Lab 1: Developing a simple shell

Operating Systems Course
Chalmers and Gothenburg University

August 30, 2022

1 Introduction

In this lab, you will develop your own shell from scratch. A shell is a program that provides an interface to the operating system. It interprets the user commands and executes them on the system as intended by the user.

For example, a user that wants to view the list of files inside a directory types `ls` inside the shell. The shell processes/parses the string "ls" and starts a search for the binary ls on the system. If the program ls is installed on the system, the shell creates a new child process that executes the program ls. In the meanwhile, the parent process (the shell) waits for that child process to finish its execution. Upon completion of the execution, the shell comes back to the command prompt waiting for the next command from the input.

Apart from this basic command interpretation, a shell can be used to handle more advanced commands that specify input and output redirection, running commands in the background, combining multiple commands using pipes, etc. These advanced features are described in more detail in the following section.

2 Specifications

In order to pass the lab, your shell, called `lsh`, needs to implement the following functionality correctly:

1. It must allow users to execute simple commands such as `ls`, `date` or `who`. It should be able to find the location of commands (e.g., if a command is in /usr/bin, /usr/local/bin or any other folder), so it needs to be aware of the *path* where it should search for commands. The standard convention on UNIX systems is that the environment variable `PATH` contains the list of directories that should be searched.
2. It must be able to execute commands in the background so that many programs can execute at the same time. For example,

   ```
   $ sleep 30 &
   ```

   runs the command `sleep 30` in the background.
3. It must support the use of one or more pipes e.g.

   ```
   $ ls | grep out | wc -w
   ```

   outputs the number of files with filenames that contain the word "out" in the directory.
4. It must allow redirection of the standard input and output to file. For example,

   ```
   $ wc -l < /etc/passwd > accounts
   ```

   creates a new file "accounts" containing the number of accounts on the machine.
5. It must provide **`cd`** and **`exit`** as built-in functions.
6. Pressing Ctrl-C should terminate the execution of a program running on your shell, but not the execution of the shell itself.
7. Ctrl-C should not terminate any background jobs.

Important: Your shell has to be completely independent, and it is not allowed to delegate the command execution to other available shells. **It is not allowed to use a system call system() to invoke `sh`, `bash` or any other system shell.**

3 Implementation

You should download the lab code from canvas. The code already contains a skeleton implementation of the shell, in the file `lsh.c`, which you have to extend with the required functionality. All your implementation should be integrated into that file, since it is the only one which will be submitted for grading.

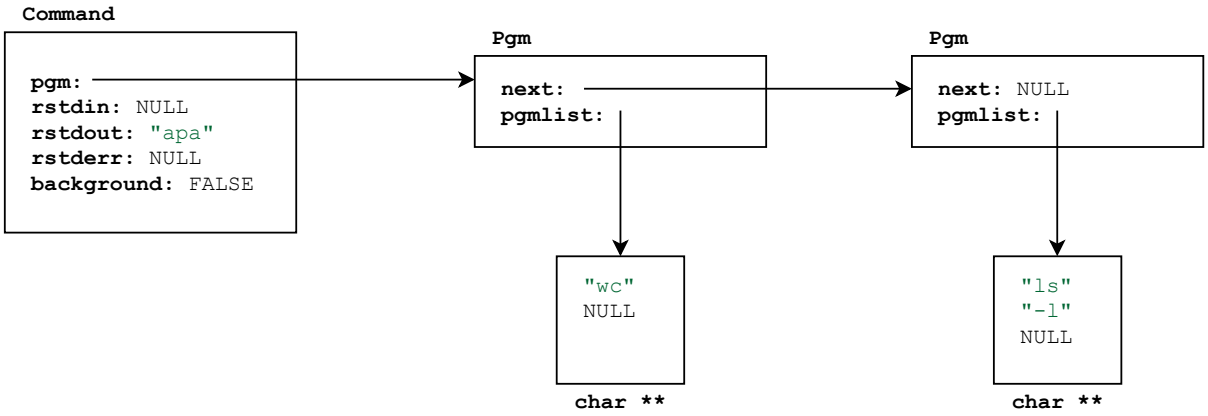

Figure 1: Example command structure.

The skeleton implementation handles the parsing of commands for you. It uses the GNU readline library, which means that commands can be entered using the same features as in `tcsh`. Further, a history function is provided, allowing the user to browse through previous commands using Ctrl-P and Ctrl-N. More specifically, the skeleton `lsh` contains the function:

```
int parse (char *line, Command *cmd)
```

which parses a command and stores it in the `cmd` variable. The parse function returns 1 if there are no errors and -1 otherwise.

For example, the following code

```
int r;
Command cmd;
r = parse( "ls -l | wc > apa", &cmd);
```

returns `r` equal to 1. After the call, `cmd` will have the structure shown in Figure 1. Note that commands are processed last-to-first (for a good reason!). You can take a look at the `DebugPrintCommand` function in `lsh.c` for an example of how to handle the `Command` structure.

3.1 Development and Debugging

It is possible to implement the functionality listed in the specifications for `lsh` in the order that they are listed. Implement one, test it, and then move on to the next one. *You should test all features of your shell on the Chalmers remote servers (distans.cdal.chalmers.se), since that is where the grading will take place.*

To be able to carry out the lab successfully, you will have to study the manual pages for various system calls. For some of them, e.g. **`exec()`**, several variants exist. You need to figure out which one suits you better; sometimes, there is more than one that is suitable. Some system calls that you will definitely need are: `fork()`, **`exec()`**, `wait()`, `stat()`, `signal()`, `pipe()`, `dup()`.

Clean code guidelines To increase your chances of passing the lab, make sure your code is clean and easy to understand. The following guidelines are a good place to start.

- ✓ Format your code! Most text editors can do this automatically for you.
- ✓ Use meaningful variable names.
- ✓ Write comments to explain difficult or strange parts of the code.
- ✓ Remove debugging statements before submission (e.g., printfs, PrintCommand, etc.)
- ✓ Verify that your code compiles and runs correctly on distans.cdal.chalmers.se

Observing processes with `top` A handy command while testing your code is `top`. By having a second terminal running `top` in the same machine as the one running your `lsh`, you can easily observe all the processes spawned by your `lsh` implementation, as well as their status (i.e., if they are running or if some have turned into zombies).

To get started with top on the chalmers remote servers, run `top` and type `u`, followed by your CID and `[enter]` to filter only processes started by you. Afterward, press `V` to enable the "forest view" where process trees are grouped together. After that, you can start your `lsh` and observe the processes it spawns below it in top's tree view. Pay special attention to the "S" column that shows the status of each process. Consult the manual of `top` for more details: `https://man7.org/linux/man-pages/man1/top.1.html`.

4 Code & Report Submission

To pass the lab, you need to implement all the requested specifications and verify your code with the self-test examples found below. You also need to write a report where you describe the design and behavior of your solution. Finally, you need to upload both the report and your code to Canvas. The following instructions describe the submission process in detail:

1. **Writing the report** For your report, begin by describing the implementation of your solution. More specifically, briefly analyze how you implemented each of the specifications described in § 2, what problems you encountered and how you dealt with them. Afterwards, go through the self-test examples found below. Include the output of each command in your report, as well as the answers to the questions. Make sure to justify your answer when your shell behaves in a strange or unexpected manner.
2. **Preparing the code** After you have verified that your code works correctly on distans.cdal.chalmers.se, run the `prepare-submission` script found in the lab folder. The script will check that your code compiles correctly and it will create an archive with only the necessary files for grading.
3. **Final submission** For the final submission, prepare an archive containing the archive of your code (prepared as per the instructions above) and the report file and upload it to canvas.

4.1 Self-Test Examples

The following examples will help you verify that your solution correctly implements the required specifications. Run all the commands in your shell and include the output of each command in your report. Make sure to also include answers to all the questions in your report.

4.1.1 Simple Commands

```
$ date
$ hello
```

The first command exists and the second one does not. Observe the system calls that are executed. If any of the programs fail, what is printed? Where? What happens to any child processes that your shell has created?

4.1.2 Commands with parameters

```
$ ls -al -p
```

4.1.3 Redirection with in and out files

```
$ ls -al > tmp.1
$ cat < tmp.1 > tmp.2
$ diff tmp.1 tmp.2
```

Is the output of `diff` what you expected?

4.1.4 Background Processes

```
$ sleep 60 &
$ sleep 60 &
$ sleep 60
```

Try to look at the parent process that is waiting for the child process using `top`, as described in the debugging section. Run the list of commands several times and use kill to see after which command it is possible to generate a prompt.

Try pressing Ctrl-C in the `lsh` after the last `sleep`. Does the foreground process stop? Do the background processes also stop? What is the expected behavior? Wait 60 seconds. Are there any zombie processes left?

4.1.5 Process Communication (Pipes)

Verify that your shell supports one or more pipes.

```
$ ls -al | wc -w
$ ls -al | wc
$ ls | grep lsh | sort -r
```

Does the prompt appear after the output of the above command?

```
$ ls | wc &
```

After running the above, when does the prompt reappear?

```
$ cat < tmp.1 | wc > tmp.3
$ cat tmp.1 | wc
$ cat tmp.3
```

Compare the output of the last two commands above. Are they the same? Why/why not?

```
$ abf | wc
$ ls | abf
$ grep apa | ls
```

What are the outputs? When does the prompt appear? Use Ctrl-D, if necessary, to let the grep finish and to let the shell process take over. Does the `grep` command terminate eventually (use `top` to check). Why/why not?

4.1.6 Built-in Commands

```
$ cd ..
$ cd lab1
$ cd tmp.tmp
```

Was there an error generated when executing the commands above?

```
$ cd ..
$ cd lab1 | abf
$ ls
```

Did the command `ls` work?

```
$ cd
```

Was there an error? Use pwd to see which the current working directory is.

```
$ grep exit < tmp.1
```

Did the shell quit, or did it consider **`exit`** as a text string to find in a file?

```
$ ..exit
```

And here? **(Use spaces instead of dots)**

```
$ grep exit | hej
```

Was there an error here? Does the prompt appear?

```
$ grep cd | wc
```

Did an output appear? Does it appear after pressing Ctrl-D?

```
$ exit
```

Are there any zombies after exiting `lsh`?

5 Appendix: Useful Unix Commands[1]

5.1 The man Utility

You will need to use the UNIX file API and the UNIX process API for this assignment. However, there are too many functions for us to enumerate and describe all of them. Therefore you must become familiar with the man utility, if you aren't already. Running the command man command will display information about that command (called a "man page"), and specifically, man `unix_func` will display the man page for the UNIX function `unix_func()`. So, when you are looking at the UNIX functions needed to implement this assignment, use man to access detailed information about them. The man program presents you with a simple page of text about the command or function you are interested in, and you can navigate the text using these commands:

1. Down arrow goes forward one line
2. Up arrow goes back one line
3. `f` or Spacebar goes forward a page
4. `b` goes back a page
5. `G` goes to the end of the man page
6. `g` goes to the start of the man page
7. `q` exits man

One problem with man is that there are often commands and functions with the same name; the UNIX command `open` and the UNIX file API function `open()` are an example of this. To resolve situations like this, man collects keywords into groups called "sections"; when man is run, the section to use can also be specified as an argument to man. For example, all shell commands are in section 1. (You can see this when you run man; for example, when you run `man ls` you will see the text `LS(1)` at the top of the man page.) Standard UNIX APIs are usually in section 2, and standard C APIs are usually in section 3.

So, if you run `man open`, you will see the documentation for the open command from section 1. However, if you run `man 2 open`, you will see the description of the `open()` API call, along with what header file to include when you use it, and so forth.

You can often even look at some of the libraries of functions by using the name of the header file. For example, `man string` (or `man 3 string`) will show you the functions available in `string.h,` and `man stdio` will show you the functions available in `stdio.h.`

5.2 Console I/O Functions

You can use `printf()` and `scanf()` (declared in `stdio.h`) for your input and output, although it is probably better to use `fgets()` to receive the command from the user. Do not use gets(), ever!!! You should always use `fgets(stdio, ...)` instead of `gets()` since it will allow you to specify the buffer length. Using `gets()` virtually guarantees that your program will contain buffer overflow exploits.

5.3 String Manipulation Functions

The C standard API includes many string manipulation functions for you to use in parsing commands. These functions are declared in the header file string.h. You can either use these functions, or you can analyze and process command strings directly.

`strchr()` Looks for a character in a string.

`strcmp()` Compares one string to another string.

`strcpy()` Copies a string into an existing buffer; does not perform allocation. Consider using strlcpy()() for safety.

[1]Source: `http://courses.cms.caltech.edu/cs124/pintos_2.html#SEC27`

`strdup()` Makes a copy of a string into a newly heap-allocated chunk of memory, which must later be `free()`d.

`strlen()` Returns the length of a string.

`strstr()` Looks for a substring in another string.

5.4 Process Management Functions

The `unistd.h` header file includes standard process management functions like forking a process and waiting for a process to terminate.

`getlogin()` Reports the username of the user that owns the process. This is useful for the command prompt.

`getcwd()` Reports the current working directory of a process. This is also useful for the command prompt.

`chdir()` Changes the current working directory of the process that calls it.

`fork()` Forks the calling process into a parent and a child process.

`wait()` Waits for a child process to terminate, and returns the status of the terminated process. Note that a process can only wait for its own children; it cannot wait e.g. for grandchildren or for other processes. This constrains how command-shells must start child processes for piped commands.

`execve()`, `execvp()` The `execve()` function loads and runs a new program into the current process. However, this function doesn't search the path for the program, so you always have to specify the absolute path to the program to be run.

However, there are a number of wrappers to the `execve()` function. One of these is `execlp()`, and it examines the path to find the program to run, if the command doesn't include an absolute path. Be careful to read the man page on `execvp()` so that you satisfy all requirements of the argument array. (Note that once you have prepared your argument array, your call will be something like `execvp(argv [0],` argv).)

5.5 Filesystem and Pipe Functions

`open()` Opens a file, possibly creating and/or truncating it when it is opened, depending on the mode argument. If you use `open()` to create a file, you can specify 0 for the file-creation flags.

`creat()` Creates a file (although why not use open() instead?).

`close()` Closes a file descriptor.

`dup()`, `dup2()` These functions allow a file descriptor to be duplicated. `dup2()` will be the useful function to you, since it allows you to specify the number of the new file descriptor to duplicate into. It is useful for both piping and redirection.

`pipe()` Creates a pipe, and then returns the two file descriptors that can be used to interact with the pipe. This function can be used to pipe the output of one process into the input of another process:

1. The parent process creates a new pipe using `pipe()`.
2. The parent process `fork()`s off the child process. Of course, this means that the parent and the child each have their own pair of read/write file-descriptors to the same pipe object.
3. The parent process closes the read-end of the pipe (since it will be outputting to the pipe), and the child process closes the write-end of the pipe (since it will be reading from the pipe).
4. The parent process uses `dup2()` to set the write-end of the pipe to be its standard output, and then closes the original write-end (to avoid leaking file descriptors).
5. Similarly, the child process uses `dup2()` to set the read-end of the pipe to be its standard input, and then closes the original read-end (to avoid leaking file descriptors)