Spring 2025
CSE486: Design of Operating Systems
Project 1: Shell

Due: January 29 (Wednesday) at 11:59 pm

1 Logistics

1. This project must be done in teams of **at most 4 members**. You may work alone or in a pair, but having fewer members will not grant extensions, bonuses, or advantages.
2. Any member in the group can submit the project report on behalf of the whole group. Please submit the report through blackboard.
3. For the following task, you must use the Linux server (`cse486.syr.edu`) dedicated to this course. You **must** know how to remotely log in to that machine, how to use the terminal, and how to copy files back and forth between your computer and the Linux server.
4. You must follow the submission instructions to avoid a loss of points.

2 Handout Instructions

1. Login into the class server `cse486.syr.edu` using the command:
   `ssh <netid>@cse486.syr.edu`
2. Copy the project 1 file from *cse486-proj* directory into your home directory by using the following command: `cp /srv/cse486-proj/proj1-shell.tar.gz /home/<netid>`
3. Untar the tarball using the following command: `tar -xvf proj1-shell.tar.gz`
4. The extracted directory is named *proj1-shell*, which contains some files, and one directory named *project-1*.
5. Take a look at the accompanying *project-1/README.md* for additional instructions such as ***how to build/run/test your program***
6. Unless stated otherwise, you **MUST NOT** modify/edit/rename/move other existing files under the *proj1-shell/* directory.
7. Unless stated otherwise, you **MUST NOT add any new** files (*e.g.*, additional C or header files) to any *project-*/* directory.
8. Unless stated otherwise, you **MUST NOT** use any additional C library (*e.g.*, math library) that requires modification to the compilation commands included in the existing Makefiles. No other library is needed for this project.

3 Submission Instructions

You must submit a **report** along with the **completed and commented project source code**. Please make sure that both your report and code submission adhere to the below instructions to ensure smooth grading.

3.1 Report Submission

Use the provided template (available in both LaTeX and Word format) on the course Blackboard to prepare your report. The report must contain the following:

1. A completed *Group Member* information table, including:
   - The group number that you enrolled on Blackboard for this project.
   - Each member's full name, NetID, major, and contribution factor (scale: 1-5).
2. The path to your zipped project code, including the file name (e.g., */home/schen154/proj1-shell/schen154-proj1.tar.gz*), specified directly below the *Group Member* information table.
3. For each task:
   - Provide screenshots of all changes youu made in the source code related to the task.
   - Include a detailed explanation of why you made these changes and their importance. This should demonstrate your understanding of the project.

Submit the report via Blackboard.

3.2 Code Submission

Follow the instructions below to submit your project code:

1. Go to (or change directory `cd` into) your *proj1-shell/* directory. Ensure that *proj1-shell/* is your current working directory.
2. The *proj1-shell/* directory contains a Makefile along with the *project-1/* directory. Run `make submit` to create a compressed archive named <*netid*>*-proj1.tar.gz*, where <netid>is your SU NetID.
3. To verify the contents of the compressed archive, run the command `tar tvzf <netid>-proj1.tar.gz` and confirm that it includes the solution file for the task.

Retain the compressed file in your directory and specify its full path, including the file name, in your report.

4 Project Description: Building a Simple Shell

For this project, you will implement a simple command line interpreter, commonly known as a **shell**. The shell should operate in this fundamental way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered; once the child process has finished, the shell prompts for more user input. The shell you implement will be similar but much simpler than the one you run daily in a Unix/Linux-based OS.

You are given a skeleton of the tiny shell in a file called project-1/tinyshell.c. Currently, it implements a few functionalities for you. For instance, the command parsing is already implemented. Therefore, the code can parse a given input command string. And we will discuss how to implement command execution during Labs. However, it does not fully implement the desired tiny shell. You have to complete the code to implement various shell functionalities. For your convenience, the source code includes some comments outlining what needs to be done and where to add your code. For example, tinyshell.c uses an explicit marker ("<<YOUR CODE GOES HERE>>" or "YCGH:") to denote where you should

write your code. tinyshell.c contains additional comments to guide you through the code and help you complete the implementation.

Your basic shell is an interactive loop program: it repeatedly prints a prompt "tinyshell> " (note the space after the > sign), parses the input that the user just typed followed by a newline (i.e., \n), executes the command specified by the input, and waits for the command to finish.

4.1 Compilation instructions

The final executable must be named tshell and compiled as follows

```
$ gcc -o tshell -Werror -Wall tinyshell.c
# altenatively, use 'make'; see project-1/README.md
$ ./tshell
tinyshell> exit
$ echo $?   # checking the exit status of the last command (i.e., ./tshell)
0           # 0 means ./tshell terminated gracefully, with no error
$
```

4.2 Task 1: Built-In Commands (12 points)

Most Unix shells include built-in commands such as `cd`, `pwd`, `echo`, `kill`, and others. Currently, your tinyshell does not support any built-in commands. To enhance the usability of your tinyshell, you are required to implement essential built-in commands that allow users to navigate and interact with the file system.

For this task, you must implement the following three built-in commands: `cd`, `pwd`, and `echo`. Their behavior should replicate that of the corresponding commands in a typical Unix/Linux shell. The detailed requirements for each command are as follows:

- `cd`:
  - When the user types `cd` without any arguments, your shell should change the working directory to the user's home directory, which is stored in the `$HOME` environment variable. Use `getenv("HOME")` to obtain the path to the home directory and then call `chdir()` to change the working directory.
  - If the user provides a path as an argument (e.g., `cd some/path`), use `chdir()` with the specified path to change the working directory.
- `pwd`:
  - When the user types `pwd`, your shell should use `getcwd()` to retrieve the current working directory and display the result.
- `echo`:
  - When the user types `echo` followed by a string or multiple arguments (e.g., `echo Hello World`), your shell should print the provided arguments as output, separated by spaces. (Hint: Iterate through all the arguments provided after `echo`, print them one by one, separating them with spaces.)
  - After printing all arguments, ensure a newline character (`"\n"`) is printed to mimic the behavior of the Unix/Linux `echo` command.
  - If the user types `echo` with no arguments, simply print a blank newline.

Below is an example of a successful implementation of `cd`, `pwd`, and `echo`:

```
$ ./tshell
tinyshell> cd /tmp
tinyshell> pwd
/tmp
tinyshell> cd
tinyshell> pwd
/home/YOUR_USER_NAME
tinyshell> echo Hello, World!
Hello, World!
tinyshell> echo

tinyshell> exit
$ echo $?
0
$
```

For other commands (e.g., `kill`, `exit`), your shell will process and execute them like any other regular command using `fork()` and `execvp()`. As long as the program exists in the environment variable `$PATH`, your shell will execute the command successfully.

4.3 Task 2: Redirection (5 points)

A shell user often prefers to redirect the output of a command or program to a file rather than displaying it on the terminal screen. This feature is commonly achieved using the > character, enabling the redirection of standard output to a specified file.

Currently, the provided shell does not support this functionality. In this task, you are required to implement output redirection in your shell. For example, when a user inputs a command like: `ls -la /tmp > output.txt`, your shell should redirect the output of the `ls` program to a file named `output.txt` in the current directory, without displaying anything on the terminal screen.

To implement this, you will need to:

- Use file management system calls, such as `open()` and `close()`, to handle the specified output file.
- Use the `dup2()` system call to replace the standard output file descriptor (`STDOUT_FILENO`) with the file descriptor of the opened file.
- If the specified output file already exists, the shell (child process) should overwrite its contents.

Additionally, your implementation must handle errors robustly:

- If the shell encounters an error when opening the specified output file, display a relevant error message using the provided macros (e.g., `PRINT_ERROR_SYSCALL`).
- Ensure that the shell exits with a status code of `1` when such an error occurs.

4.4 Task 3: Error Checking (3 points)

The provided shell code contains gaps that require proper implementation, as outlined in the tasks above. However, the code lacks adequate error-handling mechanisms.

This task is focused on ensuring that all code implemented in the above tasks includes robust error-checking and error-handling mechanisms to make the shell more reliable and user-friendly. To achieve this, you need to:

- Use the provided macros `PRINT_ERROR` and `PRINT_ERROR_SYSCALL(x)` to handle and display error messages:
  - `PRINT_ERROR`: Use this macro to display the generic error message: "An error has occurred"
  - `PRINT_ERROR_SYSCALL(x)`: Use this macro when handling errors from system calls (e.g., `fork()`, `execvp()`, `waitpid()`, `chdir()`, etc.).
- Add appropriate error-handling code to all newly implemented tasks. For example:
  - Check the return values of system calls and handle errors appropriately.
  - Exit the child process if an error occurs during command execution or file handling.
- Handle exit statuses correctly:
  - If the shell (parent process) terminates normally, it should exit with a status code of `0` using `exit(0)`.
  - If the shell exits due to an error, it must exit with a status code of `1` using `exit(1)`.

5 Advice

Some information about system calls:

- **chdir() :** `https://man7.org/linux/man-pages/man2/chdir.2.html`
- **getcwd():** `https://man7.org/linux/man-pages/man2/getcwd.2.html`
- **open() and close() :** The open() function opens the file for reading, writing, or both. It is also capable of creating the file if it does not exist. The close() function tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. You may want to look at the code given in Figure 5.4 of the Textbook `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf` and `https://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf`.
- **dup2():** `https://man7.org/linux/man-pages/man2/dup.2.html`
- **Error handling :** `https://www.tutorialspoint.com/cprogramming/c_error_handling.html`
- **C Preprocessor :**`https://gcc.gnu.org/onlinedocs/cpp/Macros.html`