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

CS307 - Operating Systems

11 April 2022
DEADLINE: 25 April 2022, 23:55

1 Introduction & Problem Description

In this Programming Assignment, you are expected to fix a concurrency problem that might occur in your previous Programming Assignment (PA1 - CLI Simulator in C) when multiple commands try to print lines to the console concurrently. To achieve this, we will provide you a template program that has the functionality required by PA1 and you will improve it with your solution to the soon to be explained concurrency problem.

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` com-

mand 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 modify the shell simulator so 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, 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 redirectionings. **You do not have to synchronize the outputs that will be printed to a file instead of the console**.

2 The Solution

As you can see, the problem we consider is a concurrency problem. Multiple programs try to print lines to the console at the same time and due to context switches and as a result of this their outputs interleave. It is actually very similar to the problem you have seen in the `"prog1.c"` of *Recitation 4*.

In the recitation, 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"`), it behaves as described in PA1: 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 in addition to the PA1 rules:
  - 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.

Inside the PA2 package, we already provide you a template C file that you must extend to implement your solution. The template contains the parser and basic steps of PA1. We provide comments inside the template file so that you can easily understand the functionality of what we provide. **You are not allowed to delete or modify existing lines in this document.** You can only add new lines depending on your needs.

3 Implementation Details

- Please compile your programs with `gcc`'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.
- Please do not forget to declare the mutex as a shared variable and initialize it.

4 Bonus

If you are really enthusiastic about multithreading, you can extend PA2 with extra functionality and earn bonus points. The bonus task is to print outputs of commands according to their order in the `"commands.txt"`. Note that this can be easily done by getting rid of threads in the shell process. However, we want you to preserve the thread structure described above for efficiency. Instead, we want you to implement *your own mutex* so that threads can get the mutex according to the order they are created by the main thread of the shell process. You can do it by modifying one of mutexes we have seen in the lectures. In addition to `lock` and `unlock`, you can add new methods to the mutex API. If you want to use complex CPU instructions like `compare-and-swap`, `test-and-set` etc., you can use the `atomic` library of C. For more details on this library, see Atomic Library in C11

5 Submission Guidelines

You are expected to submit a zip file named $<YourSUUserName>\_PA2.zip$ until 25 April 2022, 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. If you have done the Bonus part, explain your own mutex implementation.
- `cli.c`: Your entire implementation must be in this file. **You cannot remove any lines from the template version of this file provided to you.** We will not consider or compile any other `".h"` or `".c"` files even if you provide them in your submission. We will also not take any other `".c"` file with different name into consideration.
- (Optional for Bonus) `cliM.c`: Modified version of the `"cli.c"` for the Bonus part. You must implement the mutex methods like `lock` and `unlock` in this file.

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 (120 for the submissions with the Bonus part):

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. **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, you trust your implementation and if you want to get points from the next grading items, you should not do that.

4. **Synchronization (30 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.

5. **Termination (20 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.

6. **Bonus (20 pts):** Console outputs of commands occur in the order determined by the `"commands.txt"` file and your program preserves the multi-threaded structure.

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.

7 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 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

```
1  209  565 4203 hw2.c
```

output1.txt