Computing Lab (CS69201)
Project: MyTerm - A Custom Terminal with X11GUI

Maximum Marks: 100 **Deadline: October 12, 2025 (11:59 PM)**

You learned about System Calls (syscalls) today. 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 project and in general.

- dup/dup2
- execlp/execvp
- fork
- signal
- pipe
- select
- poll
- open, read, write
- chmod

In this project, we will build our **custom shell** called **MyTerm** that will have its own GUI with some basic and some advanced functionalities.

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

1. **Graphical User Interface**

   Use the X11 library to render a custom terminal interface. The GUI should mimic the behavior of a traditional bash terminal: it must accept user input, execute commands, and display the output in the window. The GUI must support multiple tabs, with each tab running as an independent shell instance. All input and output should be handled through the X11 window rather than the standard console. Please check out these links to understand how to use X11.

   Workflow for making a terminal GUI with X11:

   - Use Xlib functions:
     - XOpenDisplay(), XCreateSimpleWindow(), XMapWindow() to create a window.
     - XNextEvent() to capture user input events.
     - XDrawString() or font/text rendering for output.
   - Maintain a text buffer in your program to store what should be drawn in the window.

- Handle keyboard input by mapping X11 KeyPress events to characters, then append to the buffer.
- For multiple tabs:
  - Each tab should correspond to a separate child shell process (fork() + execvp()).
- Use a tab-switching mechanism in the GUI to select which buffer/process is active.
- Redirect the shell process's stdin/stdout through pipe() so that the child's output can be captured and drawn in the X11 window.

2. **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:

```
user@myterm>  cd ~/Downloads/mydir
user@myterm>  ls -l
user@myterm> gcc -o myprog myprog.c
user@myterm> ./myprog
```

3. **Take multiline unicode input**

The terminal should be able to take in multiline input as unicode characters. For example:

```
user@myterm> echo "We say Hello in English \n\
                  हम हिंदी में नमस्ते कहते हैं \n\
                  Aloha mākou ma ka ʻōlelo Hawaiʻi \n\
                  मराठीत नमस्कार"
We say hello in English
हम हिंदी में नमस्ते कहते हैं
Aloha mākou ma ka ʻōlelo Hawaiʻi
मराठीत नमस्कार
```

Use **read()** and **write()** with **setlocale()** to display characters in unicode.

4. **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:

   user@myterm> ./a.out < infile.txt

   user@myterm> sort < somefile.txt

5. **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:

   user@myterm> ./a.out > outfile.txt

   user@myterm> ls > abc

   a. **Combination of input and output redirection**

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

      user@myterm> ./a.out < infile.txt > outfile.txt

6. **Implementing support for pipe**
   a. **Pipe Support:**

      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:

      user@myterm> ls *.txt | wc -l

      user@myterm> cat abc.c | sort | more

      user@myterm> ls *.txt | xargs rm

7. **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 project asks for something different.

Sample Output :

user@myterm>

```
"cmd1"  , current_time   :
----------------------------------------------------
Output1
----------------------------------------------------
"cmd2" ,current_time:
----------------------------------------------------
Output2
----------------------------------------------------
"cmd1"  , current_time   :
----------------------------------------------------
Output1
----------------------------------------------------
"cmd1"  , current_time   :
----------------------------------------------------
Output1
----------------------------------------------------
```

Workflow hints for implementing multiWatch:

Main Program (shell):

a. Fork processes for each command.
b. 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.
c. Use the file descriptors for the processed into the **select/Poll** system calls
d. Keep writing to stdout as **select/Poll** returns in the given output format.

In each forked process:

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

Signal:

a. This execution should end after receiving **Ctrl+C** from the user.
b. Then, all the forked processes also must be returned / killed.
c. **Delete** all the temp files.

Note:

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

8. **Line Navigation with Ctrl+A and Ctrl+E**

   Implement support for command-line editing shortcuts similar to Bash. When typing a command in your shell:

   a. Pressing **Ctrl+A** should move the cursor to the start of the current line.
   b. Pressing **Ctrl+E** should move the cursor to the end of the current line.
   c. This makes your shell's input feel more like a real terminal.

   Guidelines for implementation:

   a. Use terminal raw mode (termios) to capture keypresses instead of waiting for an entire line.
   b. Detect control key combinations (Ctrl+A = ASCII 0x01, Ctrl+E = ASCII 0x05).
   c. Maintain a cursor index in your input buffer. Update it when the user presses these keys.
   d. Redraw the line (or move the cursor) accordingly using ANSI escape sequences.
   e. Use read() to capture input.
9. **Interrupting commands running in your shell (using signal call)**
   a. 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".
   b. 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.
10. **Implementing a searchable shell history**
    a. Maintain a history of the last 10000 commands run in your shell (hint: check how bash saves a history in a file)
    b. Implement a command "history" which will show the most recent 1000 commands.
    c. 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.
    d. 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.
    e. Otherwise print "No match for search term in history".

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

11. **Implementing auto-complete feature for file names**
    a. Implement an auto-complete feature for the shell for the file names in the working directory of the shell.
    b. 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:
        i. 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.
        ii. 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.
        iii. 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.

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 folder for the project, and name it *<roll no>project.* Zip it and upload it as *<roll no>_project.zip*. It should contain
  - The code
  - A README file telling how to compile and run your code.
  - A DESIGNDOC file stating your design of implementing each feature of the shell. Include a separate section for each feature you added.