Operating Systems Laboratory (Mini Linux Shell)

- You learned about syscalls in your lectures. Syscalls are function calls that can be used by your program (i.e., user space program) to let OS perform privileged tasks for you (e.g., reading directly from the keyboard or sending data to your printer). Specifically, the following syscalls in Linux might be useful for you in this assignment and in general.
    - dup
    - excelp
    - fork
    - signal
    - pipe
    - select
    - poll
    - open, read, write
    - chmod
- In this assignment, we will build a shell with some basic and some advanced functionalities.

Implement a shell that will run as an application program on top of the Linux kernel. The shell will accept user commands (one line at a time), and execute the same. The following features must be implemented:

a) **Run an external command**

The external commands refer to executables that are stored as files. They have to be executed by spawning a child process and invoking **execlp()** or some similar system calls. Example user commands:

./a.out myprog.c cc –o
myprog myprog.c

ls -l

b) **Run an external command by redirecting standard input from a file**

The symbol "**<**" is used for input redirection, where the input will be read from the specified file and not from the keyboard. You need to use a system call like **dup()** or **dup2()** to carry out the redirection. Example user command:

./a.out < infile.txt
sort < somefile.txt

c) **Run an external command by redirecting standard output to a file** The symbol "**>**" is used for output redirection, where the output will be written to the specified file and not to the screen. You need to use a system call like **dup()** or **dup2()** to carry out the redirection. Example user commands:

./a.out > outfile.txt
ls > abc

d) **Combination of input and output redirection**

Here we use both "<" and ">" to specify both types of redirection. Example user command:

./a.out < infile.txt > outfile.txt

e) **Run an external command in the background with possible input and output redirections**

We use the symbol "**&**" to specify running a command in the background. The shell prompt will appear and the next command can be typed while the said command is being executed in the background. Example user commands:

./a.out &

./myprog < in.txt > out.txt &

f) **Run several external commands in the pipe mode**

The symbol "**|**" is used to indicate pipe mode of execution. Here, the standard output of one command will be redirected to the standard input of the next command, in sequence. You need to use the **pipe()** system call to implement this feature. Example user commands:

ls | more cat abc.c |
sort | more

g) **Interrupting commands running in your shell (using signal call)**

- Implement a feature to halt a command running in your shell during runtime. For instance, if the user presses "Ctrl - c" while a program is executing, the program should stop executing, and the shell prompt should reappear. Note that the shell should not stop if the user presses "Ctrl - c".

- Implement a feature to move a command in execution to the background. If the user presses "Ctrl - z" while a program is executing, the program execution should move to the background and the shell prompt should reappear.

h) **Implementing a searchable shell history**

- Maintain a history of the last 10000 commands run in your shell (hint: check how bash saves a history in a file)
- Implement a command "history" which will show the most recent 1000 commands.
- For searching through the shell history implement the following: If the user presses "ctrl+r", your shell shows a "Enter search term" prompt. The prompt will take a string as an input from the user. On pressing Enter, the prompt will print the most recent command from the history of 10,000 commands which exactly matches the user input.

- In case there is no such command, print the command(s) for which the length of the longest substring match is largest with the user-given string and the length of match is > 2 characters.
- Otherwise print “No match for search term in history”.

Note : There will be some marks reserved for the efficiency of the algorithm used.

Note: Check this link to know how searching through bash history works.

i) **Implementing auto-complete feature for file names**

- Implement an auto-complete feature for the shell for the file names in the working directory of the shell.
- In your shell if you write the first few letters of a file (in the same directory where the shell is running) and press Tab key then:
    - If one file with those starting characters exists, then your shell will display the name of this file in the terminal with the rest of the typed command intact.
    - If multiple files with those starting characters exist, then your shell should print the longest substring match (starting from the beginning) for all those files. If even then multiple files exist then your shell should ask the users which file to choose using a prompt.
    - if no file with those starting characters exist, then your shell should print nothing Example:
        - Imagine the directory has files “abc.txt def.txt abcd.txt”
        - Then in your shell if the user input “./myprog de” and press Tab then your shell should print “./myprog def.txt” and wait for user input
        - in your shell if the user input “./myprog def.txt abc” and press Tab then your shell should print “1. abc.txt 2. abcd.txt” and wait for user input. If the user press (1) then the shell should print “myprog def.txt abc.txt”
        - Note that even if myprog is not there, the auto complete should work.

j) **Implementing a new command “multiWatch”**

First Take a look at the **watch** command in Linux from this link. Now we want to Implement **multiWatch** with the following functionality .

Command : multiWatch ["cmd1", "cmd2", "cmd3",...]
Function of the command : This command will start executing cmd1, cmd2, cmd3... **parallelly with multiple processes.** Then it will keep printing whatever is output by cmd1, cmd2, cmd2 etc (with unix timestamp and command name which generated the output)**.** Of course you may get different outputs each time.

Note :

- This is not the same as **watch "cmd1 && cmd2 && cmd3"** . This command will sequentially execute the first **cmd1** then **cmd2** then **cmd3** (provided no error occurred). But the assignment asks for something different.

Sample Output :

"cmd1" , current_time :

<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<Output1

->

"cmd2" ,current_time:

<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<Output2

->

"cmd1" , current_time :

<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<Output1

->

"cmd1" , current_time :

<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<Output1

->

Workflow hints for implementing multiWatch :

Main Program (shell):

- Fork processes for each command.
- Create (hidden) temporary files ".temp.PID1.txt" , ".temp.PID2.txt" for each of the commands which are run. PID is the real process id for the corresponding child process. The command should write their output to this file, and your shell should read from the file.
- Use the file descriptors for the processed into the **select/Poll** system calls
- Keep writing to stdout as **select/Poll** returns in the given output format.

In each forked process :

- Execute command (you will need to fork the process, you already did it in earlier part of this assignment)

Signal :

- This execution should end after receiving **crtl+C** from the user.
- Then, all the forked processes also must be returned / killed. ● **Delete** all the temp files.

Note :

- Marks will be deducted if output is not found in the specified format.
- Try to be as efficient as possible.

Implementation Help:

For redirecting the standard input or output, you can refer to the book: "*Design of the Unix Operating System*" by Maurice Bach. Actually, the kernel maintains a file descriptor table or FDT (one per process), where the first three entries (index 0, 1 and 2) correspond to standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**). When files are opened, new entries are created in the table. When a file is closed, the corresponding entry is logically deleted. There is a system call **dup(xyz)**, which takes a file descriptor **xyz** as its input and copies it to the first empty location in FDT. So if we write the two statements: **close(stdin); dup(xyz);** the file descriptor **xyz** will be copied into the FDT entry corresponding to **stdin**. If the program wants to read something from the keyboard, it will actually get read from the file corresponding to **xyz**. Similarly, to redirect the standard output, we have to use the two statements: **close(stdout); dup(xyz);**

Normally, when the parent forks a child that executes a command using **execlp()** or **execvp()**, the parent calls the function **wait()**, thereby waiting for the child to terminate. Only after that, it will ask the user for the next command. However, if we want to run a program in the background, we do not give the **wait()**, and so the parent asks for the next command even while the child is in execution.

A pipe between two processes can be created using the **pipe()** system call, followed by input and output redirection. Consider the command: **ls | more**. The parent process finds out there is a pipe between two programs, creates a pipe, and forks two child processes (say, X and Y). X will redirect its standard output to the output end of the pipe (using **dup()**), and then call **execlp()** or **execvp()** to execute **ls**. Similarly, Y will redirect its standard input to the input end of the pipe (again using **dup()**), and then call **execlp()** or **execvp()** to execute **more**. If there is a pipe command involving N commands in general, then you need to create N-1 pipes, create N child processes, and connect each pair of consecutive child processes by a pipe.

Submission Guideline:

- Create a single program for the assignment, and name it Assignment2_<groupno>_<roll no. 1>_< roll no. 2>.c or .cpp (replace <groupno> and <roll no.> by your group number and roll numbers), and upload it.
- You must show the running version of the program(s) to your assigned TA during the lab hours.

Evaluation Guidelines:

Total marks for this assignment is 100.

Items	Marks
Process creation and running an external command	8

Redirection of stdin	8
Redirection of stdout	3
Redirection of stdin and stdout together	5
Running an external command in background with I/O redirection	8
Running several external commands in pipe mode	10
Interrupting programs running in your shell	10
Implementing a searchable shell history	13
Implementing auto-complete feature for file names	15
Implementing a new command “multiWatch”	20
Total	100