

Computing Laboratory I (CS69011)

Autumn Semester 2023

Assignment 7: OS programming

Assignment date: October 09, 2023

Important Instructions:

1. Using Linux

- a. This assignment is based on OS programming and will be beneficial for the students to do this assignment in a Linux OS.
- b. If you don't have linux, use virtualbox + linux image, both free. We suggest Ubuntu, but feel free to use any linux version. If you decide to use cygwin or wsl, it's up to you, just remember that most of your batch mates (and the instructors) will use Linux (and ubuntu), so, naturally, we might not be able to provide support for other systems.
- c. In case of any difficulties in understanding any functionalities, you may use the same functionality in a Linux Terminal to get a better understanding of how it works.

2. Programming language

- a. This assignment is using the C programming language. You can also use cpp (without STL) if you want, as the important calls remain mostly the same.

3. Error handling

- a. A proper error handling (e.g., when the syscall or the library calls fail) is expected in this Assignment.

4. Input/output

- a. The inputs should be taken from the Terminal and results displayed in the Terminal, **if explicitly not mentioned**.

5. Documentation

- a. Always consult the document and man pages to know more about the system calls and the correct function arguments / function behavior.
-

Part - A (System Calls - 1)

In this part, you will write a basic shell. A parent process will give a prompt **shell>** to the user and wait for a command (from a set of commands given below) to be typed. For each command, the parent process will **fork** a child process which will execute that command. Normally, the parent process will wait for the command to be finished, after which it will give the **shell>** prompt again and wait for the next command.

However, if the command is followed by an **&** (ampersand) character (with or without blanks after the command), the parent process will return immediately (that is, it will not wait for the child to finish the command), and wait for the next command.

Implement the following commands:-

1. **pwd** - shows the present working directory
2. **cd <directory_name>** - changes the present working directory to the directory specified in **<directory_name>**. A full path to a directory may be given.
3. **mkdir <directory_name>** - creates a new directory specified by **<dir>**.
4. **ls <flag>** - shows the contents of a directory based on some **optional** flags, like **ls -al**, **ls**, etc. See man page of **ls** to know more about the flags.
5. **exit** - exits the shell
6. **help** - prints this list of commands
7. **If you type any other command at the prompt, the shell should just assume it is an executable file and try to directly execute it.**

Note:- A proper error message should be shown at the terminal/prompt, if the above manual (given to user) is not followed.

Hint: see **fork** and **exec** family of system calls.

Part - B (System Calls - 2)

You need to extend the shell created in the previous part-A using I/O redirection to make it possible to:-

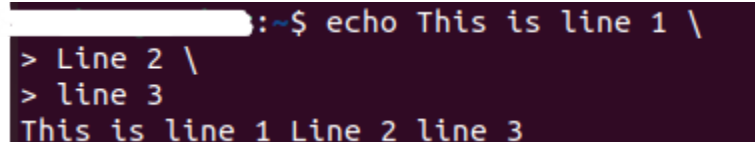
1. pipe one command through the other using the **|** operator. For example, it should be possible to give the following commands to your program:
ls -al | more or **cat text.txt | grep abcd**

Hint: See **dup** and **dup2** system call

Part - C (readline library)

Extend the shell created in the previous part-B to incorporate some shell functionalities

1. **Multi-Line commands**:- A single command may be extended to a multiline command using backslash(\) character. Example shown in Fig. 1



```
~$ echo This is line 1 \  
> Line 2 \  
> line 3  
This is line 1 Line 2 line 3
```

Fig. 1:- Example of Multi-Line Command

2. **Command history**:- Using the up arrow we can execute the previous commands.
3. **Command Editing**:- If a long command has some mistake, the user may press Ctrl+A to move the cursor to the start of the line and make corrections.

Part - D (ncurses library)

Extend the shell created in the previous part-C using ncurses by introducing your own 'vi' command which is a text editor. **vi <filename>** will open an editor terminal, that supports the following functionalities after taking input from your keyboard:-

1. **Toggle keys** - up, down, left, right arrow keys to toggle the cursor between lines
2. **Escape Key** - ESC to close the editor terminal
3. **Delete Key** - DELETE - to delete the current character pointed out by the cursor.
4. **Insert character keys** - [a-z], [A-Z], [0-9], characters at the location pointed out by the cursor.
5. **Ctrl + S** - save as a file
6. **Ctrl + X** - exit the editor terminal
7. After a successful exit operation, mention the number of lines, number of words and number of characters written/modified in the initial shell prompt.

Part - E (X11 programming)

Now that you created a shell with an inbuilt text editor, one thing is remaining. It still runs from your terminal. But a real shell should run as a standalone application. So, create a shell interface using X11 programming combining the terminal and shell by extending the previous parts. Use the shell code you have written till Part-D. Know more about Xlib from <https://tronche.com/gui/x/xlib/> and <https://tronche.com/gui/x/xlib-tutorial/>

Part - F (Threads)

You need to extend the shell created in the previous parts by using threading concept to calculate vector operations. You need to use pthreads library for creating threads and distributing jobs. Using the **pthread** library in your shell implement the following commands:-

1. **addvec <file_1> <file_2> -<no_thread>**:- Addition of two vectors, <file_1> and <file_2> both contain a line with **n** space separated numbers denoting a n-dimensional vector **v**, ($v \in \mathbb{R}^n$). **<no_thread>** denotes the number of threads to run. The default (if not anything mentioned is 3)
2. **subvec <file_1> <file_2> -<no_thread>**:- Subtraction of two vectors, $v_1 - v_2$, $v_1 \in \text{<file_1>}$ and $v_2 \in \text{<file_2>}$ both contain a line with **n** space separated numbers denoting a n-dimensional vector **v**, ($v \in \mathbb{R}^n$). **<no_thread>** denotes the number of threads to run. The default (if not anything mentioned is 3).
3. **dotprod <file_1> <file_2> -<no_thread>**:- vector dot product of two vectors, <file_1> and <file_2> both contain a line with **n** space separated numbers denoting a n-dimensional vector **v**, ($v \in \mathbb{R}^n$). **<no_thread>** denotes the number of threads to run. The default (if not anything mentioned is 3).

Hint: For this one, you need to learn and use pthreads.

- Here is simple tutorial with working codes:
<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>
- Here is a more detailed one: <https://hpc-tutorials.llnl.gov/posix/>
- And of course the man page:
<https://man7.org/linux/man-pages/man7/pthreads.7.html>