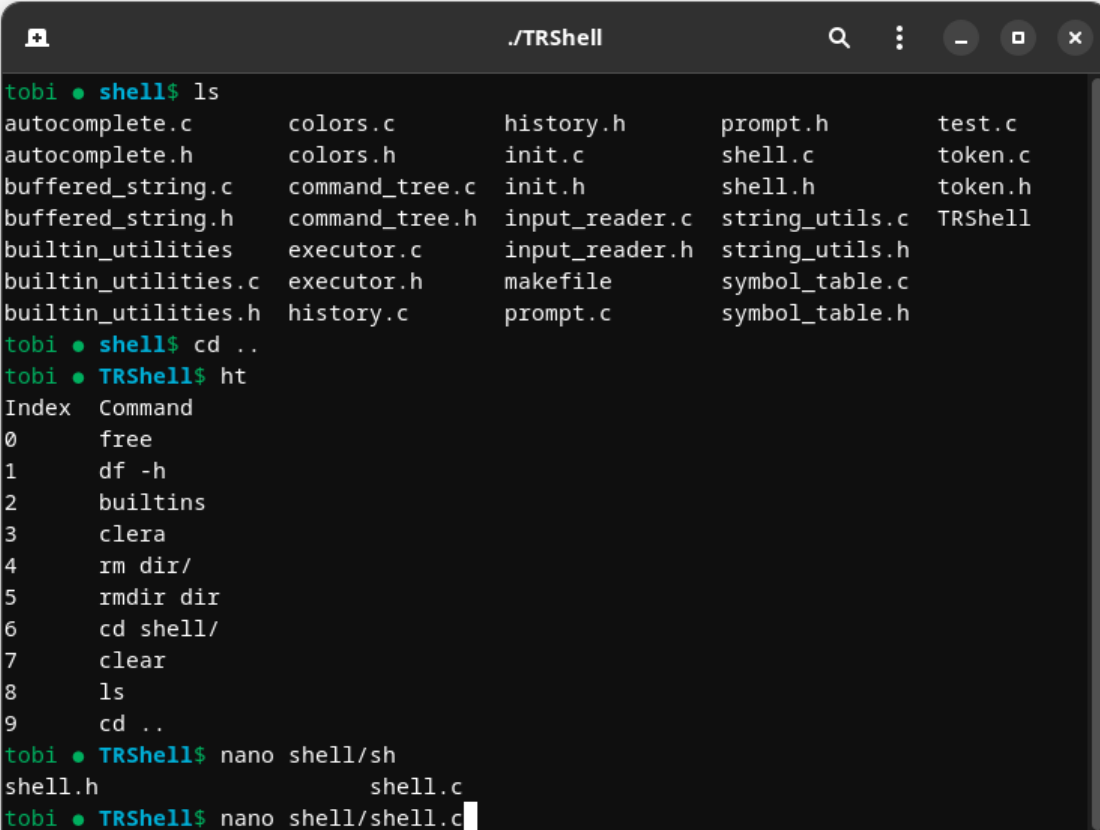


TRShell

Tobias Hafner, Ruben Hutter
Operating Systems FS 2022
University of Basel



```
tobi • shell$ ls
autocomplete.c      colors.c            history.h           prompt.h           test.c
autocomplete.h      colors.h            init.c             shell.c            token.c
buffered_string.c   command_tree.c     init.h             shell.h            token.h
buffered_string.h   command_tree.h     input_reader.c     string_utils.c     TRShell
builtin_utilities   executor.c          input_reader.h     string_utils.h
builtin_utilities.c executor.h           makefile            symbol_table.c
builtin_utilities.h history.c            prompt.c            symbol_table.h

tobi • shell$ cd ..
tobi • TRShell$ ht
Index  Command
0      free
1      df -h
2      builtins
3      clera
4      rm dir/
5      rmdir dir
6      cd shell/
7      clear
8      ls
9      cd ..

tobi • TRShell$ nano shell/sh
shell.h                                shell.c
tobi • TRShell$ nano shell/shell.c
```

Introduction

The goal of this project was to develop a simple linux shell capable of running programs, commands and offering some builtin utilities such as the command `cd` known in Linux. Further goals were to implement multitasking capabilities and the possibility to use a python based (terminal emulator) program to display multiple instances of our shell side by side. Nevertheless the focus of this project lies on the implementation of the shell with an eye on its ease of use as well as an efficient workflow. It is important to mention that not all features mentioned above were achieved by us in the time of this project for reasons elaborated later on in this report in the chapter "Difficulties".

This report consists of five chapters: "Technical background", "System overview", "Difficulties", "Lessons learned" and "Conclusion". In "Technical background" we will elaborate on the role of terminals, terminal emulators, shells, tty interfaces, ttys and ptty's and their differences. This knowledge is then used in the chapter "System overview" to explain the design of our shell and our reasoning behind our design choices. In the chapter "Difficulties" we'll elaborate why we weren't able to achieve all of the goals set by us at the beginning of the project. Further we'll go into the intricacies of the implementation of line buffering to show an example for the difficulties we faced and highlight our autocomplete system. The chapter "Lessons learned" contains what we've learned during this project and what we would do differently the next time. Lastly, we will share our conclusion of our project.

Technical background

To understand the technical requirements and find an efficient, modular and expandable design we had to understand what today's Linux users are looking at when opening the "terminal" by looking at the history of terminals. We want to share parts of what we learnt in the following sections.

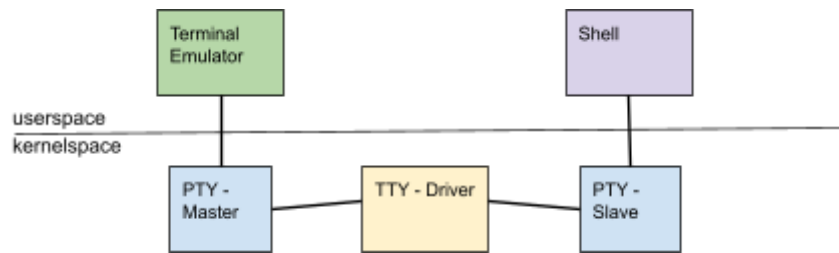
A very early form of a terminal was a very simple device called a teletype or tty for short. It consisted only of a keyboard for user input and a printer for output. This device was connected to a computer using two channels. One for input, one for output. The device would simply send all keys pressed down the first channel and immediately print everything it received from the second channel. There weren't any connections between the keyboard and the printer. Therefore the printer didn't know anything about the keys pressed, nor did the keyboard about what was being printed. The connection to the computer was constructed using a special hardware interface that was controlled by the so-called TTY driver, a software module in the kernel. The TTY driver was responsible for applying the so-called line discipline. This describes a set of rules for features like converting special characters (used for control sequences). Further it contains rules for echoing the received characters back to the TTY to be printed so the user can see their typed input.

Later on, with technical development the printer got replaced by monitors which offered advanced display possibilities such as colors, italics or the erasure of already written symbols anywhere on the screen. Special character(sequence)s are sent to the terminal to control those functions.

What we get when opening a terminal on today's computers is a program called a terminal emulator (pty = pseudo teletype). Terminal emulators are programs that emulate the behavior of a real terminal. They run in user space.

In contrast to terminals and terminal emulators a shell is a program in user space on an actual computer that uses system calls to offer a way to control the system and all its programs by prompting the user for input, handling that input and then returning the output from the underlying program or the operating system.

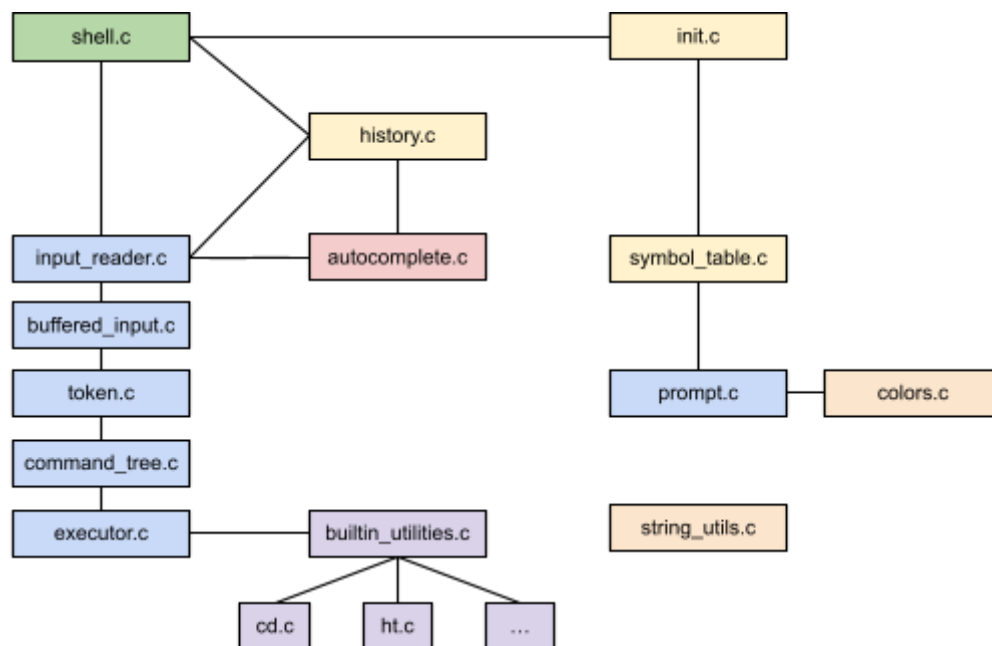
The implementation of the connection between terminal emulator and shell resembles the early day method with the two connections, one for each direction. A program such as a terminal emulator can request two character files from the kernel, one called the PTY master, the other called the PTY slave. The master side is accessed by the terminal emulator. The slave side is accessed by the program to be controlled. In our case that is the shell. The connection between the PTY master and PTY slave files is the TTY-driver mentioned before.



The takeaway from this short history section is that terminal(emulators)and shell are different programs connected by the tty driver communicating by nothing more than sending characters.

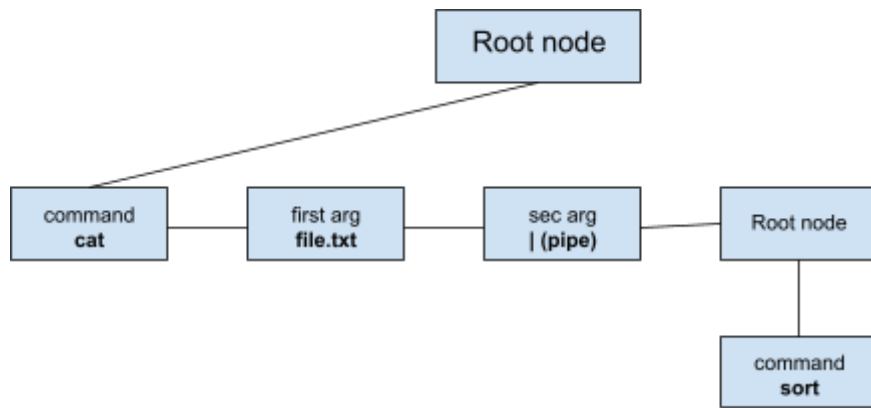
Source: (Apicella, Nicola. "Linux terminals, tty, pty and shell - DEV Community." *DEV Community* , 19 February 2020, <https://dev.to/napicella/linux-terminals-tty-pty-and-shell-192e>. Accessed 15 June 2022.)

System overview



Modules of trshell with links indicating a strong dependency on each other. Less strong dependencies are not shown for simplification.

The beating heart of any Linux shell is the Command Line Interpreter, or CLI. It has two main functions: to read and process commands entered by the user and then execute them. In our project the input_reader and the executor do just that. The input_reader reads everything the user types on the keyboard. It is first saved in a buffer, which contains all input that has not yet been confirmed for execution by the user. Upon the user pressing enter the input buffer is then transferred into a buffered_string structure (contained in buffered_input.c). It is then further broken down into tokens, fragments of the input (letters, numbers, or symbols), which all represent a unit of the input. After the tokenization, a tree is constructed out of the different tokens. Thereby the first fraction of the input, the so called command token is used as the root.



Graphical representation of an AST with a multi-command input

The other tokens are then attached to the root in order of input. A node in the tree, in addition to containing the contents of the token, has various other parameters like node type, variable type, and links to the child nodes. Similar tree structures are used also in other shells, and are called Abstract Syntax Tree, or AST. After the AST has been created, it is passed to the executor, where the root node is checked to see if it refers to a builtin utility in `builtin_utilities.c`, or one of the binaries installed on the system. This is done by comparing the entered command with the names of the binaries located at the directories stated in the `PATH` environment variable of the system. If a binary exists that matches the command, a subprocess executing the binary is created. After being executed, the AST gets freed. The original unprocessed input string is stored in a doubly linked list representing the history of commands entered. This list is used to offer the user the option to recall previously entered commands using the arrow keys or the `ht` command.

Another larger data structure and central component of our design is the `symbol_table`. The symbol table consists of a stack of hashmaps. These hashmaps are used to store the system's environment variables for quick access. Additionally shell options and properties are also stored in those hashmaps. The stacks are implemented to offer the future possibility for programs to store temporary values that are thrown off the stack after the command is executed. This feature becomes handy when processing more complex commands like these using piping. In the current state as is this feature is not used and there is only a single hashmap on the stack. This map is filled with all environment variables of the host at startup.

The most "characteristic" part of a shell, which we have not yet discussed, is the user interface, which is normally in interactive mode, where commands can be entered. The shell is in a loop, which is referred to as a Read-Eval-Print-Loop, or REPL. It prompts the user for input, evaluates it as discussed above, prints the results and starts over with the prompt. In fact, every time the user presses the enter key, a new prompt is printed and another command can be entered. The printed prompt depends on two main factors: the user running the shell and the current input state. Hues of blue and green are used for normal user privileges and hues of red and orange for the root. In addition the prompt symbol changes from "\$" to "#" when running as root. The input state describes if we are entering a multiline command or not. New lines in multiline state are marked with a leading ">".

Last but not least is our implementation offers evolved autocomplete. The completion algorithm considers what command and paths are already entered (if any) as well as installed binaries, builtin utilities, files and folders present dependent on already written paths/commands as well as the command history. Additionally a leading "?" can be used to search the history for a match. If a unique matchid found, it is completed automatically, otherwise a list with possible matches is shown, depending on the continuation of the user's input.

As this project involved a lot of string processing we implemented an own `string_processing` library called `string_utils.c`. This library contains methods for tasks like getting malloced copies of strings, the concatenation of an arbitrary amount of strings to one single one, advanced string splitting and matching or inserting a specific character before every occurrence of a specific character.

Another useful self implemented library is colors.c. It contains all code related to adding special character sequences to strings to achieve colored printouts on the terminal. This library is mainly used by prompt.c to color the prompt strings.

Difficulties

In our original project presentation we mentioned the development of a python program for our shell with split screen functionality and many shortcuts for easy navigation. Only after really understanding the differences between shells and terminal emulators did we realize that to reach this goal we would need to develop a whole terminal emulator compatible with all the control sequences for color, font styling, cursor movement and character editing to achieve our goals.

Some prototyping was done using self written python based GUI applications using the PyQt6 GUI framework. Those prototypes deal with the basics of communication between a shell and a terminal emulator using tty slave and master files and stdin, stdout and stderr pipes. The basic structure is as follows. The central module of those prototypes is what we call the wrapper. The wrapper is a python program that launches an instance of our shell c program as a new subprocess. Pipes are used to connect the new processes stdin, stdout and stderr to the main process running the wrapper. The main process uses a separate thread that constantly reads the subprocesses stdout and stderr and pushes the received output to the GUI. On the other hand the GUI can pass input to the wrapper. The wrapper then uses the stdin pipe of the subprocess to hand over the input to the shell.

We also developed another implementation of the wrapper that works in the same way as the one explained above but requests a pair of tty master and tty slave character files from the OS. These character files are then used to connect the subprocess to the main process.

The other important aspect of the prototyping consists of displaying the terminals screen on the GUI. A self-written PyQt widget inheriting from a TextEditWidget contained in PyQt6 is used to display and handle basic screen manipulation.

One of the running prototypes is the program trshell.py that uses pipes to connect the gui to a dummy_shell implementation that echoes back everything it receives. In this demo the user can enter text on the gui and see the echoed input being printed to the screen.

All the prototype code is contained in the folder trsh_py that was handed in with this report.

As a team of two people with exam preparation for many courses running alongside, we realized that writing a usable shell and a terminal emulator supporting enough advanced control sequences to allow the display of applications like nano or vim was unrealistic. That's why we focused on the main part of this project, namely the shell. For the demos, the terminal emulators on our everyday systems were used. Namely gnome terminal and alacritty.

Another thing we didn't expect were the intricacies of the communication between terminal, tty-driver and shell. We'll elaborate this by using the history function of our shell as an example. In our shell the up- and down arrows can be used at any time to browse through the commands entered before. Thereby the current command typed after the prompt is erased and replaced with the history entry. The problem arises with the fact that many modern terminal emulators are line buffered, meaning that you can type and edit your current command to your liking and then send it by pressing enter. As comfortable this feature may be it's not usable for our history as the user would have to press an up arrow and then enter to send it. To achieve the proper behavior we had to disable the line buffering in order to get individual keystrokes. This allows us to implement arbitrary shortcuts on the side of the shell. On the other hand this behavior has a huge disadvantage. As every keystroke is sent the user can not simply edit what was typed any longer as there is no buffer to edit. This means that the whole line buffer including edit functions like backspace, inserts or backspace in the middle of the input needs to be implemented as part of our shell.

Another difficulty was suppressing the text representation of the arrow keys (those are ^A, ^B, ^C, ^D for up, down, left, right) being printed to the terminal due to the echoing of the input by the tty-driver as part of the

line-disciple. We achieved this by disabling certain properties of the line disciple handled by the tty driver. Because characters are no longer echoed back to the terminal the echoing is also implemented as part of the shell.

Lessons learned

In this project we mainly learn from, and improve in, three fields of knowledge.

First we learnt a lesson in project planning. Our self set original goals as presented in the project presentation were nearly impossible to achieve. Further we learnt about prioritizing on the essence of a project in deciding to abandon the python terminal emulator branch of development. Apart from this we have proven to ourselves that our strategy to approach the problem of developing a shell paid out and enabled us to develop with ease.

The second field from which we learnt a lot was C programming. This shell resembles our up to this day biggest C project. We learnt a lot about the mechanisms and problem solving approaches used when developing in C. Additionally we learnt how to use the memory debugger valgrind to locate and fix memory issues such as memory leaks and segmentation faults.

Last we learnt about the differences between terminals, shells and the os related systems necessary for them to work like the tty driver.

Conclusion

In conclusion, we can call this project an interesting and successful expedition into the world of C development close to the operating system, and despite the headaches caused from time to time, either by uninitialized pointers, or by memory leaks that were almost impossible to find without valgrind, we undoubtedly had fun and are satisfied with our end result. We also cannot deny that we developed a certain appreciation for the simplicity and closeness to the hardware as offered by the C programming language.