

What the Shell?!

A Custom Linux Shell

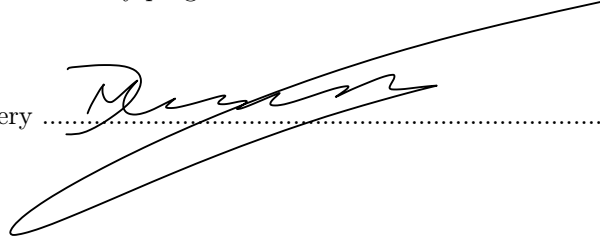
Daniel Madoery, Christian Hungerbuehler

University of Basel
Department of Mathematics and Computer Science
Operating Systems
12.06.2024

Declaration of Independent Authorship

We attest with our individual signatures that we have written this report independently and without outside help. We also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly. Additionally, we affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This report may be checked for plagiarism and use of AI-supported technology using the appropriate software. We understand that unethical conduct may lead to a grade of 1 or “fail” or expulsion from the study program.

Daniel Madoery



Christian Hungerbuehler



Introduction

In a more narrow definition, a *shell* refers to a command-line interface, CLI, as opposed to a graphical user interface, GUI. For experienced programmers, it is still the preferred way to work and interact with the operating system. However, given the prevalence of GUIs, most users have no experience working with a shell. In addition, while Linux offers a number of sophisticated shells, including the *Bourne-Again Shell*, *BASH*, and the *C shell*, this abstracts away both the inner workings of shells as well as the operating system. Implementing a custom Linux shell (re)create this bridge between user and operating system. Given the large and growing number of feature of operating systems like Linux, an important design aspect for any shell is that it can be easily extended with new commands.

Background

Inter-Process Communication, IPC

Given that each command runs as a separate child process of the shell as parent process and the necessity for the processes to cooperate by sharing data, *inter-process communication*, *IPC*, is an important feature of the shell. There are two models how processes can communicate with each other (Silberschatz et al. (2018)), *shared memory* and *message passing*.

With shared memory, two (or more) processes agree to share a region of memory. This approach has the advantage that it is faster as the operating system is only involved in creating the shared memory region. The shell uses share memory for the current working directory, which needs to be readable by many programs (e.g., `pwd` to print the current working directory) but also writable by a few (specifically by `cd`). Since multiple processes may access the same data, there is a *race condition*. As a result, the access to shared variables needs to be *synchronized*. In the shell, access to the current working directory is synchronized using a *mutex lock* (alternatively, a *binary semaphore* could have been used).

Message passing in UNIX systems is implemented using *pipes*, which are further distinguished into *ordinary pipes* and *named pipes*. Ordinary pipes can only be established between two processes and are uni-directional. The shell uses ordinary pipes to enable the user to pipe multiple commands such that the output of the previous command is directly used as input to the next command. Named pipes (also called FIFOs in UNIX systems) are more powerful than ordinary pipes as they allow bi-directional (half-duplex) communication between more than two processes.

Signal Handling

Signals are used to inform a process about a specific event and can be distinguished between *synchronous signals* that are generated and handled within the same process and *asynchronous signals* that are generated by one process and handled by another process (Silberschatz et al. (2018)). In the shell, signals (mainly errors) are generally handled synchronously as the goal was to decouple the shell (parent process) from the programs (child processes). The only connection between the two is the case when `exec()` fails, in which case the parent is responsible for terminating the child process.

Breadth-First Search, BFS

Breadth-First Search, *BFS* is an algorithm used for graph traversal search, exploring all its vertices. It is called breadth first because the algorithm always visits all vertices in a depth, before it moves along to vertices in the next depth level. The shell uses breadth-first search in the command `search`, which searches for a given file name.

Implementation, Configuration, and Setup

The Grammar

The shell is based on a simple *grammar*:

```
command [~flag] [input1] [input2]{|command [~flag] [input1] [input2]}
```

The general structure of a user-input consists of a command followed by an optional flag, which is indicated by a tilde, and up to two inputs all separated by space characters. As an example, `ls` displays all non-hidden files and folders in the current working directory, while `ls ~a` displays all files and folders including hidden ones. The shell also supports piping of two commands using the pipe-symbol `|` such that the output of the first command is used as the input of the second command. For example, `calc 3*4+5|to_file math` first calculates the operation $3 \cdot 4 + 5$ and then, instead of displaying the result on the screen, writes it to a file named `math`.

The Shell

The shell basically consists of an infinite loop. In each iteration, the user may provide an input which is then split into its components by the parser according to the grammar specified above and forwarded to the executor. If the input consists of a single command, the executor simply forks off a child-process and then attempts to execute a program with the same name as the command, providing it the flag and inputs, if available. If the input consists of more than one command, then each command is executed in a separate process which are sequentially connected using ordinary pipes. The implementation is based on Rodriguez-Rivera, G., and Ennen, J. (2015) as well as Rodriguez-Rivera, G., and Li, N. (2015).

The Programs

The shell itself is unaware of what exactly the commands and corresponding programs do making it possible to add new commands by following above grammar without having to recompile the shell. However, since some commands require a persistent change of the state of the shell, there needs to be a way for the programs to interact with it. Specifically, if the user wishes to change the current working directory of the shell, which is used to navigate through the directories, then the corresponding program `cd` cannot simply change its current working directory and it doesn't seem to be wise to have a child process change the working directory of its parent. As a solution, the current working directory of the shell is a variable kept in shared memory which programs may access and change.

Search

The command `search` goes through the whole directory starting from the `/home` directory and searches for a given `file_name`. To search efficiently, the search command uses breadth-first search to navigate through the forest of directories.

Below pseudo-code shows a simplified implementation of `search.c`:

```
//starts breadth-first search
while (!found) {
    struct node *n = dequeue();
    char buffer[BUFFER_SIZE];
    DIR* dir;
    dir = opendir(n -> dir);
    struct dirent *d;
    struct stat s;
    //appends all directories into the queue and adds all files to the files struct.
    while ((d = readdir(dir)) != NULL) {
        sprintf(buffer, "%s/%s", n -> dir, d -> d_name);
        stat(buffer, &s);
        if (S_ISDIR(s.st_mode) && (d -> d_name)[0] != '.') {
            struct node *new = make_node(buffer);
            enqueue(new);
        } else if (S_ISREG(s.st_mode)){
            files.name = d -> d_name;
        }
    }
    closedir(dir);
    //check if file is found already
    while (!found) {
        //compare file_name with found files
        //if found enter pathname into path_for_open_cd and stop all loops
    }
    destroy_node(n);
}
```

This code snippet shows how the BFS is implemented in the `search` command. In the first while loop, the least recently added node from the queue is dequeued and the path stored in this node opens with `opendir()` the next directory. The second while loop fills the queue with the newly found directories and stores all found files. In the last while loop, all in this directory found files get compared to the initially searched `file_name`. If the file is found, `search` returns the path where the file got found. The queue is used to guarantee that BFS is done properly, as we always append all directories found in the current directory, the code implemented never goes a depth deeper before it checked all directories at the same depth.

Results and Discussion

The shell supports the following commands and can easily be extended:

Command	Flag	Input 1	Input 2	Description
browse		[web_page]		Opens the <code>web_page</code> , if specified, in the user's default browser. Otherwise, <code>https://www.google.com</code> is opened.
calc		operation		Performs the calculation specified by <code>operation</code> (e.g., <code>calc 3*4+5</code> or, with spaces, <code>calc '3 * 4 + 5'</code>).
cd	~f	dir_name		Changes the current working directory. With ~f, an absolute path can be used instead of <code>dir_name</code> . With "." as input for <code>dir_name</code> the function <code>cd</code> goes back one directory.
cp		from_file	to_file	Copies the file <code>from_file</code> to <code>to_file</code> . Note that the <code>to_file</code> may include a relative or absolute path (beginning with '/').
ls	~a			Displays the files and folders in the current working directory. With ~a, additionally hidden files and directories are displayed.
md		dir_name		Creates a new directory named <code>dir_name</code> in the current working directory.
mf		file_name		Creates a new file named <code>file_name</code> in the current working directory.
pwd				Displays the current working directory.
search		file_name		Used to search a file in the directory, the search starts at <code>/home</code> and terminates when first instance is found.
to_file	~a	to_file		Used to redirect output from the console (i.e., <code>stdout</code>) to a file using piping. If ~a, then the file is opened in append-mode ("a"), otherwise in write mode ("w").
view		file_name		Displays the content of the file <code>file_name</code> .

One complication involved inputs including a space character which is the designated character to split arguments. Initially, for example, changing the current working directory to one that includes a space in its name was not possible. To resolve this problem, space character within a (sub)string enclosed in single quotation marks are ignored. To change the current working directory to a (sub)folder named `Operating Systems`, use `cd 'Operating Systems'`.

The shell also includes a basic recording feature to store the user-input in a text-file called `log.txt` which is stored in the directory of the shell. The recording feature can be toggled on and off by using the key-word `record`. By default, recording is turned off.

The shell can be closed using the key-word `exit`.

The key-word `help` displays all currently implemented commands.

On a technical note, since C doesn't automatically free memory that has been dynamically allocated (on the heap), we've been using `valgrind`¹ to avoid *memory leaks*.

¹<https://valgrind.org/>

Lessons Learned

This project gave us a good insight about what exactly a shell is and how it may be implemented. We also learned about intricacies of C that we are not used to from other programming languages like Python or Java, including memory management and the fact that a basic data structure like a queue cannot simply be imported from a standard library.

On the other hand, the project gave us a good opportunity to learn how to interact and to communicate in a team. We also learned that a good time schedule for tasks is not just important for the CPU but also for a project.

Conclusion

The customized shell that we implemented does fulfil all the requirements defined in the beginning of the project. It is capable of running different self implemented commands like `ls`, `cd`, `mf`, `cp`, `md`, or `pwd`. Furthermore, we achieved to implement a basic calculator and a search command to search for files, which are working as planned. The shell is capable of piping commands and signal (error) handling.

At the current state, the shell still depends on the default Linux shell. Therefore, as a bonus, we started to implement a GUI (`main.c`) for our shell to decoupling the customized shell from the default Linux shell. However, since fully decoupling the two turned out to be more time-consuming and required some fundamental changes to the shell, including redirecting the standard input, output, and error streams, we decided to stop the development of the GUI and, instead, finish other pending tasks.

In conclusion, as additional steps, the shell can easily be extended with more commands using the grammar explained. With more time, the shell could be integrated in an independent GUI.

References

- Kirkpatrick, M. S., "Computer Systems Fundamentals", <https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/index.html>
- Rodriguez-Rivera, G., and Ennen, J.: "Introduction to Systems Programming: a Hand-on Approach", <https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/>, 2015
- Rodriguez-Rivera, G., and Li, N.: "CS252: Systems Programming; Topic 8: Opening Files and Starting Processes", https://www.cs.purdue.edu/homes/ninghui/courses/252_Spring15/slides/CS252-Slides-2015-topic08.pptx, 2015
- Silberschatz, A., Galvin, P. B., and Gagne, G.: "Operating System Concepts", 10th Edition, Hoboken, NJ: Wiley, 2018