Project: Unix-like Shell

Due: Tuesday, December 7th at 11:59pm

No extensions can be used on this assignment

CS Linux Machine

You will need access to an Linux based machine when working on your project. You should not test your programs on macOS or Windows Linux because these operating systems do not provide all utility commands necessary for completing this and possibly future assignments. Additionally, if they do provide a command then it may not contain all options that a Unix-like system provides. **We will use and grade all assignments on the CS Linux machines and all programming assignments must work correctly on these machines**. However, you can work locally on a Unix or Unix-like machine but ensure that you test your final solutions on a CS Linux machine.

Please follow the instructions provided here

- Using Visual Studio Code and SSH

to easily access and work remotely on a CS Linux machine.

If you have any difficulties working or accessing a machine then please ask your question on Ed Discussion.

Creating Your Private Repository

For each assignment, a Git repository will be created for you on GitHub. However, before that repository can be created for you, you need to have a GitHub account. If you do not yet have one, you can get an account here: https://github.com/join.

To actually get your private repository, you will need this invitation URL:

- `Project invitation (Posted on Ed) <>`__

When you click on an invitation URL, you will have to complete the following steps:

1. You will need to select your CNetID from a list. This will allow us to know what student is associated with each GitHub account. This step is only done for the very first invitation you accept.

> **Note**
> If you are on the waiting list for this course you will not have a repository made for you until you are admitted into the course. I will post the starter code on Ed so you can work on the assignment until you are admitted into the course.

2. You must click "Accept this assignment" or your repository will not actually be created.
3. After accepting the assignment, Github will take a few minutes to create your repository. You should receive an email from Github when your repository is ready. Normally, it's ready within seconds and you can just refresh the page.
4. **You now need to clone your repository (i.e., download it to your machine).**
   - Make sure you've set up SSH access on your GitHub account.
   - For each repository, you will need to get the SSH URL of the repository. To get this URL, log into GitHub and navigate to your project repository (take into account that you will have a different repository per assignment). Then, click on the green "Code" button, and make sure the "SSH" tab is selected. Your repository URL should look something like this: git@github.com:mpcs51082-win23/proj-GITHUB-USERNAME.git.
   - If you do not know how to use `git clone` to clone your repository then follow this guide that Github provides: Cloning a Repository

If you run into any issues, or need us to make any manual adjustments to your registration, please let us know via Ed Discussion.

Unix-like System shell

The final project gives you the opportunity to show me what you learned in this course and to build your own emulated Unix-like system. In particular, your Unix system will include its own users, `/proc` virtual filesystem, and its own implementation of a shell. The main focus of the project will be building the shell; however, shell commands will interact with user specific directories, `/proc` filesystem and `etc` directory for the shell. Please note that as you implement this project there will be certain implementations the differ from an actual Unix-like system so please keep that in mind. We will point that in the specification.

The first task is understanding the repository structure, which is described in the next section.

Task 0: Repository Structure

Inside your repository, you will see the following structure (directories are **bold**)

- **proc**
  - (empty at the moment)
- **home**
  - **root**
    - .tsh_history
- **etc**
  - passwd
- tsh.c

The contents of each directory is explained below:

- **proc** - represents the `proc` virtual filesystem for the shell. This directory will contain `PID` directories each with their own `proc/PID/status` file. We will discuss the contents of each of these files in a later section.
- **home** - represents the home directories of the users. Similar to a Unix system, each user will have a separate home directory. Our shell won't have permissions so everything is accessible by all users. Inside each home directory, will be a `.tsh_history` file that contains the last 10 commands ran by the user before they quit/logged out of the shell. You can actually see this yourself by typing in `history` on the CS linux servers; however, this shows much more than the last 10 commands.
- **etc** - contains only one file which is the `etc/passwd` file. As a reminder, this file contains information about users. Unlike in the normal `/etc/passwd` that contains multiple fields, the shell's passwd file will have the following structure

  ```
  username:password:home_directory_path
  ```

  Each line will represent a single user of the shell. Currently there is only user inside the file `root`

  ```
  root:pass:/home/root
  ```

  Unlike in a normal Unix system, we do embed the actual password of the user inside the `passwd` for security purposes but rather in encrypt the password within `/etc/shadow`. However, for our shell we will keep things simple (in a vey unsecure way) by having the actual passwords in the `etc/passwd` file. Additionally, `root` normally does not have a home directory but for the purposes of this assignment it will.
- **tsh.c** - this is where you will implement the entire shell.

Task #1: Understanding the `tsh.c` File

Looking at the `tsh.c` file, you will see that it contains a functional skeleton of a simple Unix shell. To help you get started, we have already implemented the less interesting functions. Your assignment is to complete the remaining empty functions listed below along with writing additional functions and possibly struct definitions.

- eval: Main routine that parses and interprets the command line.
- builtin_cmd: Recognizes and interprets the built-in commands: quit, fg, bg, and jobs.
- login: will login a specific user and return the username of the user logged in.
- do_bgfg: Implements the bg and fg built-in commands.
- waitfg: Waits for a foreground job to complete.
- sigchld handler: Catches SIGCHILD signals.
- sigint handler: Catches SIGINT (ctrl-c) signals.
- sigtstp handler: Catches SIGTSTP (ctrl-z) signals.

Please take sometime to look over the comments and code in the file to make sure you understand how to use them. **You may need to add/modify these functions. Additionally you may need to write additional helper functions, define structs, global variables to implement all aspects of the project.**

Task #2: User Login

The compiling and running of the shell can be done as follows

```
$ gcc -std=gnu11 -o tsh tsh.c
$ ./tsh
```

When the shell begins running it must prompt the user to enter in a username (`username:`) and password (`password:`) in order to start the shell. The shell must then perform user authentication by verifying the username and password matches one inside the `etc/passwd` file. if their is a match then it will begin the shell `tsh>`; otherwise, if the user entered in an incorrect username and/or password then the shell responds with `"User Authentication failed. Please try again."`. The shell will continuously keep asking the user to login until an authentication is successful or the user enters in the command `quit`, which terminates the shell. Here are a few example runs

```
$ ./tsh
username: root
password: badPass
User Authentication failed. Please try again.
username: lamonts
password: pass
User Authentication failed. Please try again.
username: quit
$ ./shell
username: root
password: pass
tsh>
```

Task #3: The tiny (`tsh`) Shell

The main objective for the project is to implement `tsh` (tiny shell). As a reminder, a *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a *command line* on **stdin**, and then carries out some action, as directed by the contents of the command line. The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the child. The child processes created as a result of interpreting a single command line are known collectively as a **job**. In general, a job can consist of multiple child processes connected by Unix pipes.

If the command line ends with an ampersand `"&"`, then the job runs in the *background*, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

For example, typing the command line

```
tsh> jobs
```

causes the shell to execute the built-in `jobs` command. Typing the command line

```
tsh> /bin/ls -l -d
```

runs the `ls` program in the foreground. Alternatively, typing the command line

```
tsh> /bin/ls -l -d &
```

runs the `ls` program in the background.

> **Note**
> Notice that we had to provide the full bath to ls (i.e., `/bin/ls`) instead of saying `ls`. You will need to provide the full path to the executables for your shell. I would recommend the `/bin/sleep/` command to help with testing background jobs.

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Typing ctrl-c causes a SIGINT signal to be delivered to each process in the foreground job. The default action for SIGINT is to terminate the process. Similarly, typing `ctrl-z` causes a `SIGTSTP` signal to be delivered to each process in the foreground job. The default action for `SIGTSTP` is to place a process in the stopped state, where it remains until it is awakened by the receipt of a `SIGCONT` signal. Unix shells also provide various built-in commands that support job control. For example:

- `jobs`: List the running and stopped background jobs.
- `bg <job>`: Change a stopped background job to a running background job.
- `fg <job>`: Change a stopped or running background job to a running in the foreground.
- `kill <job>`: Terminate a job.

`tsh` Specification

Your tsh shell must have the following features:

1. The prompt should be the string `"tsh> "`.
2. The command line typed by the user should consist of a name and zero or more arguments, all separated by one or more spaces. If name is a built-in command, then `tsh` should handle it immediately and wait for the next command line. Otherwise, `tsh` should assume that name is the path of an executable file, which it loads and runs in the context of an initial child process (In this context, the term job refers to this initial child process). When a user logouts, the last 10 commands need to be saved in the history file (i.e., `.tsh_history`) for the user such that if the user logs back in that file should be reloaded with the previous 10 commands ran.
3. Any process (i.e. a foreground or background) started, must have an entry in the `proc` directory. Specifically, the directory created will be `proc/PID` where `PID` is the unique identifier for the process. Each `proc/PID` directory will only contain a single file `status` that has the following structure

   ```
   Name:  <name of process, argv[0]>
   Pid:   <unique identifier for the process>
   PPid:  <unique identifier for the parent process>
   PGid:  <unique identifier for the process group>
   Sid:   <unique identifier for the session leader id>
   STAT:  <process state, two letter state see m7 Slide 20>
   Username: <the name of the user who owns this process>
   ```

   As the process is running, the only line changing in this file is the `STAT` line. The shell will always be the session leader and is required to have an entry in the `proc` directory. If a process changes it's state then their `proc/PID/status` file must be updated. If a process is terminated then the `proc/PID` directory is removed.
4. `tsh` need not support pipes (`|`) or I/O redirection (`<` and `>`).
5. Typing `ctrl-c` (`ctrl-z`) should cause a `SIGINT` (`SIGTSTP`) signal to be sent to the current foreground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.
6. If the command line ends with an ampersand `&`, then `tsh` should run the job in the background. Otherwise, it should run the job in the foreground.
7. Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer assigned by tsh. JIDs should be denoted on the command line by the prefix `'%'`. For example, `"%5"` denotes JID 5, and "5" denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.). You can assume the job list will never be filled completely.
8. `tsh` should support the following built-in commands:
   - `quit` command terminates the shell immediately, which means you need to terminate all running and stopped jobs along with deleting the associated `/proc` files.
   - `logout` command logs out the user from the shell and then terminates the shell. If there are any suspended (i.e., stopped) processes then the command print `"There are suspended jobs."` and does not log the user out. The user must `kill` or bring them back into the foregrouund to allow them to terminate. Once all jobs are no longer suspended then running the `logout` command terminates the shell. If there are any running jobs and the user wishes to logout then the shell must terminate the running jobs and then logout.
   - `history` shows the last 10 commands ran by the user, each numbered on a separate line. The first represents the oldest command and the last line represents the most recently ran command. Each line is numbered starting from 1 up to N where N is at most equal to 10.
   - `jobs` command lists all the jobs currently active. I will provide the implementation for this as follows

     > ```
     > if (strcmp(argv[0],"jobs") == 0) {
     >    listjobs(jobs);
     > }
     > ```

   - `!N`, where `N` is a line number from the history command - reruns the `N` command from the user's history list. Do not add `!N` command to the history of the user.
   - The `bg <job>` command restarts `<job>` by sending it a `SIGCONT` signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
   - The `fg<job>` command restarts `<job>` by sending it a `SIGCONT` signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.
   - `adduser new_username new_password` commands creates a new user for the shell. This command can only be done if the `root` user is logged in. If any other user tries to run this command then the command returns `"root privileges required to run adduser."` Otherwise, the shell will create an entry for the new user inside the `etc/passwd` file and create a new home directory (i.e., `home/new_username`) and an empty `.tsh_history` file. We do not have a delete user command for the shell. Print an error message if a user has already been added to the shell.
9. `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then tsh should recognize this event and print a message with the job's PID and a description of the offending signal.

Hints & Tips

- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The `WUNTRACED` and `WNOHANG` options to `waitpid` will also be useful.
- When you implement your signal handlers, be sure to send `SIGINT` and `SIGTSTP` signals to the entire foreground process group, using `"-pid"` instead of `"pid"` in the argument to the kill function.
- **One of the tricky parts of the assignment is deciding on the allocation of work between the `waitfg` and `sigchld` handler functions. We recommend the following approach:**
  - In `waitfg`, use a `whilte(1)` loop around the `sleep` function.
  - In `sigchldhandler`, use exactly one call to `waitpid`.

  While other solutions are possible, such as calling waitpid in both waitfg and sigchld handler, these can be very confusing. It is simpler to do all reaping in the handler.
- In eval, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmas``k` after it adds the child to the job list by calling `addjob`. Since children inherit the blocked vectors of their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program.

  The parent needs to block the `SIGCHLD` signals in this way in order to avoid the race condition where the child is reaped by sigchld handler (and thus removed from the job list) before the parent calls `addjob`.
- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell. Stick with simple text-based programs such as `/bin/ls`, `/bin/ps`, and `/bin/echo`.
- When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the foreground process group. Since typing ctrl-c sends a `SIGINT` to every process in the foreground group, typing ctrl-c will send a `SIGINT` to your shell, as well as to every process that your shell created, which obviously isn't correct.

  Here is the workaround: After the fork, but before the `execve`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type ctrl-c, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).
- The `remove(path)` and `rmdir(path)` functions will be helpful with deleting files and removing directories
- You may want to define a history array as follow `char history[MAXHISTORY][MAXLINE];` where `MAXHISTORY=10` to easily store the history of the current user. For example, You can do something like `strcpy(history[1], "/bin/ls ls -l")` to easily store the history for the user.
- To update the `passwd` file use the `"a"` flag to store append to the file when adding a new user.
- Make sure to not add a new user if they already exist in the `passwd` file. You can display an error message such as `User already exists.` if they do appear.
- built-in commands do not need to have proc file since it's handled by the shell itself.

Project Grading

When grading the project, we will use the following as the criteria for getting specific grades for the project:

High A Range (96-100)

The project is fully-working based on the specification above. All job-control is working (i.e., signals handlers) and the built-in commands are working as specified.

High B and Low A Range (86-95)

Job-control is working fully for foreground processes. Proc files are created/deleted correctly and history commands are saved correctly for foreground processes. All built-in commands are working correctly with the exception of handling background processes or signals (i.e., `bg`, `fg`, `jobs`, etc.). You do not need to have the signal handlers working fully/correctly to receive a grade in this range.

High C and Low B Range (75-85)

Job-control is not fully working for either background/foreground processes. However, significant progress has been made to get them working correctly. The user can login and the root user can create new users. The built-in commands are implemented with the exception of handling foreground and background processes or signals correctly (i.e., `bg`, `fg`, `jobs`, etc.).

Lower then Low C (74-0)

This is be graded on a case by case basis based on what is submitted; however, you cannot receive higher than a 75 on the project.

As you can see, We will be pretty lenient on grading the project. The ranges in the above categories are there because design and style will be a factor in the grading. Please make sure you have modular code (i.e., break you code down into functions). Even if you are missing edge cases in implementing a few features you can still receive a good grade in the above ranges.

Submission

Before submitting, make sure you've added, committed, and pushed all your code to GitHub. You must submit your final work through Gradescope (linked from our Canvas site) in the "Project" assignment page via two ways,

1. **Uploading from Github directly (recommended way)**: You can link your Github account to your Gradescope account and upload the correct repository based on the homework assignment. When you submit your homework, a pop window will appear. Click on "Github" and then "Connect to Github" to connect your Github account to Gradescope. Once you connect (you will only need to do this once), then you can select the repository you wish to upload and the branch (which should always be "main" or "master") for this course.
2. **Uploading via a Zip file**: You can also upload a zip file of the homework directory. Please make sure you upload the entire directory and keep the initial structure the **same** as the starter code; otherwise, you run the risk of not passing the automated tests.

> **Note**
> For either option, you must upload the entire directory structure; otherwise, your automated test grade will not run correctly and you will be **penalized** if we have to manually run the tests. Going with the first option will do this automatically for you. You can always add additional directories and files (and even files/directories inside the stater directories) but the default directory/file structure must not change.

A few other notes:

- You are allowed to make as many submissions as you want before the deadline.
- Please make sure you have read and understood our Late Submission Policy.
- Your completeness score is determined solely based on the automated tests, but we may adjust your score if you attempt to pass tests by rote (e.g., by writing code that hard-codes the expected output for each possible test input).
- Gradescope will report the test score it obtains when running your code. If there is a discrepancy between the score you get when running our grader script, and the score reported by Gradescope, please let us know so we can take a look at it.

Acknowledgments

The shell writeup and starter-code of the project comes from the "Shell Lab" from the course textbook *Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e)*.