Custom Alpine Shell

# Abstract

The Custom Alpine Shell is a shell written in C, which allows for easier access of the filesystem.

# Introduction

For the project, we have chosen to write a Custom Linux Shell. Our goal was to implement a shell that has many own features and thus stands out from other common shells. We have therefore decided to focus on file systems and their management. We wanted to optimise the use of file systems with our own features. Apart from that, our shell should still have the basic functionality of a conventional shell. This includes starting processes on the kernel and executing applications.  
  
We used C as programming language. We decided to do this because we already had experience with C. Furthermore, it is a standard Linux language.

Finally, we had to choose the Linux distribution where we wanted to implement our project. Our choice fell on Alpine Linux.  
Alpine Linux is a terminal-based Linux distribution. We have already worked with Alpine in another lecture. We found the management of files in Alpine very cumbersome and confusing. We wanted to make this easier with our shell.

# Implementation

## Command Parser

As Alpine does not use a Graphical User Interface, all user commands must be accessible by the command line. This means that the Command Line Input must be correctly interpreted and parsed.

After a line is entered, the whole input is taken by the reader. The reader serves as tool to read over the input and go back and forth on it. With the help of the reader, the input of the user is split into tokens. A token represents either the command or an argument of the command. Either a space or a tab is used as a separator.

The parser takes these tokens and structures them in a linked list. The head of the list is the command token. To distinguish between the individual commands, a type is assigned to each command. The head of the linked list is compared with the list of commands. If there is a match, the command type is set.

The commands are run in the executor. A switch case is used there to distinguish between the individual command types. Error handling also takes place in the executor. If the command entered could not be assigned to a type, it is assigned the type "error". The user is then informed that the command is not known. The user is also notified if too many or too few arguments are entered.

After executing the command, the allocated memory must be released. This is done in the parser. There, on each token of the linked list and on the command itself, free is called.

## Directory Management

For the directory management it was important to keep the ease of use in mind. Thus, the files and directories inside of the current working directory are saved inside Linked Lists. When using commands, instead of having to specify the filename or directory name, it is accessible by using the unique integer identifier of the corresponding Linked List.

These identifiers are displayed on the screen alongside their corresponding file/directory. Whenever a new directory is accessed, a new corresponding Linked List is generated, and the old one deleted. The user does not have to enter the whole file or directory name. Entering the integer identifier is saving time.

An other important aspect of the directory management was the colour coding. By using different colours for directories than for file should make our shell more clear.

## Command Implementation

### Show

When using show, all files and directories inside the current directory are displayed to the user with different colours for easier identification. Files and directories are sorted in alphabetic order.

### Showfiles

### When using showfiles, all files inside the current working directory are displayed to the user. The same list is used if the user has to select a file with a integer identifier.

### Showdirectories

When using showdirectories, all directories inside the current working directory are displayed to the user. The same list is used if the user has to select a directory with a integer identifier.

### Move

When the move command can be called without further arguments. The files inside the current directory are displayed to the user, alongside the identifiers of these files. The following user input is then evaluated, and the complete file path is saved internally. Then all the subdirectories are displayed with their identifiers, with which the user can navigate through the directories. Once the final directory is chosen, the new path is saved internally and the file is moved using these saved parameters.

However, the user can also directly enter a filename and a destination to move a file. Therefore, the first argument must be the filename with the right path from the current directory. The second argument must be the desired destination respectively to the current directory.

### Copy

Conceptually the copy command works the same as move. The difference is that instead of being able to directly move a file using a system call, a new file is created at the destination, into which the content of the first file is written into.

The command can be entered without arguments with integer identifiers or by specifying filename and destination.

### Rename

The rename command expects two arguments. The name of the file the user wants to rename and the new filename. The file is then directly renamed, with an error being displayed to the user if it is not possible.

### Go

By calling go, all the directories inside the current directory are displayed. When a number is typed, the current working directory is changed to the corresponding directory.

### Run

When run is called, any following arguments are entered into a system call, where it is executed.

### Delete

Delete can be called by directly specifying a filename.

If delete is called without an argument, the files inside the current working directory are displayed and the file can be chosen by specifying the identifier.

### Help

By calling help, all the commands that are possible are displayed to the user. Additionally a short explanation to every command is printed.

### Log

The log command displays all previously entered commands to the user. If an error occurred during a command execution, it is indicated in the log by a “FAILED” behind the command. The log is saved in a text file called “log.txt”. This file remains after terminating the shell. If the shell is started again, the old log file is deleted and a new one is created immediately. Moving or deleting the log file while running the shell leads to a loss of information.

### Exit

With exit, the user can terminate the shell. Exit can also be used to terminate a command where specifying a integer identifier is required. When the command is cancelled, no changes are made by the terminated command.

# Discussion

## Problems

There were several problems which had to be overcome for the project. The most common ones were cause by memory leaks as well as differences between the used operating systems, sometimes both.

### Windows-Alpine-Linux-Segmentation errors

An often-occurring problem was a segmentation error, caused by a different function implementation between Windows and Linux. The first part of the project was done with a Windows operating system. Windows was easier to use compared to Alpine Linux because it was the host OS and did provide a Graphical User Interface. However, some functions seemed to work in the same way on Windows and Alpine but differed slightly.

Finding the segmentation fault was difficult. The program didn’t crash at the actual spot where the variable is accessed but later when a printf-statement was made. To find the error we used the GDB-debugger. The segmentation error occurred because some functions did have different return values on Linux than on Windows.

### Memory leaks

The lack of already implemented dynamic data structures in C such as an Array List meant that it was needed to implement things manually, most prominently the Linked List. As C does not have a garbage collector, wrong implementations could lead to memory leaks. To avoid memory leaks, we used the debugger Valgrind. This provided us with information about our current memory management.

## Lessons Learned

There are some things we would do differently next time.

First, we would stick more strictly to the schedule. Debugging took more time than we thought. We also ended up a bit short on time due to other lectures.

We should have already worked with Linux at the beginning. At first it seemed more pleasant to work on the familiar Windows host, but the changeover to Linux cost us a lot of time.

Besides, we should have used a different Linux distribution. There is relatively little information about Alpine Linux on the internet. Installing packages with the Alpine package manager apk is also difficult, as much of the documentation is outdated or impossible to find.

## Division of Labour

At the beginning of our project, after research and design we worked separately on the directory management and parser. However, it quickly proved more efficient to work together more closely, as it was difficult to connect our different methods.

Research: Both

Design: Both

Directories: Matthew (+Elia)

Parser: Elia (+Matthew)

Commands: Both

Debugging: Both

Report: Both

# Conclusion

A fully functional Shell for Alpine was written in C, which allows for easier manipulation of the file system, including renaming, moving copying and deleting of files. Any functionality of the normal shell which are not implemented in the custom shell can be accessed with an integrated system call.