PA-3: A Tiny-shell for Linux

Total points: **100 pts** Due on **Friday Mar 6, 2020 at 11:59 PM**

Instructor: Endadul Hoque

Logistics

1. This assignment must be done **individually, not in groups**.
2. For this programming assignment (PA-3), you have to download an archive called `student-pa3.tar.gz` from Blackboard. Upon extraction, you will see a directory named `student/`, which contains 1 directory named `task-1`.
3. For the task, you have to complete a `C` program provided in the respective directory. For instance, for **Task-1**, you have to complete `tinyshell.c` located in `task-1` directory.
4. Take a look at the accompanied **`README.md`** (if any) in the task directory for additional instructions, such as ***how to build/run/test your program***.
5. Unless stated otherwise, you **MUST NOT** modify/edit/rename/move other existing files under the `student/` directory.
6. For all the following tasks, you have to use the Linux server (`lcs-vc-cis486.syr.edu`) dedicated for this course. You **must** know how to remotely login to that machine, how to use `terminal` and how to copy files back and forth between your computer and the Linux server.
7. Some helpful links are listed in Appendix A. You may want to leverage them for solving PA-3.

Obtain the source code for this PA

Obtain `student-pa3.tar.gz` from `blackboard`. Unpack the package to find the skeleton code required for this PA. The extracted directory is named `student/`.

```
$ tar xzf student-pa3.tar.gz
```

Submission Instructions

1. Go to the **task** directory and clean the directory using `make clean`. For example,

   ```
   $ cd task-1/
   $ make clean
   ```

2. Change directory (`cd`) to your `student/` directory. Now `student/` should be your current working directory.
3. **Use your netid** to create a compressed archive (`<netid>-pa3.tar.gz`) of your solution directory. For instance, if your netid is `johndoe`, create a compressed archive as follows:

```
$ cd ..       # moving to the parent directory of student/
$ mv student/ johndoe-pa3/
$ tar czf johndoe-pa3.tar.gz johndoe-pa3/
```

4. Use **Blackboard** to submit your compressed archive file (*e.g.*, `johndoe-pa3.tar.gz`).

Grading

We will use an automated grading tool with pre-determined inputs to check your solution for correctness. ***Please make sure you follow the instructions provided throughout this handout; otherwise your assignment will not be graded at all by the auto grader.***

- You must make sure your source code complies with `gcc`, otherwise you will not obtain any points.
- You must not rename/move (say, `tinyshell.c`) files under `task-*/` directory.
- You must not modify/edit/rename/move other existing files/directories in `student/` or in its sub-directories.
- You must follow the naming convention for each task as directed, while creating files, directories, archives, and so on.
- Your solutions must execute on the Linux machine (`lcs-vc-cis486.syr.edu`), otherwise your solutions will not be considered for grading.
- Your must follow the submission instructions, or it will not be graded properly by the auto grader.

Task 1. Tiny shell (100 pts)

Why build another shell?

There are three objects to this PA:

- To familiarize yourself with the Linux programming environment (particularly, using **C**)
- To learn how processes are created, destroyed, and managed
- To gain exposure to the necessary functionality in shells

Overview

In this assignment, you will implement a command line interpreter, or as it is more commonly known, a **shell**. The shell should operate in this basic 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 to, but much *simpler* than, the one you run every day in an Unix/Linux-based OS.

Specifications

You are given a skeleton of the tiny shell in a file called `task-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. But it does not fully implement the desired tiny shell. For your convenience, it **includes some comments outlining what needs to be done or where you need to add your code**.

You **do not need to modify** the existing code. But if you modify, do it at your own risk; and you need to make sure it operates as expected and passes the provided testcases.

> Please note that passing the provided testcase(s) **do not necessarily mean** your solution is **fully correct**.

Feature 1.1. (10 pts) Basic shell and its prompt

Your basic shell is basically an **interactive** loop: 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.

Termination: Your shell repeats its execution cycle until the user types `exit`, **without anything following it other than a newline**. If `exit` is followed by any other token/string, then your shell should consider it as an error (More information on error checking in Feature 1.8). Your shell should also terminate if it reads **EOF** (typed using `ctrl+d` or if there is nothing to read when the input redirection is used).

Whenever the shell terminates normally (due to `exit` or `EOF`), it should return/terminate/exit with **exit-status `0`**. Otherwise, it should return/terminate/exit with a **non-zero status** (you should use `1`).

> The interactive loop and handling of `exit` command are already implemented in the code.

Compilation & Execution: The final executable should be named `tshell` and compiled as follows

```
$ gcc -o tshell -Werror -Wall -O3 tinyshell.c
$ ./tshell
tinyshell> exit
$
```

Feature 1.2. (20 pts) Command structure and its processing

Assumptions: Each command will be in a single line terminated by a newline character. You can safely assume that each line will have **a maximum of 1024 characters** including the newline character (*i.e.*, `'\n'`) at the end of the line.

Delimiters: You can assume that the command and each of its arguments will be separated by one or more `tab` or `space` characters. For example, "`ps -ef > filename &`" is a valid command; however, "`ps -ef>filename&`" is not. In fact, your shell should interpret the latter as a command with two tokens: "`ps`" and "`-ef>filename&`"; and it should attempt to execute `ps` with `-ef>filename&` as its argument.

Processing: Your shell should be able to parse the input (aka command) and run the program corresponding to the command. For example, if the user types `ls -la /tmp`, your shell should run the program `ls` with all

the given arguments and print the output on the screen (aka `stdout`). Note that the shell itself does not "implement" `ls` and many other commands like this. All it does is to find the executable in the environment `$PATH` (*e.g.*, for `ls`, the actual executable file is at `/bin/ls`) and create a new process to run the executable. However, there are few **exceptions** to this approach, which we will discuss later.

For each typed command, your shell should spawn a child process using `fork()` which will carry out the given command using `exec()` (or one of its variants). You may simply use `execvp()`. There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Secondly, it allows concurrency: multiple commands can be started and allowed to execute simultaneously.

> You may want to look at the Figure 5.2, 5.3 and 5.4 from Chapter 5 of the textbook `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf` for examples on how to use these system calls.

Feature 1.3. (20 pts) Built-in commands

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is, the command should not be executed like other programs (*e.g.*, `ls`). Instead, your shell will invoke your implementation of the built-in command.

For example, `exit` is a built-in command; when the user types `exit` command, your shell must terminate with `status 0`. One simple way to implement this is to call "`exit(0)`", which is a C library function (see `man exit`), or to use "`return 0`" if you are inside the `main()` function. Your shell already includes the `exit` built-in command.

Most Unix shells have many other built-in commands such as `cd`, `pwd`, `echo`, `kill` etc. In this project, you ***have to implement*** two built-in commands: `cd` and `pwd`.

For other commands (*e.g.*, `echo`, `kill`), your shell will process them and attempt to execute them like other regular commands using `fork()` and `execvp()`. If there exists a program available in the environment `$PATH`, your shell should successfully execute the command. For example, a `kill` (resp., `echo`) command should eventually be executed by running the `/bin/kill` (resp. `/bin/echo`) program. If no such program exists, `execvp()` will return an error, which your shell (to be precise, the child process) should handle and display the corresponding error message.

`exit`, `cd`, and `pwd` formats. The formats for `exit`, `cd`, and `pwd` are as follows: (here, ␣␣␣ denotes one or more `tab`/`space` characters and ␣ denotes one `space` character)

```
tinyshell> ␣␣␣exit␣␣␣
tinyshell> ␣␣␣pwd␣␣␣
tinyshell> ␣␣␣cd␣␣␣
tinyshell> ␣␣␣cd␣␣␣dir␣␣␣
```

When the user types `cd` (without 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 simply call `chdir()`. When a user changes the current working directory (*e.g.*, `cd somepath`), you simply call `chdir()` with the provided path.

You **do not** have to **support tilde** (∼). Although in a typical Unix shell you could go to a user's directory by typing `cd ∼username` or `cd ∼` or `cd ∼/some/path`, in this project you do not have to deal with tilde. Instead, your shell should treat it like a common character, *i.e.*, you should just pass the whole word (*e.g.*, `∼username`) to `chdir()`, and `chdir` will return an error.

When a user types `pwd`, you simply use `getcwd()` to find out the current working directory and display the result.

Once the built-in commands are implemented, your shell should behave like this:

```
$ ./tshell
tinyshell> cd /tmp
tinyshell> pwd
/tmp
tinyshell> cd
tinyshell> pwd
/home/<YOUR_USER_NAME>
tinyshell> exit
$ echo $?
0
$
```

Feature 1.4. (20 pts) Output redirection

Often a shell user prefers to send the output of his/her command/program to a file rather than to the screen (*i.e.*, `stdout`). The shell provides this nice feature with the > character. Formally this is named as redirection of standard output. Your shell should also implement this feature. For example, if a user types `ls -la /tmp > output`, nothing should be printed on the screen; instead, the output of the `ls` program should be rerouted to the `output` file in the current directory. If the `output` file already exists before running the above `ls` command, your shell should simply overwrite the file (after truncating it).

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`.

Feature 1.5. (10 pts) Background jobs

Sometimes, when using a shell, the user may want to run multiple jobs concurrently. In most shells, this is implemented by letting the user put a job in the **background**. This is done as follows:

`tinyshell> ls &`

By typing a trailing ampersand (`&`), the user tells the shell to launch the job, but not to wait for the job's completion. Thus, the user can start more than one job by repeatedly using the trailing ampersand. Your shell needs to implement this feature. Some example commands for background jobs:

```
tinyshell> ls &
tinyshell> ps &
tinyshell> find . -name *.c -print &
```

Feature 1.6. (10 pts) Foreground jobs

For a command that does not have a trailing ampersand (`&`) (*i.e.*, not a background job), your shell must wait for the command to finish its execution. Once the command finishes (*i.e.*, terminates/exits), only then your shell should accept a new user command.

> See the code given in Figure 5.4 from chapter 5 of the Textbook or `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`. It shows how to wait for a child process. `wait()` may seem to work, but it may not work as expected when one or more background jobs are currently running while your shell is waiting for a foreground job to finish. Study both `wait()` and `waitpid()` to identify which one to use. It is recommended to use `waitpid()` for this project.

Feature 1.7. (No pts) Not supported commands and options

Your shell does **not** need to support piped commands, that is, commands of the following form. Simply put, your tiny shell does not need to handle commands containing the pipe character (i.e., `|`).

```
ps -ef | grep root | wc -l
```

It also does not need to handle the following forms of I/O **redirections**: <, >>, `&2 >`, etc.

It does not need to support **wildcard/asterisk** (`*`) expansion. For instance, given `ls *.c` command, many UNIX shells will list all the files with `.c` extension in the current directory. Those shells consider the asterisk as a meta-character[1] and interpret it as zero or more of any character when searching for a pattern (`*.c`). For many commands like `find`, those shells apply the asterisk expansion first and then pass the result list to the command.

> Take a look at the `find` command shown in Feature 1.5. Note that you do not have to surround `*.c` with double-quotation (*i.e.*, `"*.c"`) as you would do when using one of the common UNIX shells (say, bash). The reason to surround wildcard patterns with double-quotation is to prevent the shell from expanding the wildcard.

For these non-supported commands/options, your shell should consider them (*e.g.*, `|`, >>, `*`, `;`, `#`) as **regular characters** and pass them as arguments to the command (identified by the first token/word). The command may return an error, which your shell needs to display.

Feature 1.8. (10 pts) Error checking

Your program should do appropriate error checking. In case of an error, your shell must print the following error message: "`An error has occurred\n`". For your convenience, the skeleton code `tinyshell.c` includes two macros `PRINT_ERROR` and `PRINT_ERROR_SYSCALL(x)`. The source code includes their definition. Instead of printing the error message using `printf()`, use one of the macros as required.

`PRINT_ERROR_SYSCALL(x)`: Use this macro when you are checking if a syscall function (*e.g.*, `fork`, `execvp`, `getcwd`, `getenv`, `open`, `chdir`, `wait`, `waitpid`) returns an error. This macro not only prints the error message from the syscall failure using `perror()`, but also prints "`An error has occurred\n`". Pass the syscall name when using this macro like this: "`PRINT_ERROR_SYSCALL("open");`".

[1] `https://www.lifewire.com/linux-metacharacters-using-them-2192773`

PRINT_ERROR: It just simply prints "`An error has occurred\n`". Use it for other errors in your code like this: "`PRINT_ERROR;`".

Exit status: When your shell terminates normally, it must exit with status 0, *e.g.*, using `exit(0)`. If your program needs to exit/terminate due to an error, it must exit with the status code 1, *e.g.*, using `exit(1)`.

> **How to build/run/test your program?**: See `task-1/README.md` file.

Appendix A: Some Useful Links

1. C - Error Handling – `https://www.tutorialspoint.com/cprogramming/c_error_handling.htm`
2. C Preprocessor – `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`