CS 333: Operating Systems Lab
Autumn 2022
Lab3: Shells and Signals

In this lab, you will learn about handling signals and building a simple interactive shell of your own to execute user commands, much like the bash shell in Linux.

Before You Begin:

- Get familiar with various process-related system calls in Linux: fork, exec, exit and wait. The *man pages* in Linux are a good source of learning. You can access the man pages from the Linux terminal by typing man fork, man 2 fork and so on. You can also find several helpful links online (e.g., manpage).

- Understand signals and signal handling in Linux.
  Signals are an OS-assisted mechanism for inter-process eventing/communication. Understand how processes can send signals to one another using the kill system call. Read up on how to write custom signal handlers to "catch" signals and override the default signal handling mechanism, using interfaces such as signal() or sigaction().

  Resources: man signal, man 7 signal,

  http://www.alexonlinux.com/signal-handling-in-linux

  http://www.cs.princeton.edu/courses/archive/spr06/cos217/lectures/23signals.pdf

- The operating system sends many signals, but we can also send signals manually.

  – **kill** sends a specified signal to a specified process (poorly named; previously, the default was to just terminate the target process). raise sends the specified signal to the current (calling) process itself.

Playing with signals:

In this part, we will work with writing custom signal handlers of different types. The most common way of sending signals to processes is using the keyboard. There are certain key presses that are interpreted by the system as requests to send signals to the process with which we are interacting:

- **Ctrl-C**

  Pressing this key causes the system to send an **INT** signal **(SIGINT)** to the running process. By default, this signal causes the process to terminate immediately.

- **Ctrl-**

  Pressing this key causes the system to send an **ABRT** signal **(SIGABRT)** to the running process. By default, this signal causes the process to terminate immediately. Note that this redundancy (i.e. Ctrl-\ doing the same as Ctrl-C) gives us some flexibility.

You can override the default signal handler to **IGN** (ignore) or write your own custom handler. This is useful if you would like your program to react differently to certain kinds of signals.

Signal: SIGCHLD → Description: When a child changes state, the kernel sends a SIGCHLD signal to its parent → Default Action: Ignore.
Note1: Signals aren't Queued, a signal handler is called if one or more signals are sent.
Note2: If you are generating the SIGINT with Ctrl+C on a Unix system, then the signal is being sent to the entire process group. ** can "see" in tour.c

1a. Write a program (sig_ignore.c) that prints the process ID and loops continuously, printing "Waiting..." after every three seconds and should ignore the SIGINT and SIGTERM signals.

- kill -INT 6421 sends a SIGINT to process 6421.
- A kill without a signal name is equivalent to SIGTERM.
- Terminate this program(forcefully) using kill -9. e.g, if the process ID is 6421, kill -9 6421 (SIGKILL is signal 9 that is sent by the kill command)

SAMPLE OUTPUT

```
~$ gcc sig_ignore.c -o sig_ignore.o
~$ ./sig_ignore.o
Process Id is: 9512
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Killed
~$
```

```
~$ kill -INT 9512
~$ kill 9512
~$ kill -9 9512
```

1b. Write a program (sig_generate.c) that takes the **PID of another process as command line arguments** and generate the following three signal sequentially.

i. SIGINT

ii. SIGTERM

iii. SIGKILL

(Hint: https://www.csl.mtu.edu/cs4411.ck/www/NOTES/signal/kill.html). Verify by running this program parallel to sig_ignore.c. It should be able to kill the sig_ignore program.

```
~$ gcc sig_generate.c -o sig_generate.o
~$ ./sig_generate 9512
SIGINT signal sent to PID: 9512
SIGTERM signal sent to PID: 9512
SIGKILL signal sent to PID: 9512
```

2. We have provided a template file (tour.c), modify/add your code in it or you can create your own (tour.c) from scratch.

   Five friends visit the Dinosaurs Park. Each of the five people wants to explore the park for a different amount of time (for simplicity, assume 5 * #people(i.e., 5, 10, 15, 20, 25) seconds). They may print a custom message - "Friend #<num> with process ID - <pid> has completed the tour." without quotes before exit. tour.c should spawn a child process for each person (1 to 5) and have a custom signal handling code that catches the Ctrl+C signal received from the terminal and exits only when all 5 friends have completed the tour. Also ensures that the child processes don't receive the Ctrl+C signal.

   (different pgid)

SAMPLE OUTPUT

```
# Compile tour.c using gcc
  gcc tour.c -o tour.o

# Run the file tour.o
  ./tour.o
# Sample Output of the program tour.c
  Welcome to the Dinosaurs Park.
  The process ID of Dinosaurs Park: 19690
  Friend #1 with process ID - 19691 has completed the tour.
  Friend #2 with process ID - 19692 has completed the tour.
  ^C

  You have interrupted the tour.
  Oh Sorry! Only 2 out of the 5 friends have completed the tour.
  Friend #3 with process ID - 19693 has completed the tour.
  Friend #4 with process ID - 19694 has completed the tour.
  Friend #5 with process ID - 19695 has completed the tour.
  ^C

  You have interrupted the tour.
  All 5 friends have completed the tours.
  Thank you for visiting the Dinosaurs park
```

Note:
^C means Ctrl + C signal is sent from the keyboard.
PIDs may/will not be the same as above.
You can send Ctrl + C at any time before completion of all five tours.

Additional Reading

Signal Mask: Sometimes we need to block some signals so that critical sections are not interrupted. Every process maintains a signal mask telling which signals are blocked. Learn more about this by visiting the following links: - signal mask, sigfillset, sigprocmask

Simple shell

This part of the lab focuses on building a simple shell using **C**, to run simple Linux commands. The idea is for the main shell program to act as a parent process that accepts and parses commands and then instantiates child processes to execute the desired commands. You will need to use **fork** and **exec** system calls to implement the required functionalities.

We have provided **shell.c**, a file which has skeleton code containing a function **tokenizer** to tokenize the input command. You have to add your code to this file and submit this file in moodle with the same naming convention.

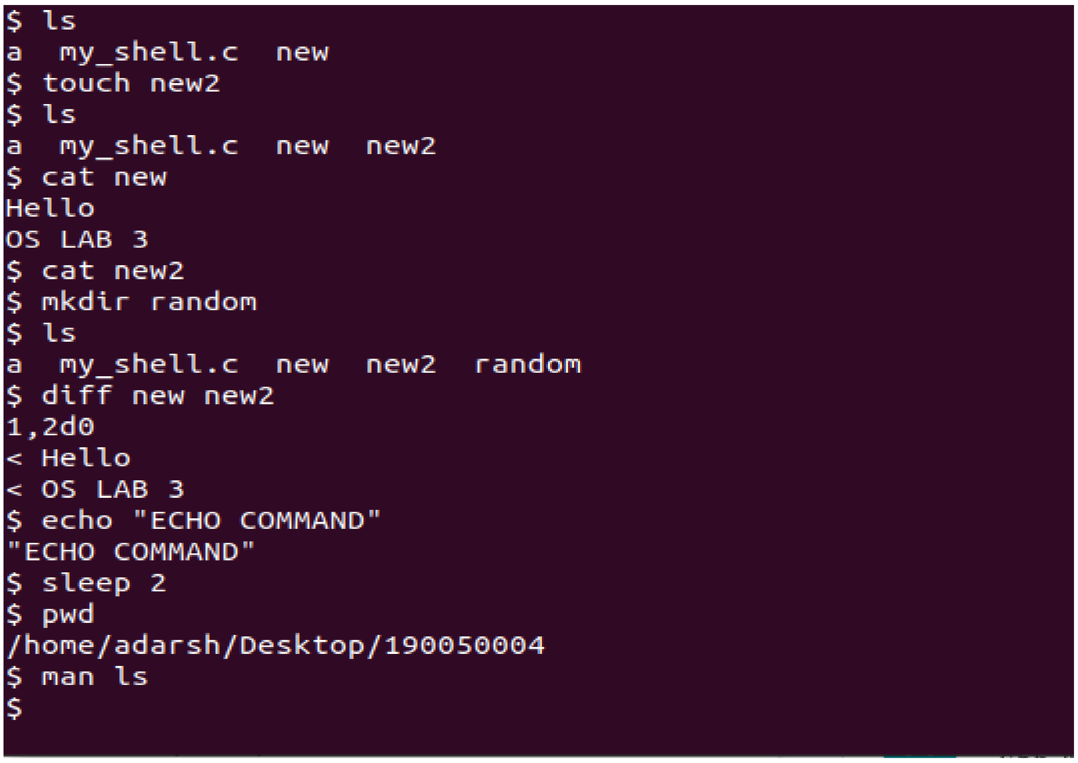
You must not use the `system` function call provided by the **C**-library. Also, you must execute Linux system programs wherever possible, instead of re-implementing functionality. For example, to implement **echo**, you should not implement **echo** binary, instead use the binary already implemented in Linux using an **exec** system call.

The following functionalities should be supported:

- Simple standalone built-in commands of Linux should be executed, (the list of commands which will be checked is given as a list below) as they would be in a regular shell. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during executing these commands must be displayed in the shell. Note that simple versions of the following commands are supposed to work properly, we are not expecting flagged versions and piped commands.

  Commands: ls, cat, echo, sleep, pwd, ps, man, touch, mkdir, rm, rmdir, kill, diff, clear

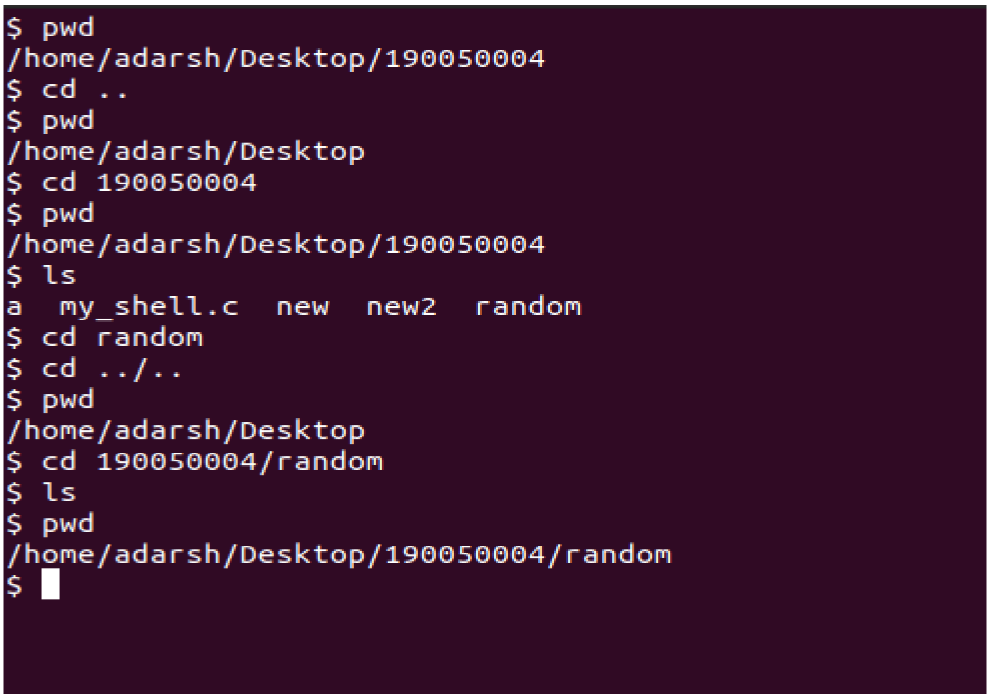
  Expected Shell Behavior:



```
$ ls
a  my_shell.c  new
$ touch new2
$ ls
a  my_shell.c  new  new2
$ cat new
Hello
OS LAB 3
$ cat new2
$ mkdir random
$ ls
a  my_shell.c  new  new2  random
$ diff new new2
1,2d0
< Hello
< OS LAB 3
$ echo "ECHO COMMAND"
"ECHO COMMAND"
$ sleep 2
$ pwd
/home/adarsh/Desktop/190050004
$ man ls
$
```

- **cd directory-name -** The command must cause the shell process to change its working directory (without using the **cd** system binary). This command should take one and **only one argument**; an incorrect number of arguments (e.g., commands such as cd, cd dir1 dir2 etc.) should print an error in the shell. the **cd** should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). For this, and all commands below, the incorrect number of arguments or incorrect command format should print an error in the shell. After the shell prints such errors, the shell should not crash. It must simply move on and prompt the user for the next command.

  Expected cd command behaviour:



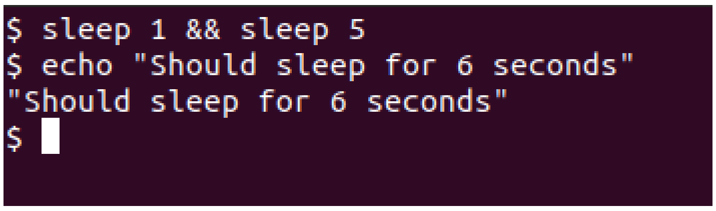
```
$ pwd
/home/adarsh/Desktop/190050004
$ cd ..
$ pwd
/home/adarsh/Desktop
$ cd 190050004
$ pwd
/home/adarsh/Desktop/190050004
$ ls
a  my_shell.c  new  new2  random
$ cd random
$ cd ../..
$ pwd
/home/adarsh/Desktop
$ cd 190050004/random
$ ls
$ pwd
/home/adarsh/Desktop/190050004/random
$ 
```

* to make process run in background (like "sleep 10 &" on terminal), just don't make parent wait for child -- so child running in background √

- Multiple users commands separated by **&&** should be executed one after the other serially in the foreground. The shell must move on to the next command in the sequence only after the previous one has been completed (successfully, or with errors) and the corresponding terminated child is reaped by the parent. The shell should return to the command prompt after all the commands in the sequence have finished execution.

  Expected Behavior:

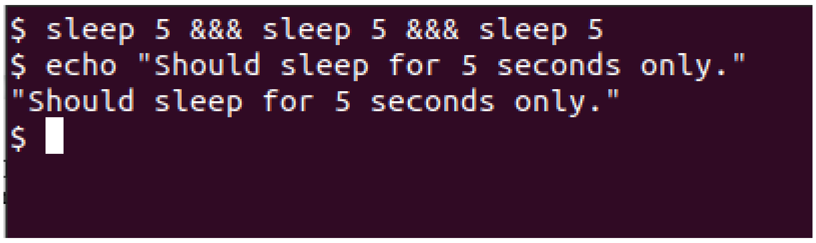
  1. Multiple sleep commands can be used and each command should be executed one by one



```
$ sleep 1 && sleep 5
$ echo "Should sleep for 6 seconds"
"Should sleep for 6 seconds"
$ 
```

- Multiple commands separated by **&&&** should be executed in parallel in the foreground. That is, the shell should start execution of all commands simultaneously, and return to the command prompt after all commands have finished execution and all terminated children reaped correctly.

1. Multiple sleep commands can be used and each command should be executed parallelly.



```
$ sleep 5 &&& sleep 5 &&& sleep 5
$ echo "Should sleep for 5 seconds only."
"Should sleep for 5 seconds only."
$ 
```

2. You may assume that the commands entered for serial or parallel execution are simple Linux commands, and the user enters only one type of command (serial or parallel) at a time on the command prompt.

#define MAX_NUM_TOKENS 64

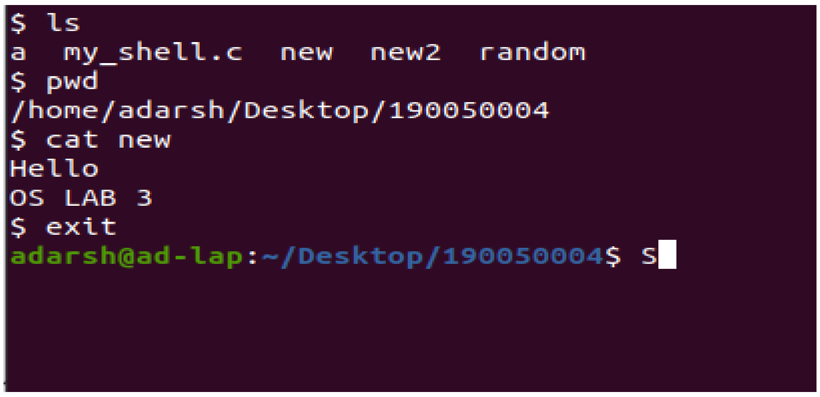
3. You may assume that there are no more than 64 foreground commands given at a time. Use multiple long-running commands like **sleep** to test your series and parallel implementations, as such commands will give you enough time to run **ps** in another window to check that the commands are executing as specified.

- Implement a mechanism to exit the shell, using the “exit” command. On giving this command, the shell should free its memory, **and kill all the processes spawned by the shell** (to do this you can keep track of all the processes running using their pids and then kill on exit) and the shell programs should terminate without any error. Note that the exit command will not be one of the commands for serial or parallel execution parts.

but pids can be reused... so shouldn't wait then (if kill being used...) [?]

in code, wait used ✓

Expected Behavior:



```
$ ls
a  my_shell.c  new  new2  random
$ pwd
/home/adarsh/Desktop/190050004
$ cat new
Hello
OS LAB 3
$ exit
adarsh@ad-lap:~/Desktop/190050004$ S
```

Some Guidelines:

- When a process completes its execution, all of the memory and resources associated with it are de-allocated so they can be used by other processes. This cleanup has to be done by the parent of the process and is called reaping the child process. The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input.

- By carefully reaping all children, the shell must ensure that it does not leave behind any zombies or orphans when it exits.

- You may assume that the input command has no more than 1024 characters, and no more than 64 “tokens”. Further, you may assume that each token is no longer than 64 characters. #define's on top...

- You will find the **chdir** system call useful to implement the **cd** command

Submission Instructions

• All submissions via moodle. Name your submission as <rollnumber_lab3>.tar.gz

(e.g 190050096_lab3.tar.gz)

• The tar should contain the following files in the following directory structure:

```
<rollnumber_lab3>/
        |__shell.c
        |__sig_generate.c
        |__sig_ignore.c
        |__tour.c
```

• Your code (shell.c, sig_generate.c, sig_ignore.c and tour.c) should be well commented and readable.

• tar -czvf <rollnumber_lab3>.tar.gz <rollnumber_lab3>

Deadline: Thursday 18th August 2022 11:58 PM via moodle.