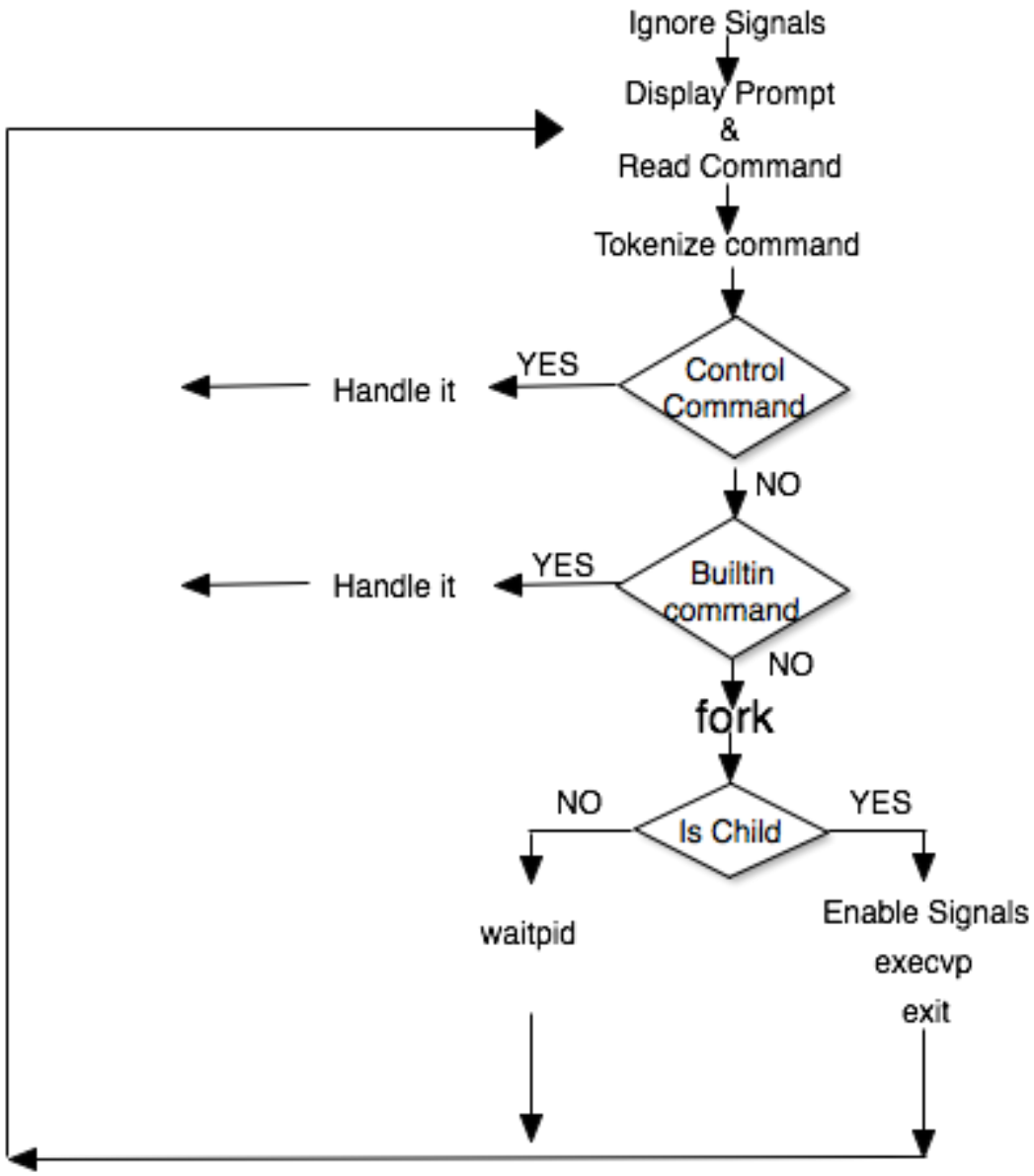
Creating UNIX Shell (100 mks)

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:
The first version of **cs525shell** has been discussed in the class and developed in the video lecture available online at (**http://www.arifbutt.me/lec22-design-code-unix-shell-utility-arif-butt-pucit/**). It has the following capabilities/characteristics:

a. The shell program displays a prompt, which is **cs525@pwd**, i.e., the present working directory. You may like to indicate other things like machine name or username or any other information you like. Hint: **getcwd()**
b. The **cs525shell** allow 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 **cs525shell** program again displays the prompt and waits for the user to enter next command.
c. The shell program allows the user to quit the shell program by pressing **<CTRL+D>**
d. If the user run the **less /etc/passwd** program and press **<CTRL+C>**, i.e., send **SIGINT** to the **less** program. Both the **less** program and the shell should not terminate. Rather the signal should be delivered to the **less** program only and not to **cs525shell**

```
$ ./cs525shellv1
cs525shell@/home/arif/:- ls -l /etc/passwd
 -rw-r--r--  1  root root 2021  Feb 7  07:35 /etc/passwd
cs525shell@/home/arif/:-
```

Version02:
This version should be able to redirect **stdin** and **stdout** for the new processes by using **<** and **>**. For example, if the user give 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**.

```
$ ./cs525shellv2
cs525shell@/home/arif/:- cat < file1.txt > file2.txt
cs525shell@/home/arif/:- cat /etc/passwd | wc
  96     265    5925
cs525shell@/home/arif/:-
```

Version03:
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. The regular shell also allows the user to separate commands on a single line with semicolons. This version should also support executing multiple commands on a single line if a semicolon separates them (;). Hint: try **waitpid(pid, status, options)**

```
$ ./cs525shellv3
cs525shell@/home/arif/:- find / -name f1.txt &
[1] 1345
cs525shell@/home/arif/:- echo "PUCIT" ; date ; pwd
PUCIT
Sat Mar 16 18:27:24 PKT 2021
/home/arif
cs525shell@/home/arif/:-
```

Version04:
This version should allow the user to repeat a previously issued command by typing **!number**, where number indicates which command to repeat. **!-1** would mean to repeat the last command. **!1** would mean repeat the command numbered 1 in the list of command maintained in your history file. Your history file should keep track of last 10 commands only and then start overwriting. (Coding gurus can try using up arrow key and the down arrow key to navigate your own maintained history file Hint: **readline()**)

Version05:
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 do **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 numbered 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

Version06:
This version add the functionality of adding an **if** control structure in your shell (no else required). The **if** control structure works as follows:

- The shell runs the command that follows the word **if**
- The shell checks the exit status of the command
- An exit status of 0 means success, nonzero means failure
- The shell executes commands after the **then** line if success
- The shell executes commands after the **else** line if failure
- The keyword **fi** marks the end of **if** block

Version07 (Bonus):
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, a tree (choice is yours ☺). Beginners like me can use an array of structs

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

Version 08 (Bonus):
This version should add the feature of file/command name completion on tab key as the bash shell do. The GNU **readline** library makes this easy

Note: Read carefully submission instructions on next page.

Submission Instructions:

- **What to Submit:**
  - Source Code files:
    - Design your code to be in multiple **`.c`** files for better understanding.
    - Commit your code (`.c`) files properly.
    - Your code needs to be properly commented and have necessary error checking.
  - README file:
    - There should be a README file that should state your code status and bugs you have found
    - List the features you have implemented, especially any additional features not mentioned in assignment.
    - Acknowledge the helpers or the Internet resources if you have.

- **How to Submit:**
  - Create a private repository named **`asgn_3_yourrollNo.git`** (all lower case) on you own bitbucket account
  - Make your course instructor (arif@pucit.edu.pk) and course TA (mscsf19m015@pucit.edu.pk) the member of your repository (Access level minimum read).
  - You have to commit your code (.c files) on created repository after completion of every version of above programs. Every version must have its own README file describing all the functionalities of the code.
  - **Dead Line for uploading all the versions of this assignment on bitbucket account is Sunday, April 18, 2021 till 11:59pm.**
  - Any cheating case will result in a zero mark in this assignment and may be a grade down in the overall course as well.

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