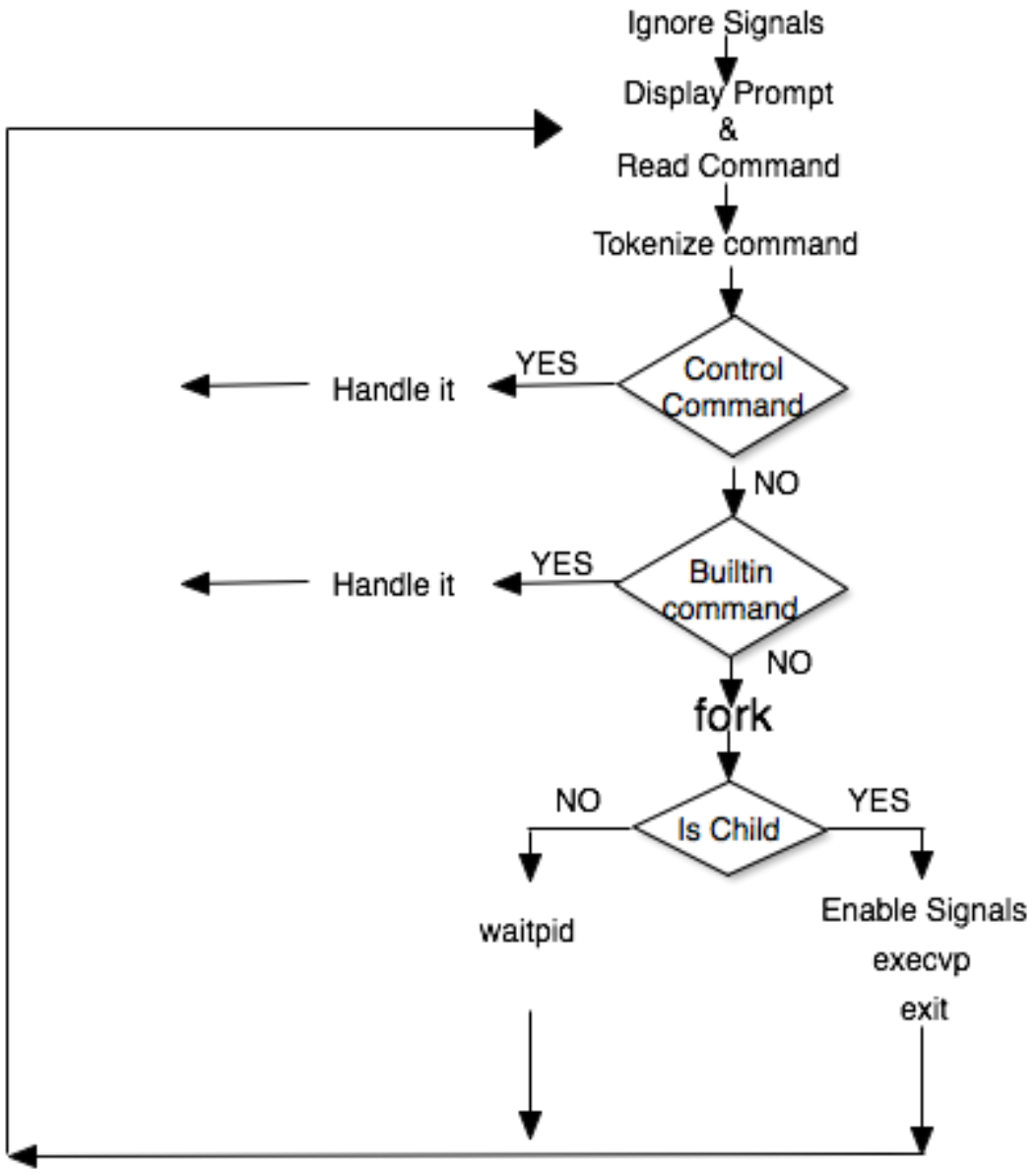
Creating UNIX Shell (100 marks)

Problem Statement

The purpose of this assignment is to give you practice in the use of UNIX system calls and writing a command interpreter on top of UNIX. The Shell as described by Richard Stevens in his book Advanced Programming in the UNIX environment is a command-line interpreter that reads user input and execute commands. For some it is a user interface between user and the internals of operating system whose job is to intercept user's command and then trigger system calls to ask OS to accomplish the user's tasks. For me it is a program executing another program.

A shell mainly consists of two parts: parsing user requests and accomplishing user request with system call's help. In this assignment you will write your own command shell to gain experience with some advanced programming techniques like process creation and control, file descriptors, signals, I/O redirection and pipes. You will do increment programming and develop different version of your own UNIX shell whose specifications are mentioned in the following paragraphs. A simple flow chart of the expected shell is given below:



Version01:

[30]

The first version of **shell** has been discussed and developed in the video lecture available online at (https://youtu.be/F7oAWvh5J_o?si=_DK3xzetUApoysV-). It has the following capabilities/characteristics:

a. The shell program displays a prompt, which is **PUCITshell:-**.You may like to indicate other things like machine name or username or any other information you like. Hint: **getcwd()**
b. The shell allows the user to type a string (command with options and arguments) (if any) in a single line. Parse the line in tokens, **fork,** and then pass those tokens to some **exec** family of function(s) for execution. The parent process waits for the child process to terminate. Once the child process terminates the parent process, i.e., our **shell** program again displays the prompt and waits for the user to enter the next command.
c. The shell program allows the user to quit the shell program by pressing **<CTRL+D>.**

Version02:

[20]

This version should be able to redirect **stdin** and **stdout** for the new processes by using **<** and **>**. For example, if the user gives the command **$mycmd < infile > outfile**, the shell should create a new process to run **mycmd** and assign **stdin** for the new process to **infile** and **stdout** for the new process to **outfile**. Hint: Open the **infile** in read-only mode and the **outfile** in write-only mode and then use **dup2()**. For your ease, you may assume that each item in the command string is separated on either side by at least one space. This version should also handle the use of pipes as we all are used to of it. For example, the first sample command will create a copy of **file1.txt** with the name of **file2.txt.** The second command will print the number of lines in the file **/etc/passwd** on **stdout**.

```
$ ./myshellv2
PUCITshell@/home/arif/:- cat < file1.txt > file2.txt
PUCITshell@/home/arif/:- cat /etc/passwd | wc
  96     265    5925
PUCITshell@/home/arif/:-
```

Version03:

[10]

This version should be able to place commands (external only) in the background with an **&** at the end of the command line. Running a process in the background means you start it, the prompt returns at once, and the process continues to run while you use the shell to run other commands. This one sounds tricky, but the principle behind it is surprisingly simple. Need to handle signals and avoid zombies. Sounds like an adventure movie.

```
$ ./myshellv3
PUCITshell@/home/arif/:- find / -name f1.txt &
[1] 1345
```

Version04:

[20]

This version should allow the user to repeat a previously issued command by typing **!number**, where the number indicates which command to repeat. **!-1** would mean to repeat the last command. **!1** would mean repeating the command numbered 1 in the list of commands maintained in your history file. Your history file should keep track of the last 10 commands only and then start overwriting. (Coding gurus can try using the up arrow key and the down arrow key to navigate your own maintained history file Hint: **readline()**)

Version05:

[20]

Built-in commands are different from external commands. An external command is a binary executable, which the shell searches on the secondary storage and then does **fork** and **exec** to execute it. On the contrary, the code of a built-in command is part of the shell itself. So before calling **fork** and **exec**, you have to see if the command is built into the shell. This version should allow your shell program to use some of the built-in commands like

- **cd:** should change the working directory
- **exit:** should terminate your shell
- **jobs**: provide a numbered list of processes currently executing in the background
- **kill**, should terminate the process number in the list of background processes returned by jobs by sending it a SIGKILL signal. Hint**: kill(pid, SIGKILL**)
- **help**: lists the available built-in commands and their syntax

Version 06 (BONUS): **[10]**

Like any programming language, a UNIX shell has variables. You can assign values to variables, retrieve values from variables, and list variables. The shell includes two types of variables: local/user-defined and environment variables. This version should add the functionality of accessing and changing user-defined and environment variables. To add variables to your shell, you need a place to store these names and values. This storage system must distinguish local variables from global variables. You can implement this table with a linked list, a hash table, or a tree (the choice is yours ☺). Beginners like me can use an array of structs

```
struct var{
      char *str;         //name=value string
      int global;       //a Boolean
}
```

Note: Read carefully the submission instructions on the next page.

Submission Instructions:

- **What to Submit:**
  - Source Code files:
    - Design your code to be in multiple `.c` files for better understanding.
    - Your code needs to be properly commented on and has necessary error checking.
  - README file:
    - There should be a README file that should state your code status and the bugs you have found
    - List the features you have implemented, especially any additional features not mentioned in the assignment.
    - Acknowledge the helpers or the Internet resources if you have them.
- **How to Submit:**
  - **The assignment must be submitted by November 4, 2024, before the lab session begins.**
  - Create a private repository named **`asgn_1_yourrollNo`** (all lowercase) on your own GitHub/bitbucket account.
  - Make your course instructor (arif@pucit.edu.pk) and course TAs the members of your repository (Access level minimum read).
  - You have to commit your code to the created repository after completion of every version of the above programs.
  - The viva/evaluation will be conducted in the lab on November 4. Any case of cheating will result in a zero for this assignment and may also lead to a reduction in your overall course grade.

TIME IS JUST LIKE MONEY.
THE LESS WE HAVE;
THE MORE WISELY WE SPEND IT.
Manage your time and Good Luck