CS314:Operating Systems Laboratory

Lab 1 Report Group 10

Utkarsh Prakash - 180030042 Shriram Ghadge - 180010015

---

1. Introduction

In this lab we were expected to implement a shell in C language. This should implement all basic shell commands like `ls`, `cd`, `pwd`, `cat` etc. Moreover, the shell should also be able to run commands in background as well as in foreground mode (both sequential and parallel execution). We also had to implement `exit` command and CTRL+C to kill all the foreground processes.

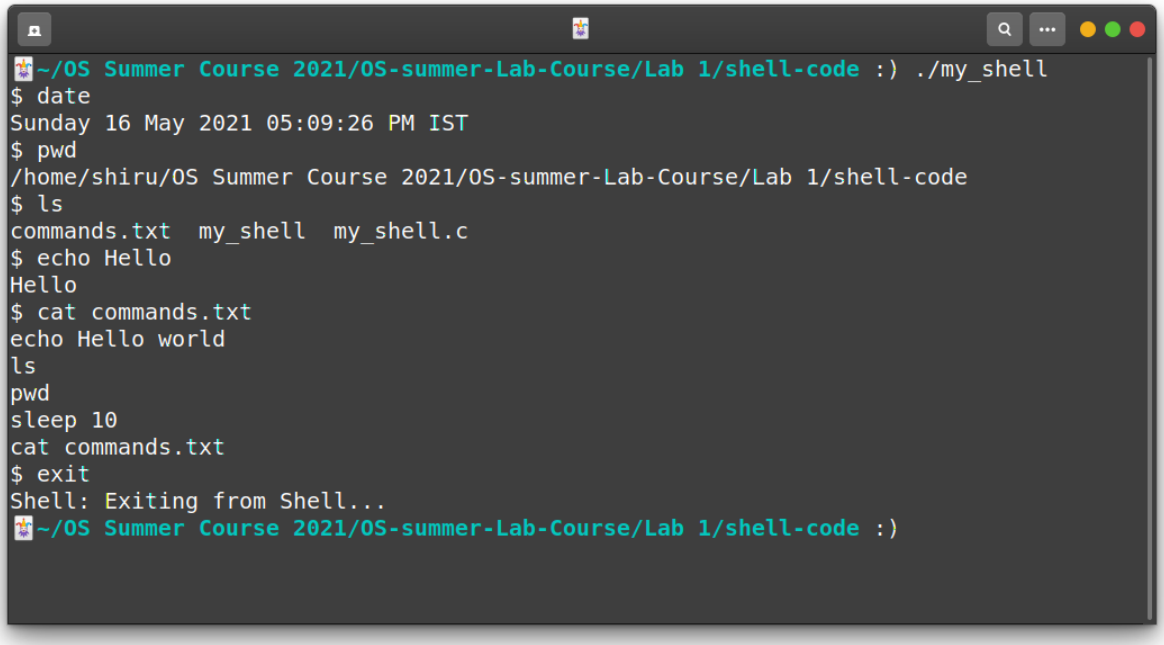


Fig 1. Basic Shell Commands

2. Part-A: Implementing basic Shell

In order to implement a basic shell we fork a new process from the parent. The child process runs the commands entered by the user. This is done using the `execvp()` system call which takes two arguments (one the command like `ls`, `cat`,`date` etc. and the other is the arguments of the command).

Since, `cd` command is to be executed by the parent we don't fork a child for that. It's implemented using `chdir()` system call.

3. Part-B: Foreground and Background Execution

a. Background Execution

To execute processes in the background, we fork a child process from the parent and run the command entered by the user using the `execvp()` system call. However, we don't use `wait()` system calls for the parent process to wait for the child. When the child process terminates, we use SIGCHLD signal to check whether the child terminated was a background process. If yes, then we print '`Shell: Background process finished`'.

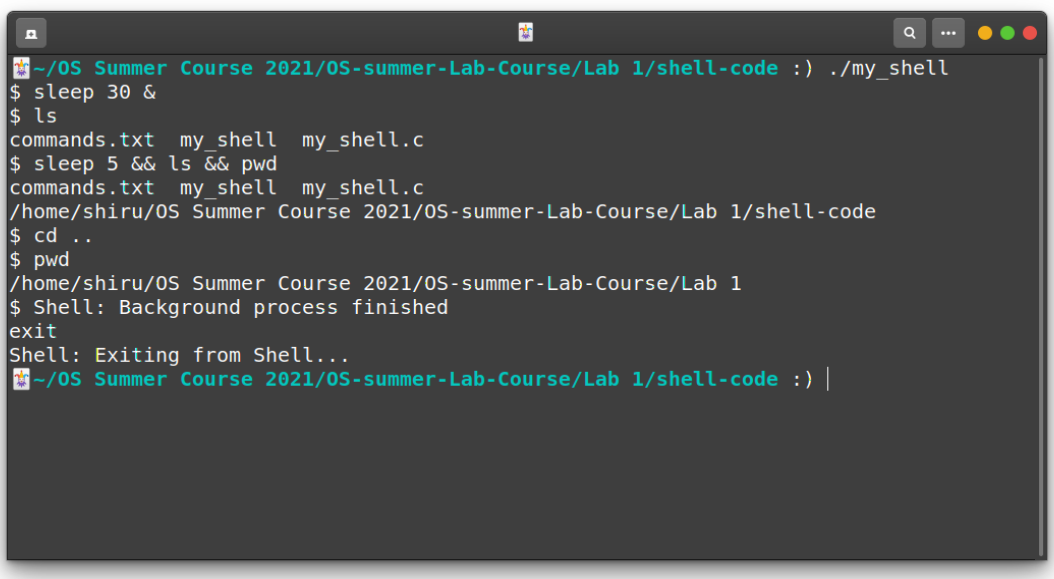


Fig 2. Background Execution

b. Serial execution

To execute the processes serially in foreground, we fork a new child process to run the command entered by the user. Meanwhile, the parent process waits for the completion of the child process using the `waitpid()` system call. This slightly different version of the `wait()` system call is used, so that the parent process can know that the signal of completion from the child was from a foreground process. Once the parent process receives a completion signal from the child process, it executes the next command in the series.

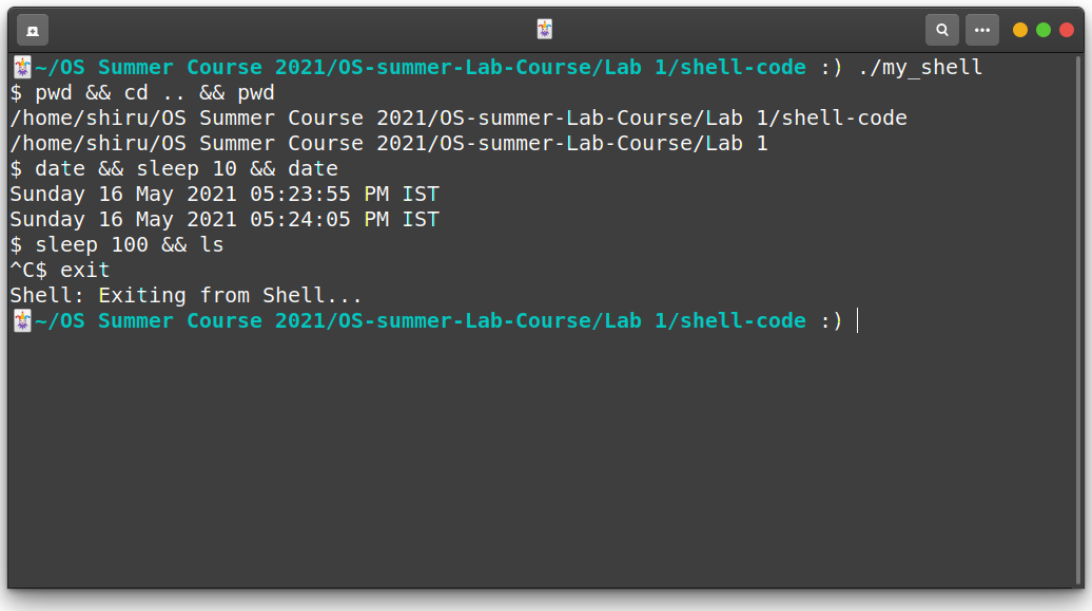


Fig 3. Serial execution

c. Parallel execution

In order to execute the command parallely in the foreground, we fork child processes to run each command simultaneously. Meanwhile, the parent process waits for the completion of all the child processes. In other words, until all the child processes are completed the parent process does not continue it's execution.

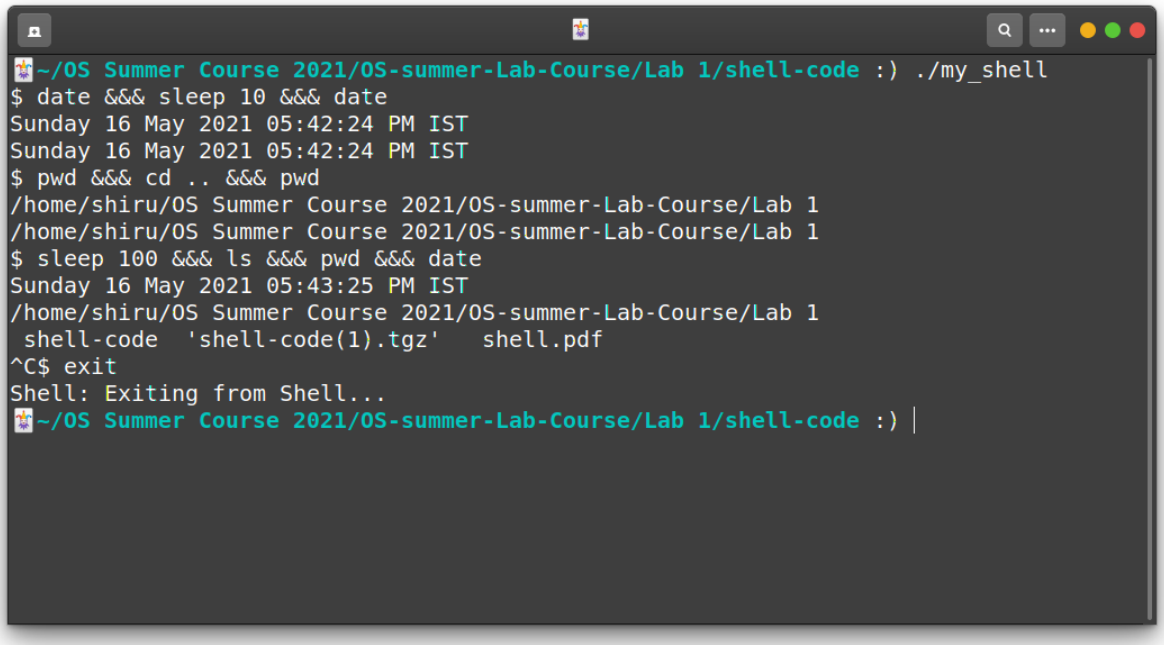
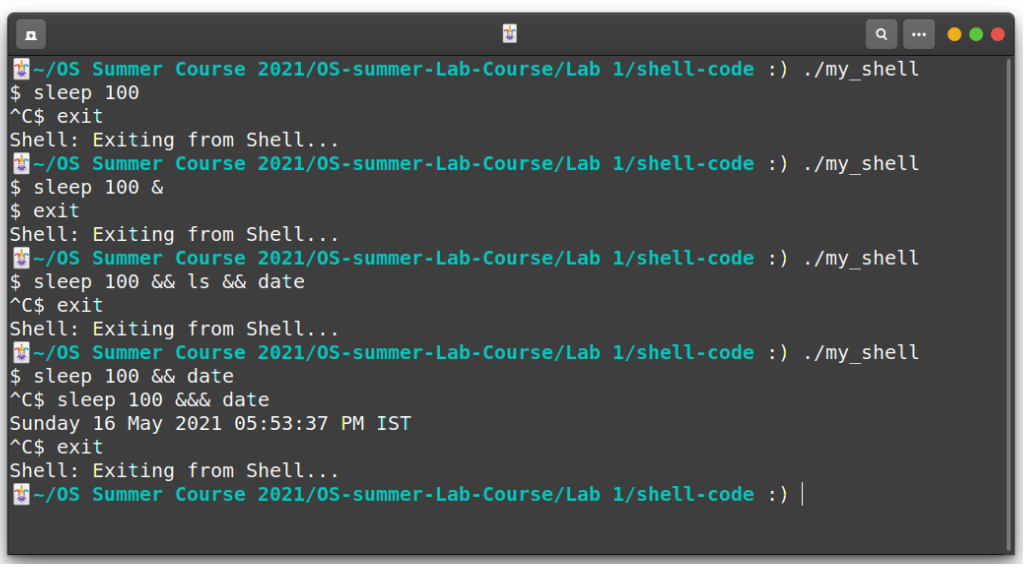


Fig 4. Parallel execution

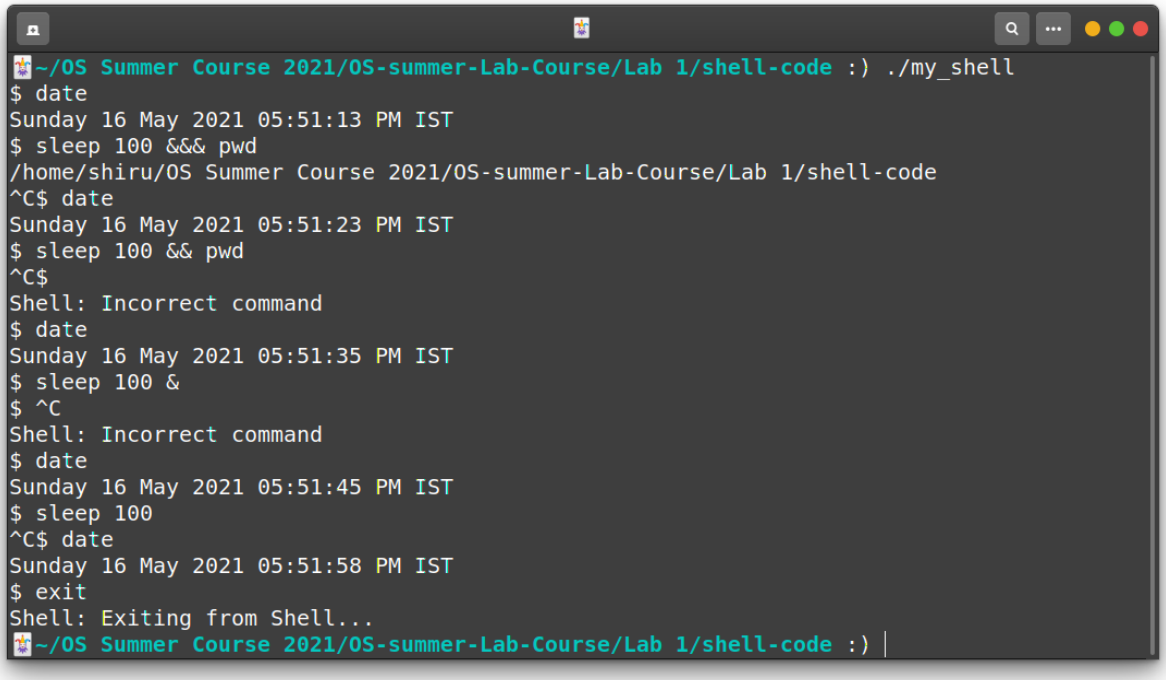
4. Part-C: exit command

In order to kill all the processes running in the background, we iterate over an array of all the background processes spawned by the shell and kill them using the `kill()` system call. Once, all the background processes are killed, then we `exit()` from the shell (i.e. ./a.out).



5. Part-D: CTRL+C

In order to implement the CTRL+C functionality i.e. when the user presses CTRL+C all the foreground running processes should be killed and the user should be prompted for a new command, we used `setpgid()` system call to group all the background processes. We have used this command so that SIGINT signal does not terminate any of the running background processes. Moreover, to stop any of the sequential commands from execution after pressing CTRL+C, we just maintain a boolean variable which does not spawn a process for any new command.



```
~/OS Summer Course 2021/OS-summer-Lab-Course/Lab 1/shell-code :) ./my_shell
$ date
Sunday 16 May 2021 05:51:13 PM IST
$ sleep 100 &&& pwd
/home/shiru/OS Summer Course 2021/OS-summer-Lab-Course/Lab 1/shell-code
^C$ date
Sunday 16 May 2021 05:51:23 PM IST
$ sleep 100 && pwd
^C$
Shell: Incorrect command
$ date
Sunday 16 May 2021 05:51:35 PM IST
$ sleep 100 &
$ ^C
Shell: Incorrect command
$ date
Sunday 16 May 2021 05:51:45 PM IST
$ sleep 100
^C$ date
Sunday 16 May 2021 05:51:58 PM IST
$ exit
Shell: Exiting from Shell...
~/OS Summer Course 2021/OS-summer-Lab-Course/Lab 1/shell-code :)
```