UTD -Summer 2025
CS 4348 Operating Systems
Project 1 - Implementing a Unix Shell
Due Date: June 22, 2025
100 Points

This project aims to help you understand how process control works by implementing a simple shell user interface. This covers shell variables, the interactions between parent and child processes, the procedures required to generate a new process, and a primer on validating and parsing user input.
For this project, you may work in **a group of two students**. You must create your shell on a UTD CS Linux class server, which can be accessed at cs1.utdallas.edu.

You must create and execute a simple shell interface that allows for the running of other programs and a set of built-in functions, as detailed below. The shell must be resilient; for instance, it shouldn't crash in any situation other than hardware failure.

The shell (command line) that you need to implement is simply a program that repeatedly solicits input from the user, performs tasks on their behalf, resets itself, and then asks for input again. For instance, consider this:
while(1){:
    This creates an infinite loop, meaning the shell will continue to run indefinitely until explicitly told to exit.
    /* Get user input */:
    This represents the shell prompting the user for a command and receiving their input.
    /* Exit? */:
    This step would involve checking the user's input to see if it's an exit command (like exit, quit, or Ctrl+D).
    /* Do something with input */:
    This is where the shell would parse the user's command, potentially execute a program, or perform other actions based on the command.
}
In essence, the code represents the continuous cycle of:

1. **Prompting the user:** The shell displays a prompt to indicate it's ready for input.
2. **Receiving input:** The shell reads the user's command from the standard input.
3. **Processing input:** The shell analyzes the command, possibly executing a program or performing built-in actions.
4. **Reseting for next input:** The loop resets, ready to accept the next command.

• The following is an example of what the prompt should resemble:
`ProjSh$`
• To prevent the user's input from visually running into the prompt, a space should be included after the dollar symbol.

The shell must extract the command name and arguments into "tokens" before it can start running a command. To enable you to parse each token in order, it would be useful to keep them in an array. The name of the program we want to run will always be the first token in our shell, and any other tokens, possibly even the first one, will be arguments to that program.

The following presumptions should be noted:
- No leading whitespace
- One space separates the command line tokens.
- No trailing whitespace
- You can assume that each token is no longer than 80 characters.
- You can assume that a command will have at most 10 space-separated tokens

User input (the command and its arguments) can be collected from the screen and saved as a string (C character array) using the C library method fgets(). For more details, see the fgets man pages.

How to handle non built-in command (i.e., external commands)?
When the shell recognizes that a non built-in command (i.e., external commands, some examples of such commands are listed below) has been typed, you will need to utilize the fork() system call to create a new process because running another program requires generating a new one. The execvp() system function must be used by the newly created child process to run such an none built-in command. Finally, before releasing the child's resources using the waitpid() system call, the parent (shell) process must wait for the child process to finish.
Examples non built-in command (i.e., external commands):
- ls: Lists files and directories in a given location.
- cat: Concatenates and displays the contents of files.
- mkdir: Creates new directories.
- cp: Copies files.
- mv: Moves or renames files or directories.
- rm: Removes files or directories.
- grep: Searches for text patterns within files.
- find: Locates files based on specified criteria.
- tar: Creates and extracts archive files.
- df: Displays the available disk space.
- free: Displays memory usage information.
- who: Lists currently logged-in users.
- chmod: Changes file permissions.

If an error occurs, though, the execvp() system function could return. Your shell ought to display an error, reset, and request fresh input if it does. Here is an illustration:

```
ProjSh$ hahaha -a
Error: Command could not be executed
ProjSh$
```

How to handle built-in commands?

Your shell must also implement two built-in commands (namely: ***exit*** and ***cd***). In this case, you should not use fork(), execvp() , and waitpid() to run either of these two commands. Rather, one of the following two subroutines' implementations should be called by your shell:

1. ***exit*** Implementation:
   - When the shell encounters ***exit***, it should terminate the program execution.
   - This can be achieved by using exit(0) (or exit(<exit_status>) to return a non-zero status) in the C code of the shell.
   - The shell would not need to call fork() and execvp() as it is directly terminating itself.

```
ProjSh$ exit
exit
(shell exits)
```

2. ***cd*** Implementation:
   - The ***cd*** command should change the current working directory.
   - The shell needs to access the environment variables (specifically PATH and potentially HOME) for directory navigation.
   - It should use the chdir system call (available in C/C++) to change the current working directory.
   - If the directory to change to is not specified (e.g., cd), the shell might need to access environment variables to determine the user's home directory.

```
ProjSh$ pwd
/user/xyz/os/project1
ProjSh$ cd ..
ProjSh$ pwd
/user/xyz/os
ProjSh$ cd project1
ProjSh$ pwd
/usr/xyz/os/project1
```

Create a README file

Please create a README text file that contains the following:

- The names of all the members in your group
- A listing of all files/directories in your submission and a brief description of each
- Instructions for compiling your programs
- Instructions for running your programs/scripts
- Any sources you used to help you write your programs/scripts

Project Submission

Both the C code file and the README text file should be turned in. Navigate to the class eLearning page and click on the Project Submission link on the left. Follow the links to submit your C code file and README file separately.