PA-2: C Programming on Linux and Shell

Total points: 100 pts

Instructor: Joseph Waclawski

Introduction

PA-2 Objectives

1. Providing insights into the creation, destruction, and management of processes, enhancing understanding of fundamental concepts in systems programming.

2. Offering exposure to essential shell functionalities, fostering practical knowledge in command-line interfaces and system interactions.

Logistics

1. This assignment must be done individually, not in groups.
2. For this programming assignment (PA-2), you have to download an archive called student-pa2.tar.gz from Blackboard to your Linux environment. Upon extraction, you will see a directory named student/, which contains 5 directories named task-1, task-2, and, so on - one for each task.
3. For each task, you have to complete a C program provided in the respective directory. For instance, for Task-1, you have to complete solution-1.c located in task-1 directory.
4. Take a look at the accompanied README.md in each task directory for additional instructions, such as how to build/run/test your program.
5. Unless stated otherwise, you should not need modify/edit/rename/move other existing files under the student/ directory. If you do, please comment the updates in the files modified.
6. For all the following tasks, you have to use the Linux environment you have dedicated for this course.
7. Some helpful links are listed in Appendix A. You may want to leverage them for solving PA-2.

Obtain the source code for this PA

Obtain student-pa2.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-pa2.tar.gz
```

Submission Instructions

1. Go to each 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>-pa2.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-pa2/
$ tar czf johndoe-pa2.tar.gz johndoe-pa2/
```

4. Use Blackboard to submit your compressed archive file (e.g., johndoe-pa2.tar.gz).

Grading

- You must not rename/move solution-*.c files under each 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 within a Linux environment, otherwise your solutions will not be considered for grading.
- Unless stated otherwise, you MUST NOT add any new files (e.g., additional C or header files) to any task-* /directory.
- 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.

Task 1. File operations (20 pts)

You need to write a C program (location: task-1/solution-1.c) that will take a single command line argument. The command line argument will be the name of a file. The file will contain a sequence of Unix/Linux commands for file manipulations: one command in each line. Your program needs to execute each of these commands. For this task, you must not use system(); instead, you have to use fork(), execvp(), wait()/waitpid() functions.

For each command, your program will spawn a process using fork() which will carry out the given file operation using exec() (or one of its variants). You can use execvp(). See this link ([1]) for exec() man-page and this link ([2]) for a pictorial description of its family.

Assumptions: You can safely assume that each line will have a maximum of 1024 characters including the newline character at the end of the line. The file contains only two types of commands: (1) cp <source-file-path> <destination-path>, and (2) mv <source-file-path> <destination-path>.

Error checking: Your program should do appropriate error checking. In case of an error, your program must print the following error message: "An error has occurred\n" and then exit with the error code 1, e.g., using exit(1).

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

Building a Simple Shell

For task 2 thru 7 of this PA, 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 task-2/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. 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.

Compilation instructions

The final executable must be named tshell and compiled as follows

```
$ gcc -o tshell -Werror -Wall tinyshell.c
# altenatively, use 'make'; see task-2/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
$
```

Task 2. Command Parsing and Execution (16 pts)

Currently, your tinyshell is unable to execute any user-inputted commands. This limitation means that when a user inputs a command, such as "ls -la /tmp", the shell does not respond with the expected action.

Your shell program should be able to execute the command parsed from the given input. For example, if the user types "ls -la /tmp", your shell should run the program ls with all the given arguments, and then ls will print its output on the screen (via stdout). Note that the shell does not "implement" ls and many other commands like this. All your shell needs to do is find the executable in the user's environment **$PATH** (e.g., for ls, the actual executable file is at /bin/ls) and create a new process to run the executable of the given command.

Details on input command parsing:

- **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 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 invalid. 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.

- **Input parsing:** The source code already implements the parsing of the input command line when provided by the user

For each command typed by a user, your shell should spawn a child process using `fork()`, and this child will carry out the given command using `execvp()`. The `fork()` portion is already implemented in the code. You must implement the rest by utilizing the populated `cmd string` structure called C.

Task 3. Foreground Jobs (10 pts)

Your tinyshell currently can't wait for a forked process to finish before showing the prompt tinyshell>. Hence, output from a long-running program might appear after a new prompt.

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

To complete this task, you must understand the `wait()` and `waitpid()` system calls, choose the appro- priate one, and implement it.

Task 4. Built-In Commands (16 pts)

Most Unix shells have built-in commands such as `cd, pwd, echo, kill` etc. Your tinyshell does not have these commands implemented. To make your tinyshell usable, you must allow your users to navigate through the different files and directories on your computer.

For this task, you must implement two built-in commands: cd and `pwd`. Their functionalities are identical to `cd` and `pwd` in a typical Unix/Linux shell.

The exact requirements for `cd` and `pwd` are detailed below:

- **cd**. 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 call `chdir()`. When a user changes the current working directory (e.g., cd somepath), you call `chdir()` with the provided path.
- **pwd**. When a user types `pwd`, you use `getcwd()` to find the current working directory and display the result.

An example output of a successful implementation of `cd` and `pwd`:

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

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

Task 5. Redirection (18 pts)

Often, a shell user prefers to send the output of their command/program to a file rather than to the terminal screen. The shell provides this nice feature with the '>' character. Formally, this is called the redirection of standard output. The current implementation of your shell cannot redirect standard output, a useful feature often desired by shell users.

This task requires implementing output redirection in the provided shell code. For instance, when a user inputs a command like "ls -la /tmp > output," the shell should redirect the output of the "ls" program to a file named "output" in the current directory without displaying anything on the screen.

To do this, you need to use file management system calls. If the specified output file already exists, the shell (child) should overwrite its content. Importantly, if the shell encounters issues opening the output file, you

must handle the error, display the relevant error message using appropriate macros like **PRINT_ERROR_SYSCALL**, and exit with the status code 1.

Hint: Use `open()` and `close()` system calls.

Task 6: Background Jobs (10 points)

The user may want to run multiple commands concurrently when using a shell. By convention, these multiple commands in this context are often referred to as background jobs. In most shells, this is implemented by letting the user put a job in the background. The shell program currently does not facilitate concurrently executing multiple commands, limiting its functionality.

To enhance the shell's capabilities, you must implement background jobs, enabling users to run more than one job concurrently. Specifically, the user should be able to initiate multiple background jobs by appending a trailing ampersand (&) to the command.

The shell program, acting as the parent process, should not wait for the completion of background jobs and should seamlessly move on to the next command. Some example commands for background jobs:

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

Task 7: Error checking (10 points)

The program exhibits gaps requiring code implementation and lacks proper error handling.

Your task is to fill these gaps with code incorporating comprehensive error checking. Specifically, ensure that the added code includes appropriate error-handling mechanisms.

To do this, you must utilize the provided macros **PRINT_ERROR** and **PRINT_ERROR_SYSCALL(x)** to print the error message: "An error has occurred \n". Use **PRINT_ERROR_SYSCALL(x)** when checking if

a syscall returns an error. Replace `printf()` with the appropriate macro as demonstrated in the provided skeleton code (tinyshell.c). Regarding exit status, when the shell (parent process) terminates normally, it should exit with status 0 using `exit(0)`. If the shell needs to exit due to an error, it must exit with the status code 1, e.g., using `exit(1)`.

Appendix A: Some Useful Links

1. **exec()**: The exec family of functions replaces the current running process with a new process. It is used to run a C program by using another C program. With the `execvp()` command, the created child process does not have to run the same program as the parent process. More information is listed on `https://man7.org/linux/man-pages/man3/exec.3.html` and `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf` .

2. **exec.md** : `https://gist.github.com/fffaraz/8a250f896a2297db06c4`

3. **fork()**: The fork system call is used for creating a new process in Linux and Unix systems, called the child process, which runs concurrently with the process that makes the `fork()` call (parent process). After creating a child process, both processes will execute the next instruction following the `fork()` system call. More information is listed on `https://man7.org/linux/man-pages/man2/fork.2.html` and `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`.

4. wait(): A call to `wait()` blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent continues its execution. More information is listed on `https://man7.org/linux/man-pages/man2/wait.2.html` and `http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf`.

5. **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`.

6. **chdir()** : `https://man7.org/linux/man-pages/man2/chdir.2.html`

7. **getcwd()**: `https://man7.org/linux/man-pages/man2/getcwd.2.html`

8. **Error handling** : Error Handling in C (tutorialspoint.com)

9. **C Preprocessor**: `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`