A0 Report

Daniel, Hanna, Jesper

September 2020

1 Introduction

This report will explain our approach to Assignment 0 (A0). The report will attempt to contain a clear summation and critical review of our work. The goal of A0 is to create a program that can recognize 3 different types of files and to test this program thoroughly.

To compile the code, execute

```
make file.c
```

to run the program, execute

```
./file <input file>
```

substituting `<input file>` with the file you want to run the program on. To run our tests, execute

```
bash test.sh
```

2 Implementation

The program file.c consists of 3 functions:

1. main(), where the program logic is executed,
2. check_type(), which checks a given file character by character and decides which type it is,
3. and print_error(), which is taken from the assignment description, and prints an error corresponding to the current value of errno.

The program starts by counting arguments. Given 0, it outputs a string to stderr as specified in the assignment description, given 1 it executes the program, saving the argument as 'path'. In any other case, the program terminates with exit code 1. Then the program attempts to open the file at the 'path' destination. If successful, the given file is run through check_type(), otherwise print_error() is called.

The program sorts a file as empty if the first byte of the file has the value EOF. The program sorts the file as an ASCII file if none of the bytes are outside of the intervals specified in the assignment description. Otherwise the program sorts the file as a data file. These types are implemented as enums.

To check if the file is an ASCII file, we implemented a while loop that uses fgetc() to store the bytes from the file in the local variable c. fgetc() checks a file byte per byte and is therefore suitable for this task. The aforementioned while loop checks if any of the bytes in the given file are not ASCII.

3 Design

A0 is an introduction to the Computer Systems course and the program has little complexity. If the program had more complexity, then choosing a proper software programming life-cycle would have been in order. In this case we simply followed the code and testing structure that was suggested in the assignment description.

Although the program for sorting files is simple we kept core coding principles in mind to make sure that the program was as extendable and maintainable as possible. Therefore we simply made a quick prototype and modified it accordingly.

Something we did that ensures extendability is implementating enums for printing the file type of the argument. This is readable because there is only one fprintf-statement for specifying the data type. If anyone decided to make to new type of file to distinguish between, they could simply add an enum and a corresponding string to the code along with a new branch in the function check_type().

To ensure maintainabilty and readability we made sure that the program had functions with only one responsibility. In our first prototype we had most of the code in the main fuction. Once we had done this, we realised that the functionality of the main function could be split into smaller tasks. This led to the function check_type() and the function print_error().

4 Testing

For testing, we have been given a shell-script test.sh, which compares the output of our file() with the output of the linux file(1) using diff(1).

The assignment demands that we make a dozen test cases for both ASCII and data types of files alongside 1 test case for empty files.

The test file checks if the file.c implementation outputs the right file-type to each of these test files. The user runs the tests by entering the src directory in the terminal and running "bash test.sh". The test file will output all test cases with the individual results.

For ASCII, we have tested rather randomly, with cases including:

- text, whitespace and escape characters,
- pure text,
- pure whitespace,
- numbers and arithmetic operators,
- and a slew of random words and sentences

For data we have been considerably less inventive, as the way the test.sh script tests has limited us in regards to which files are regarded as data files. Cases include a null-character in every case, as that is the only way we could think of to make the linux file(1) program return data. Variations include: starting null, end null, sequences of double, triple and quadruple nulls and nulls randomly sprinkled in between ASCII and UTF-8 symbols.

As for empty, a case can be made that only 1 test for empty files is adequate as only 1 way exists to actually have an empty file: the type where the first byte of the file is EOF. Honorable mentions could be missing files, but those are handled differently than empty ones and unreadable files, which are likewise handled differently.

5 Limitations and Potential Problems

The program file.c only discerns between empty, ASCII and data, where data is a blanket-designation which covers everything not included in the prior two categories. This posed a small issue during testing, as files which our program deemed data were judged as UTF-8 encoded by the linux file(1). These test cases have since been replaced by less interesting ones. The catch-all nature of the data file designation may also result in problems we have not foreseen at present.

6 Conclusion

The implementation of file.c is able to check whether a file is empty, is a ascii file or a data file. All parts of the assignment has been implemented and works according to the tests that are implemented in test.sh.