Depending on whether this job is a background job or not, the shell process waits for this command process to terminate.
- Otherwise, if the command does not have **output** redirectioning part, then your program performs the following:
  - The shell process creates a *pipe* (channel) *for this command* that will enable the communication between the new command process and the shell process before forking the new process. You can find an explanation for pipes and example programs under Recitation 3 material. More detailed information on how to create and use pipes can be found here: Creating A Pipe.
  - The shell process creates a new thread *for this command*. This new command listener thread of the shell process first tries to acquire the mutex that is shared among threads of the shell process. This ensures that the lines printed by this thread after that point will not be interrupted by other shell threads. Afterwards, it first prints a starting line of the form `"---- tid"` where `tid` is the thread identifier of the new thread. Then, it starts reading strings from the read end of the unique pipe between the command process and the shell process and prints them to the console. We strongly recommend you to use file streams (see File Streams) for reading data from the pipe. After the stream finishes and the command process stops sending data, the listener thread again prints `"---- tid"` as the last line and terminates. See sample console outputs at the end of this document to understand the behavior of the shell listener threads better.
  - The child process that is newly forked for this command also needs to do something extra. It has to redirect the `STANDARD_OUTPUT` of the child process to the write end of the pipe before calling the `execvp` command so that after executing `execvp` all the print statements by this command is sent to the pipe instead of the console. You can achieve this by using `dup` or `dup2` system calls.

You can find an explanation for these commands and example programs under Recitation 3 material. More detailed information on how to use these system calls: dup and dup2 System Calls.

- The way shell process handles the `wait` command must be modified as well. When this command is executed by the shell process, it does not only wait for all the background command processes to terminate, but also it has to wait until all the corresponding listener threads to print their content and terminate.

Note: It is possible to have issues when reading from the pipes, as your program may try to read it before writing it. If you encounter such an issue, one possible way to solve it is to use blocking option when using *read* function. If you want to learn more, you can use this link .If you decide to use C++, another way to solve it is to use file streams. You can use functions like *fgets* and *fdopen* to read from pipe until *EOF*.

2 Implementation Details

- Please compile your programs with `gcc`'s/`g++`'s `"-lpthread"` option.
- Use `fysnc` or `fflush` system calls inside shell listener threads after printing everything to ensure that your write actions are realized before releasing the mutex.
- When you are printing thread IDs to the terminal, you might see that some of the threads have the same ID. This is not unusual. When a thread ends, its ID might be reused for a new thread.
- You are not allowed to call pthread_join right after you create your thread, since if you implement it this way it will not actually be a multi-threaded program as there will only be one thread active at a time (Because pthread_join blocks the program and makes it wait until that thread is finished).
- Please do not forget to declare the mutex as a shared variable and initialize it.

3 Submission Guidelines

You are expected to submit a zip file named *<YourSUUserName>_PA2.zip* until Apr 26, 2023, Monday, 23:55.

The content of the zip file is as follows:

- `report.pdf`: Your report that explains the design decisions you have made for your implementation. You should clearly explain how you created pipes and established channels among command processes and the shell process, whether you used a single pipe for all commands or a pipe per command, how and where the threads are created and how they operate. Moreover, please describe the mutex you used for synchronization and how it works in your report.
- `cli.cpp/cli.c`: Your entire implementation must be in this file. We will not consider or compile any other `".h"` or `".c"\".cpp"` files even if you provide them in your submission. We will also not take any other `".c"\".cpp"` file with different name into consideration.

If your submission does not fit to the format specified above, your grade may be penalized up to 20 points. So pay attention to use zip option when compressing your file (do not use other options like rar or tar), and make sure your report is in pdf, not a txt or word file. Also you need to make sure the name of your submission fits what we asked, if you leave it something like *Cs307_PA2*, your grade will be penalized.

4 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 source code can be compiled and built without any problems.

2. **Report (20 pts):** Your report clearly explains the design decisions you have made and they are compatible with your implementation.

3. **Parsing (20 pts):** Your program correctly reads and parses the commands given in the file `"commands.txt"` and prints them in the format given in the format described in Section 1.1.3 to the file `parse.txt`.

4. **Multi-threading (20 pts):** For each background job, your listener threads print their first and last lines (lines of the form `"-----tid"` properly, i.e., two consecutive dashed lines must have the same `tid` value for the odd numbered lines assuming that line numbers start from 1. If you cannot implement the piping structure, you can just close the `STANDARD OUTPUT` for all the command processes. Then, if you implement the synchronization good enough, you can get full points from this part. Of course, if you trust your implementation and if you want to get points from the next grading items, you should not do that.

5. **Synchronization (20 pts):** Your program can process given `"commands.txt"` files without the concurrency error. All the lines by the same command are printed without interruption, although the order of commands might change.

6. **Termination (10 pts):** The shell process, command processes and the listener threads of the shell process terminate properly. When run from the CLI, your program does not freeze or deadlock.

For all grading items $i \in \{3, 4, 5\}$, i is a precondition for the grading item $i + 1$. Grading item 1 is a precondition for every other grading item. If a grading item x is a precondition for another grading item y, it means that you will get 0 from y unless you get full points from x. For example, if your code fails at item 4, you will automatically get 0 points from items 5 and 6. If you do not aim for the full points, we advise you to focus on completing initial items.

5 Sample Run

We provided one sample run for you. Please pay attention to the output format. Because there are some commands which will run as a background job, you might not get the same output ordering from the *commands.txt*

```
grep danger < input1.txt &
ls -a &
wc hw2.c > output1.txt
grep clearly -i < input1.txt &
```

commands.txt

```
You clearly don't know who you're talking to.
I am not in danger, Skyler.
I am the danger.
A guy opens his door and gets shot,
and you think that of me?
I am the one who knocks!
```

input1.txt

```
---- 139932422342400
.
..
a.out
commands.txt
hw2.c
input1.txt
---- 139932422342400
---- 139932430735104
I am not in danger, Skyler.
I am the danger.
---- 139932430735104
---- 139932413949696
You clearly don't know who you're talking to.
---- 139932413949696
```

Terminal

```
 209  565 4203 hw2.c
```

output1.txt

```
----------
Command: grep
Inputs: danger
Options:
Redirection: <
Background Job: y
----------
----------
Command: ls
Inputs:
Options: -a
Redirection: -
Background Job: y
----------
----------
Command: wc
Inputs: hw2.c
Options:
Redirection: >
Background Job: n
----------
----------
Command: grep
Inputs: clearly
Options: -i
Redirection: <
Background Job: y
----------
```

parse.txt