CSCE 3600: Systems Programming
Major Assignment 2 – The Shell and System Calls

COLLABORATION
You should complete this assignment as a group assignment with the other members of your group as assigned on Canvas using our GitLab environment (i.e., your assigned group). If desired, you may submit only one program per group, but all source code *must* be committed in GitLab. Also, make sure that you list the names of all group members who participated in this assignment for each member to get credit.

BACKGROUND
A shell provides a command-line interface for users. It interprets user commands and executes them. Some shells provide simple scripting terms, such as `if` or `while`, and allow users to make a program that facilitates their computing environment. Under the hood, a shell is just another user program. The file `/bin/bash` is an executable file for the bash shell. The only thing special about your login shell is that it is listed in your login record so that `/bin/login` (i.e., the program that prompts you for your password) knows what program to start when you log in. If you run "`cat /etc/passwd`", you will see the login records of the machine.

PROGRAM DESCRIPTION

GROUP COLLABORATIVE PORTION

In this assignment, you will implement the shell “engine” as the “group” component, where all members are responsible for the following functionality:

- **A Command-Line Interpreter, or Shell**

  Your shell should read the line from standard input (i.e., interactive mode) or a file (i.e., batch mode), parse the line with command and arguments, execute the command with arguments, and then prompt for more input (i.e., the shell prompt) when it has finished.

  1. **Interactive Mode**

     In interactive mode, you will display a prompt (any string of your choosing) and the user of the shell will type in a command at the prompt.

  2. **Batch Mode**

     In batch mode, your shell is started by specifying a batch file on its command line. The batch file contains the list of commands that should be executed. In batch mode, you should not display a prompt, but you should echo each line you read from the batch file back to the user before executing it.

  You will need to use the `fork()` and not `exec()` family of system calls. You may not use the `system()` system call as it simply invokes the system’s `/bin/bash` shell to do all of the work.

You **may assume** that arguments are separated by whitespace. You do not have to deal with special characters such as `'`, `"`, `\`, etc. However, you will need to handle the redirection operators (`<` and `>`) and the pipeline operator (`|`), which will be specified in the "individual" portion of this assignment.

Each line (either in the batch file or typed at the prompt) may contain multiple commands separate with the semicolon (`;`) character. Each command separated by a `;` should be run sequentially (i.e. when you use semicolons, each command/program will run regardless if the preceding one fails), but the shell should not print the next prompt or take more input until all of these commands have finished executing (the `wait()` or `waitpid()` system calls may be useful here).

You **may assume** that the command-line a user types is not longer than 512 bytes (including the '`\n`'), but **you should not assume** that there is any restriction on the number of arguments to a given command.

INDIVIDUAL PORTION

In this assignment, each member of the group will implement the following components as defined below. This means that the individual group member responsible for each portion MUST commit in GitLab the code that supports their responsible area.

- **Built-In Commands**

  Every shell needs to support several built-in commands, which are functions in the shell itself, not external programs. Shells directly make system calls to execute built-in commands, instead of forking a child process to handle them. *Each group member is expected to implement 1 of the following built-in commands.*

  Note that the expectation for this assignment assumes that a group contains 4 students, but **if, for some reason, a team has only 3 students, then only 3 of the following built-in commands would need to be supported (i.e., 1 for each group member).**

  1. Add a new built-in `cd` command that accepts one optional argument, a directory path, and changes the current working directory to that directory. If no argument is passed, the command will change the current working directory to the user's `HOME` directory. You may need to invoke the `chdir()` system call.

  2. Add a new built-in `exit` command that exits from the shell itself with the `exit()` system call. It is not to be executed like other programs the user types in. If the `exit` command is on the same line with other commands, you should ensure that the other commands execute (and finish) before you exit your shell.

     These are all valid examples for quitting the shell:

     ```
     prompt> exit

     prompt> exit;
     cat file1 prompt> cat file1; exit
     ```

3. Add a new built-in `path` command that allows users to show the current pathname list, append one pathname, or remove one pathname. In your shell implementation, you may keep a data structure to deal with the pathname list. If you do not use `execle()` or `execve()` that allows you to execute with your own environment variables, you will need to add it to the "real" PATH environment variable for executables in the path to work correctly. The initial value of path within your shell shall be the pathname list contained in the `PATH` environment variable. Implement the `path` command as follows:

   - `path` (without arguments) displays the pathnames currently set. It should show pathnames separated by colons. For example, "`/bin:/user/bin`".

   - `path + ./bin` appends the pathname to the `path` variable. You may assume that only one pathname is added at a time.

   - `path - ./bin` removes the pathname to the `path` variable. You may assume that only one pathname is removed at a time.

   You may assume that there are no duplicate pathnames present, being added, or being deleted. You will restore your `PATH` environment variable to its original state (i.e., before your shell was invoked) when the user exits your shell.

4. Add a new built-in `myhistory` command that lists the shell history of previous commands run in your shell (not the bash shell). Note that this does not have to work with the up-arrow key as in bash, but only with a new `myhistory` command run inside your shell. You may not make use of the `history` builtin command, but instead keep track of your history of commands in some sort of data structure. Your `myhistory` built-in command should support a history of 20 most recent commands (i.e., this means that the 21st command will overwrite the 1st command, for example).

   Your `myhistory` command should support a couple flags: `-c` to clear your `myhistory` list by deleting all entries, and `-e <myhistory_number>` to execute one of your twenty `myhistory` commands in your list.

• Redirection, Pipelining, Signal Control, and Alias Support:

1. Extend your shell with I/O redirection (mandatory for teams of 3 or 4)

When you start a command, there are always three default file streams open: `stdin` (maps input from the keyboard by default), `stdout` (maps output to the terminal by default), and `stderr` (maps error messages to the terminal by default). These and other open files may be redirected, or mapped, to files or devices that users specify.

Modify your shell so that it supports redirecting `stdin` and `stdout` to files. You do not need to support redirection for your shell built-in commands (i.e., `cd`, `exit`, `path`, and `myhistory`). You do not need to support `stderr` redirection or appending to files (e.g., `cmd3 >> out.txt`). You may assume that there will

always be spaces around the special characters < and >. Be aware that the "< file" or "> file" are not passed as arguments to your shell program.

Some redirection examples include:

```
$ cmd1 < in.txt
```

executes cmd1, using in.txt as the source of input, instead of the keyboard.

```
$ cmd2 > out.txt
```

executes cmd2 and places the output to file out.txt.

You will **need to understand Linux file descriptors** and use the open(), close(), and dup()/dup2() family of system calls. This portion of the project should only require implementing single input redirection and single output redirection (not both redirection or working with pipes).

2. *Extend your shell with pipelining (mandatory for teams of 3 or 4)*
   The command

   ```
   $ cmd1 | cmd2 | cmd3
   ```

   connects the standard output of cmd1 to the standard input of cmd2, and again connects the standard output of cmd2 to the standard input of cmd3 using the pipeline operator '|'. 5 of 9

   You will need to use the pipe() system call. Your shell should be able to handle up to three commands chained together with the pipeline operator (i.e., your shell should support up to two pipes pipelined together). This portion of the project should only require implementing support for a pipeline of 2 pipes/3 commands (no working with redirection).

   Your shell does not need to handle built-in commands implemented above (i.e., cd, exit, path, and myhistory) in pipeline.

3. *Support Signal Handling and Terminal Control (mandatory for team of 3 or 4)*

   Many shells allow you to stop or pause processes with special keystrokes, such as Ctrl-C or Ctrl-Z, that work by sending signals to the shell's subprocesses. If you try these keystrokes in your shell, the signals would be sent directly to the shell process itself. This is not what we want since, for example, attempting to Ctrl-Z a subprocess of your shell will also stop the shell itself. Instead, we want to have the signals affect only the subprocesses that our shell creates. To help you accomplish this, you might find the following helpful:

   a. Process Groups

      Every process has a unique process ID (i.e., pid). Every process also has a possibly non-unique process group ID (i.e., pgid) which, by default, is the same as the pgid of its parent process. Processes can get and set their

process group ID with the system calls `getpgid()`, `setpgid()`, `getpgrp()`, or `setpgrp()`.

Keep in mind that, when your shell starts a new program, that program might require multiple processes to function correctly. All these processes will inherit the same process group ID of the original process. So, it may be a good idea to put each shell process into its own process group for simplicity. When you move each subprocess into its own process group, the `pgid` should be equal to the `pid`

b. Foreground Terminal

Every terminal has an associated foreground process group ID. When you type Ctrl-C, your terminal sends a signal to every process inside the foreground process group. You can change which process group is in the foreground of a terminal with `tcsetpgrp(int fd, pid_t pgrp)`. The `fd` should be 0 for standard input `stdin`. You may want to perform a `man tcsetpgrp` command for additional help and implications on using this library call.

In your shell, you can use `kill -XXX pid`, where `XXX` is the human-friendly suffix of the desired signal, to send any signal to the process with process ID `pid`. Since you can use the signal function to change how signals are handled by the current process, your shell should basically ignore most of these signals, whereas your shell's subprocesses should respond with the default action. Be aware that forked processes will inherit the signal handlers of the original process. You may want to check out `man 2 signal` and `man 7 signal` for more information on this.

You want to ensure that each program you start is in its own process group. When you start a process, its process group should be placed in the foreground. Stopping signals should only affect the foreground program(s), not the background shell.

4. *Implement a new `alias` command (mandatory for teams of 4 and not required for teams of 3)*

Add a new built-in `alias` command that allows you to define a shortcut for commands by essentially defining a new command that substitutes a given string for some command, perhaps with various flags/options. The syntax is as follows: `alias alias_name='command'`. For example, you can define an alias with `alias 5='ls -al'`, so that the user can then enter `5` at the prompt to execute the `ls -al` command. Although there is an `alias` command in bash, you must implement your own version and not make use of the built-in `alias` bash command. Typically, `alias` is a built-in command, but since this affects how your shell functions (i.e., you simply cannot just pass an aliased command to the `exec` family system call as you are managing the execution which would otherwise result in a `not found` message.

Specifying `alias` with no arguments should display a list of all existing aliases. You may remove a single alias with the command `alias -r alias_name` or all defined aliases with `alias -c`. Be sure to handle the case if a user enters the `alias` command incorrectly. You can perform a `man alias` for help in understanding how this built-in command is used, but only the functionality specified here is required.

DEFENSIVE PROGRAMMING (GROUP COLLABORATIVE EFFORT)

Check the return values of all system calls utilizing system resources. Do not blindly assume all requests for memory will succeed and that all writes to a file will occur correctly. Your code should handle errors properly. Many failed system calls should not be fatal to a program. Typically, a system call will return –1 in the case of an error (`malloc` returns NULL on error).

An OS cannot simply fail when it encounters an error. It must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner. By "reasonable", this means that you should print a meaningful and understandable error message and either continue processing or exit, depending upon the situation.

Many questions about functions and system behavior can be found in the manual pages.

You should consider the following situations as errors – in each case, your shell should print a message to `stderr` and exit gracefully:

- An incorrect number of command line arguments to your shell program; and
- The batch file does not exist or cannot be opened.

For the following situation, you should print a message to the user (`stderr`) and continue processing:

- A command does not exist or cannot be executed.

Optionally, to make coding your shell easier, you may print an error message and continue processing in the following situation:

- A very long command line (over 512 characters including the '`\n`').

Your shell should also be able to handle the following scenarios, which are not errors (i.e., your shell should not print an error message):

- An empty command line (e.g., `$ ; ;`), which can simply be eaten with no output – note that this is different behavior than bash;
- Extra white spaces within a command line; and
- Batch file ends without `exit` command or user types '`Ctrl-D`' as a command in interactive mode.

In no case should any input or any command-line format cause your shell program to crash or exit prematurely. You should think carefully about how you want to handle oddly formatted command lines (e.g., lines with no commands between a semi-colon). In these cases, you may choose to print a warning message and/or execute some subset of the commands. However, in all cases, your shell should continue to execute.

REQUIREMENTS

Your code must be written in C and be invoked exactly as follows:

```
newshell [batchFile]
```

The command-line arguments to your shell are to be interpreted as follows:

- `batchFile`: an optional argument (indicated by square brackets as above). If present, your shell will read each line of the `batchFile` for commands to be executed. If not present, your shell will run in interactive mode by printing a prompt to the user at `stdout` and reading the command `stdin`.

For example, if you run your program as: (your_user_id is a dummy name, you can replace it with your own id)

```
newshell /home/your_user_id/csce3600/batchfile
```

then it will read commands from `/home/your_user_id/csce3600/batchfile` until it sees the `exit` command or EOF.

OPTIONAL SHELL FUNCTIONALITY

Teams who have completed all requirements for this program and are looking for an additional challenge may add the following optional functionality to gain bonus points added to your team's overall score:

- Allow the user to customize the prompt (+5 points).
- Any extra significant functionality that you might think of, but it MUST be approved by your instructor first (+5 or more points).

However, all required functionality must be implemented prior to attempting this extra credit work as no points will be given for attempting this functionality if all requirements have not been completed.

In other words, make sure your program is complete before attempting this extra credit.

GRADING

Your C program file(s), `README`, and makefile shall be committed to our GitLab environment as follows:

- Your C program file(s). Your code should be well documented in terms of comments. For example, good comments in general consist of a header (with your name, course

section, date, and brief description), comments for each variable, and commented blocks of code.

- A **`README`** file with some basic documentation about your code. This file should contain the following four components:
    - `Your name(s).`
    - Organization of the Project. Since there are multiple components in this project, you will describe how the work was organized and managed, including which team members were responsible for what components – there are lots of ways to do this, so your team needs to come up with the best way that works based on your team's strengths. Note that this may be used in assessment of grades for this project.
    - Design Overview: A few paragraphs describing the overall structure of your code and any important structures.
    - Complete Specification: Describe how you handled any ambiguities in the specification. For example, for this project, explain how your shell will handle lines that have no commands between semi-colons.
    - Known Bugs or Problems: A list of any features that you did not implement or that you know are not working correctly.
- A completed group assessment evaluation (given at a later date) for each team member. ***Please be aware that a student receiving a poor evaluation with regards to their performance on the team will have his/her grading marks reduced by an appropriate amount, based on the evaluation****. In addition, the rubric for this assignment allows modification of each individual's portion of the group grade to account for individual contribution to the group's submission, which may result in a member of the group receiving a much higher or much lower grade than other members of the group. This implies that* ***in the individual portion, each group member is responsible for "committing" his or her own code. Additionally, there are points allocated for participation and contribution to the overall project.***
- A `makefile` for compiling your source code, including a clean directive.
- Your program will be graded based largely on whether it works correctly on the CSE CELL machines, so you should make sure that your program compiles and runs on a CSE CELL machine.

Your program will be tested using a suite of about 20 test cases on the CSE machines, some of which will exercise your program's ability to correctly execute commands and some of which will test your program's ability to catch error conditions. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites.

SUBMISSION

- Each team will ensure that all source code and header files, the **`makefile`**, and the **`README`** file are committed to our GitLab repository by the due date and time. If desired, one student may submit all applicable files to Canvas, but it is not required and only code found in the GitLab repository will be graded.