PA2 Report

Synchronizing the CLI Simulator

CS307 - Operating Systems

Mert Ekici 26772
ekicimert@sabanciuniv.edu

I. Command Line Interface

Command Line Interface allows users to interact with the Operating System by entering commands in a loop. In the previous assignment, designed CLI was open the concurrency problems for the cases where multiple commands try to print lines to the console concurrently. To solve this problem, given program - ***cli.c*** (solution for the PA1) will be improved. New CLI can overcome the concurrency problem with the added functionalities using ***pipe*** and ***multithreading***. Before moving to the next section, let's describe the functionality of the provided program ***cli.c***. This program can read and execute the commands from ***commands.txt***. It allows input – output redirection, supports background jobs and wait command.

II. Problem Description

While the shell is processing ***commands.txt***, it creates children command processes to handle background jobs ('***&***' sign). Since these commands run in the background, they can be executed concurrently and their lines could intervene with each other. Resulting output would be undesirable because it is difficult to identify and follow. This concurrency problem does not occur only among background jobs, it can appear in between foreground jobs as well. It must also be solved considering the commands without '***&***' sign. Furthermore, the same problem might happen for file streams but the case for ***output file stream*** is not in the scope of this assignment.

III. The Solution

As mentioned above, relevant problem is a ***concurrency*** problem. Multiple commands try to print lines to the console at the same time and context switch disrupts the output. The part of the code that should not be intervened is called ***critical section***. These critical sections should be wrapped with ***locks*** to mutually exclude other concurrent commands, programs, or ***threads***. Covering the critical part with ***mutual exclusion*** locks or ***mutexes*** makes that segment of execution ***atomic*** which stands for the executions that can be executed in one step without interruption. All of the ***printf*** statements should be wrapped with ***lock()*** and ***unlock()*** functions and just before the ***unlock()*** function call, ***fflush or fsync*** should be used to ensure that the lines printed by this thread cannot be interleaved by the others. Nevertheless, mutexes cannot be used directly since they are based on shared address spaces and heap of the same process to provide concurrency. However, mutexes can not provide synchronization among processes because they do not share states and each process has its unique address space. The shell simulator creates a new process for each line of the commands.txt (for each command). Thus, mutexes cannot be used immediately. To achieve this obstacle, this program will direct every console output of the commands to the parent shell so that only one process becomes responsible for the printing lines. Thereby, we can use mutexes for the synchronization. After this functionality, the shell has to continue processing new lines from the txt file by ***forking*** new processes and printing them to the console synchronously. It is a necessity to keep the background jobs meaningful so single-threaded solution is not allowed. Possible scenarios are given below in a sequence;

- The shell process gets a new line (command) from ***"commands.txt"***.
- If it the command has ***output redirectioning ('>' sign)***, it behaves as described in PA1. Shell forks a new process and this new process manipulates ***standard stream file handlers*** and at the end, it calls ***execvp().*** If the job is a ***background job*** then the shell process (parent) does not wait for the child process to terminate.
- Otherwise, if there is no ***output redirectioning ('>' sign)***, then the program performs the following in addition to the PA1;

    1. The shell process (parent) ***creates a pipe (channel) for this command*** for the communication between the new process and the parent process. This pipe will be used to listen and print the outputs of the child process' ***execvp*** call.
    2. The shell process (parent) ***creates a new thread for this command.*** First, this new listener thread tries to obtain the mutex that is shared among the threads of the shell process (parent). This ensures that the line printing process will not be interrupted by other shell threads. Afterward, it first prints "***---- tid"*** (***tid*** is the thread identifier of the new thread) as a starting line. Then, it starts ***listening and reading*** the strings from the ***read end of the unique pipe*** between the command process and the shell process and prints them to the console. After the stream finishes and the command process (child) stops sending more data, the listener thread prints "***---- tid"*** again as the last line before terminating.
    3. The child process that is newly forked has to redirect the ***STANDARD_OUTPUT*** of the child process to the write end of the pipe before calling the ***execvp***. This redirection puts all the print statements from ***execvp*** to the ***write end of the pipe*** and consequently, it can be read from the ***read end of the pipe*** by the shell process (parent). This functionality can be achieved by ***dup*** or ***dup2*** system calls. If ***dup*** is going to be used, then using ***close*** and closing the ***STDOUT_FILENO*** before calling the ***dup*** with the ***write end of the pipe*** would be enough.

- Finally, ***wait*** command should be handled as well. It must be modified in a way that, it waits not only for the background processes (not waited ones) but also waits for all the corresponding listener threads to print their content.

IV. Pipe

Described solution above requires a ***unique pipe*** for each command to make command process (child) and shell process (parent) communicate. Since the maximum number of threads is determined as 50 in ***threads[50]*** array, the implementation has ***fds[100] integer array*** that represents 2 sides of a unique pipe for each possible thread (command). In each iteration of ***while(fgets) loop***, a unique pipe is created by using ***pipe()*** *function* if the current command does not have output redirectioning. Based on the numbers stored in ***thrCount*** (integer that counts threads) and ***pointThread*** (index of the current thread in the ***threads*** array), ***pipe()*** function creates the pipe according to these integer values, e.g if the pointThread is 5, ***pipe(fds + 5 * 2)*** is called.

- pointThread = 5
- read end of the unique pipe = fds[5 * 2] = fds[10]
- write end of the unique pipe = fds[5 * 2 + 1] = fds[11]
- dup2(fds[11],STDOUT_FILENO) is called to redirect the output of execvp()

Then, in the child process ***write end of the relevant pipe*** is duplicated to ***STDOUT_FILENO*** with ***dup2 (at the end of each block relevant sides of the pipe is closed to prevent problematic "hang" cases).*** Concurrently, newly created ***thread (just before fork() call)*** of the shell process listens to ***read end of the pipe (unique pipe to the current command is passed to the display() function, so thread listens this unique pipe)*** after acquiring the ***mutex lock***. When nothing to print is left among the lines written by ***execvp***, the thread releases the ***lock*** and terminates.

V. Multi-threading

The shell process also creates a ***new thread*** for the cases where ***there is no output redirection*** of each iteration on the "commands.txt", just after it creates a ***unique pipe***. Following the current thread index of ***pointThread;***

- ***pthread_create(&(threads[pointThread]), NULL, display, &fds[pointThread])***

is called. Basically, this function call creates a new thread that executes display function by passing the unique pipe as an argument (parameter) to it. By doing this for each iteration (no output direction), the program obtains the desired concurrency that has concurrently running threads which executes commands from the txt file. In the ***display()*** function that is executed by the newly created thread, first thread tries to acquire the ***lock*** which is a global variable of ***pthread_mutex_t*** and it is initialized at the beginning of the program with ***PTHREAD_MUTEX_INITIALIZER***. If the ***mutex lock*** is available, it prints the "***---- tid",*** and starts to listen the ***read end of the unique pipe*** to print the output of the child process' execvp.

During the stream if there is something to read, ***read*** function reads the ***read end of the pipe*** to the ***character array*** called ***buf*** with a predefined size at the beginning of the program which is ***BUFSIZE.*** After that, ***buf*** is printed char by char. When there is nothing to print, ***display*** prints "***---- tid"*** and calls ***fflush()*** to ensure that the lines printed by this thread cannot be interleaved by the others. Finally, thread releases the ***lock.***

VI. Synchronization

Synchronization of the comands including both foreground and background jobs is established by threads and the shared global ***pthread_mutex_t*** variable ***lock.*** As explained in the Multi-threading section, active threads try to acquire the mutex called ***lock*** with ***pthread_mutex_lock*** function and when they are done, they release the mutex with ***pthread_mutex_unlock.*** All the lines by the same command are printed without interruption. Possible deadlocks (hold and wait cases) and freezes are avoided with the help of the correct usage ***pthread_mutex*** variable and its functions. Besides the synchronization provided by the ***lock*** variable, background and foreground jobs are handled in harmony with this synchronization. If the job is a foreground job (without output redirection), it is directly joined with ***pthread_join*** function and the child process is waited afterward with ***waitpid***. If the job is not a background job on the other hand, it is not joined and the child process is not waited specifically. They are handled at the end of the program or they might be handled by a ***wait command*** if it exists in the ***commands.txt.*** The functionality of the ***wait*** and mandatory waiting before terminating the ***main process*** is explained in the following section.

VII. Wait

In this programming assignment, ***wait*** command must be modified to wait (join) all the corresponding listener threads in addition to waited child processes. Since foreground jobs are waited and joined immediately after ***execvp*** is executed, ***wait*** function call should only wait for the threads with background jobs. For this functionality, the indexes (***pointThread***) of the background jobs in ***threads*** array are collected in ***bgs[50] integer array with bgscounter*** to follow the number of elements and using these indexes free background threads will be waited when the wait command appears in the txt file. After all of the corresponding background threads are joined using ***pthread_join*** and terminated, this code segment also waits for the free child processes with ***wait(NULL)*** which can be created for **the output redirectioning commands** or ***the background child processes*** that have not terminated yet***.*** Similarly to the first programming assignment, all of the running child processes and threads are waited at the end of ***main*** function after all lines are read (using the same method with the ***wait command*** functionality).

VIII. Bonus

Bonus extension of the PA2 asks for an extra functionality which ensures that outputs are printed in the same order with commands' appearance in the "commands.txt" preserving the thread structure described above for efficiency. In order to do this, a custom mutex must be implemented so that threads can obtain the mutex (lock) according to the order they are created by the main thread of the shell process. First, ***orderlock*** struct is defined with two attributes, ***int turn*** to store the current turn of the mutex and ***pthread_mutex_t lock*** to utilize mutex functionality. There are three ***void*** functions related to ***orderlock*** struct;

- ***lock_init(struct orderlock *l): orderlock initializer***
  - l->turn = 0
  - l-> lock = ***(pthread_mutex_t)PTHREAD_MUTEX_INITIALIZER***

- ***lock(struct orderlock *l, int myturn): lock the orderlock***
  - if myturn and l->turn is not equal wait with a ***while loop***
  - else ***pthread_mutex_lock***

- ***unlock(struct orderlock *l, int myturn): unlock the orderlock***
  - if myturn and l->turn is not equal wait with a ***while loop***
  - else increase the l->turn by one and ***pthread_mutex_unlock***

Implemented ***orderlock*** struct is used instead of ***pthread mutex***, and an additional ***turn*** variable allows us to follow the turn of the threads and give the lock accordingly. However, we have to pass the current turn (***pointThread***) to the ***display*** function with the ***unique pipe fd*** now. The simplest way of doing this is storing the ***turn*** information and the ***unique pipe fd*** within another ***struct*** which will be called ***turnHolder.*** The address of this struct will be passed to ***display*** function with **pthread_create,** *the* ***unique pipe*** *and* ***myturn (or the turn of the created thread)*** will be handled separately inside the ***display*** function. Newly defined ***lock, unlock*** functions are used instead of ***pthread lock and unlock*** functions in the same lines of the ***display*** function. Since this new implementation follows the current turn and distributes the lock based on this information, order property is ensured by the program. In ***cliM.c***, custom design mutex (order-specific) can be found. Other c file, ***cli.c***, has the main solution for the PA2.