CSCE 451/851 Programming Assignment 1

Writing a Unix Shell

Parsing, Forking, and Logical Operators

Assigned: Jan 21, 2020
Due: Feb 6, 2019 23:59:59

IMPORTANT PRELIMINARIES!!!

- **DO NOT PUT THE FORK() STATEMENT INSIDE AN INFINITE LOOP!!**
  - This means no `while(1)`, or equivalent **ANYWHERE** in your code.
  - Do not use `while(should_run)` from the text
  - use a bounded `for` loop instead
    (e.g., `for (int should_run = 0; should_run < 25; should_run++)`)
- The program should exit on `exit` command.
- Finally, you **may not** use the `system()` function.

---

1 Overview of Project (PA1 and PA2)

This project consists of writing a C (or C++) program to serve as a shell interface that accepts user commands and executes each command in a separate process. A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file prog.c on the terminal using the Unix `cat` command.)

```
osh> cat prog.c
```

The above is an example of a simple command (i.e., it does not contain any operators). We will extend this program to execute more complex commands which contains one or more simple command connected together by an operator such as:

```
osh> cat < code.c
osh> cat < code.c > out
osh> ps | cat > out
```

Your shell will support the following operators (much like the standard `csh` or `bash` shells in Unix).

- > - Redirect stdout to a file
- < - Redirect stdin from a file
- >> - Append stdout to a file
- `&&` - Execute next command only on success.
- `||` - Execute next command only on failure.
- `;` - Execute next command regardless of success or failure.
- `|` - Pipe stdout of one command into stdin of next.

The functionality defined here for the above operators are basic, and may slightly deviate from the functionality defined by the standard shells available on Unix. But it is not the purpose of this project to write a fully functional Unix shell. Rather, it is to introduce and familiarize ourselves with various concepts useful in understanding operating systems and systems programming such file I/O, processes control, and inter-process communication. As such, we will only code parts of the shell that help introduce these topics.

PA1 This project is broken up into two programming assignments. In PA1 you will add support for:

- Input parsing
- `fork` and `exec` to create new processes
- All logical operators listed above except `|` (pipes)

PA2 In PA2, you will build upon PA1 and add support for:

- Interprocess communication via pipes

2 Submission

Use web handin to hand in your assignment. Submit a single zip file, `<UNL username>_<pa#>.zip` (e.g., `jdoe2_pa1.zip`) containing **only**:

- source files
- makefile
- README

Executing `make` in your project directory should produce an executable named `osh`. In the README file:

- Document instructions to compile, run, and test your program.
- Document any problems you had.
- Include a paragraph on what you learned in doing this programming assignment.

Remember to verify that your code compiles and runs on the CSE servers.

3 PA1: Evaluation and Points Distribution

The project zip contains 5 test scripts, alongside the corresponding expected output files. The table describes the test case, and the points awarded for each test case.

File	Answer Script	Test Case Description	Points
1.singleCommand.txt	ea1.txt	Simple command with arguments (no operators)	16
2.simpleRedir.txt	ea2.txt	Single command with single redirector (input or output <, >, >>)	16
3.moreRedir.txt	ea3.txt	Single command with multiple redirector (input and output <, >, >>)	16
4.logicalConditional.txt	ea4.txt	Two command connected by a single logical operator (&&, ||, ;)	16
5.malformed.txt	ea5.txt	Identify malformed commands	16
makefile		The program compiles successfully on command 'make'	10
README file		As described in submission guidelines above	10
Total			**100**

The test scripts can be used to test your program and are the test scripts we use for grading. A fully functional executable, `osh`, is included in the distribution zip file for this project. You can use it to see what your outputs should look like.

3.1 Critical "Gotchas" for grading

- **Please make sure your code complies on the CSE server**. If it doesn't, you'll get a 0
- Ensure that the output binary produced by makefile is `osh`.
- Points will be awarded only if your output matches the expected output **exactly**. Please conform to the error strings as specified in the expected output for Malformed commands.
- **Please make sure '-t' option is implemented**. This is described below. If you don't, you'll get a 0.

4 PA1: Detailed Discussion and Description

This assignment is based on programming assignment Project 1 (Part I) which is described on pg 157 of the text. We will not implement Part II of the project in the text. Instead we will extend the shell to support the file redirections, inter-process communication (pipes) and command combinations described above.

To help make the project more manageable we've broken PA1 into four phases.

4.1 Phase 1 - Parsing the Input String

The first step is to parse the input line and figure out what it means. Mainly,

- identify the name of the program
- supply the correct arguments to the program
- check if input or output needs to be redirected
- check if the output/input redirected from/to a file or another program
- check if there are any logical operators which control the execution of programs
- repeat the same step for each command in the input

In order to achieve this, we need to

- correctly parse the input line by separating each token
- identify separate commands and arguments for each command from the tokens
- building a data structure to represent the parsed data
- save additional information (about how it interacts with the next/previous command, if at all). This could be saving file handles, logical operators, etc.

With this data structure, we should be able traverse the list with ease, retrieve arguments, or skip to the next commands, etc.

We must also be able to identify malformed commands. Since we are able to traverse the command structure, we should be able to analyze which command/operator combinations make sense, and which don't. Mainly, we need to identify if

- we have a null command (e.g., `osh > cat || file`)
- if there is no file after a redirector symbol (e.g., `osh > echo >`)

Examples for malformed commands are shown in "5.malformed.txt" test file.

You are welcomed and encouraged to write your own parser. However, since the goal of our course isn't tokenizing, we've supplied you with two methods for doing it much easier.

4.1.1 Using parse.cpp This is a helper file that does the tedious steps of parsing the input line, and puts it in a doubly linked list. This is the easiest and quickest way to implement phase 1. Additional details of what the file does, and how to use it can be found in the "docs" directory that came in the zip file. It is not required to understand the implementation of parse.cpp, you only need to understand how to use the parser. The parse.cpp does not check for malformed commands. You will have to make sure your code checks for malformed commands.

4.1.2 Using parser from Yutaka Tsutano (former TA) Alternatively, a former TA, Yutaka Tsutano, has provided his own version of the parser available on his GitHub repo[1]. It is written in C++ and is a bit cleaner. He asks that you cite his work in your README file. If for some reason, his site is no longer available, let me know and we can get you a copy.

4.2 Phase 2 - Forking a child process and executing a command

After building the data structure, start by executing simple commands (and ignore the logical operators for the time being). Then you can extend the program to handle file redirection. This would involve

- extracting a simple command from the data structure (executable name and arguments)

[1] `https://github.com/ytsutano/osh-parser`

- creating a new process for the executable
- supplying correct arguments to the newly created process
- wait for the process to complete, collect exit code
- output the result to console (for now… later we'll modify this to handle a redirection operator)

We will use the following API:

- `fork()`
- `exec()`
- `wait()`

Please go through the man page for each of these. They are fundamental to process management in an operating system.

4.2.1 Step 1 Use the `fork()` and `exec()` commands with their respective arguments, ignoring file redirects and pipes for now (treat pipes as "`;`").

Create a child process to launch the executable. As covered in class you must use fork to exec a new program, else the shell will be replaced with the exec command and cease to exist. Shown below is pseudocode to accomplish this. Note the four rules when using fork.

1. Never place `fork()` in a `while(1)` loop.
2. Trap and exit if the fork fails.
3. Always have an `exit` in the child (NOT Shown in Fig 3.9 of the text).
4. Place a `wait()` in parent (unless we have multiple `exec`'s - (i.e., pipes)).

You have a choice of six exec functions, `execl()`, `execlp()`, `execle()`, `execv()`, `execv()`, `execvp()`, `execve()`. You are informed by two facts: 1) you do not know ahead of time how many arguments you will have, and 2) our shell does not maintain a hash table of executables and their locations. We would instead like the `exec` function to find the executable for us (which it *can* do). What all this means is that only one of the above exec function will work.

Pseudocode:

```
/* Rule 1: Dont forget this should not go inside a while(1)
loop anywhere in your program */

pid_t cpid = fork();
if (cpid < 0) { /* Rule 2: Exit if Fork failed */
    fprintf(stderr, "Fork Failed \n");
    exit(1);
}

else if (cpid == 0) {
    execlp(....);
    fprintf(stderr, "Exec Failed \n");
    exit(1) /*Rule 3: Always exit in child */
}

else {
    int status;
    wait(&status); /* Rule 4: Wait for child unless
                      we need to launch another exec */
    printf("Child caught bla bla bla\n");
}
```

4.2.2 Step 2 Next, modify your program to handle redirection operators. A program writes its output to `stdout`. `stdout` points to the console by default. In order to write to a file, you need to overwrite `stdout` to point to a file handle. In order to do this, modify your program as follows,

- when you read the command, check for the input/output redirector (e.g., <, or >)
- if so, get the file name
- open the file with the appropriate option (new file vs. append)
- after the new process is created, and before doing `exec()`, overwrite the `stdin/stdout`

In order to overwrite the `stdin/stdout`, use the `dup2()` system call. Here is a good description of `dup2()` and its use in a shell-like scenario. And here is a nice little article about `dup2()`.

Finally, you can add a loop to execute each command in the input line as a separate command (ignore the functionality of logical operators and pipe).

4.3 Phase 3 - Logical operators

Handling logical operators is fairly straightforward. The execution of the command is determined by the exit status of the previous command. In the previous section, we implemented the logic to collect the exit status which we can now use to determine whether or not the current command should be executed. Here are the steps:

- check if current and previous command are separated by a logical operator
- if it does, get the exit status of the previous command
- based on the exit status and logical operator, determine whether you need to execute current command or skip it
- if you need to skip it, set the exit status of the current command to be the same as the previous command

The above logic, is sufficient to handle chains of logical operators.

4.4 Phase 4

Finally, in order to evaluate your program, the program should implement a switch `-t`. When used, the program does not print the `osh>` prompt to the console. This helps us grade easier by not having to see the `osh>` prompt regularly. If this isn't implemented you will likely not get any points as the output will not match on the test scripts. The exact command we will run (for example, for test 1) will be:

```
osh -t < 1.singleCommand.txt
```