Compsys - A0

Christian R (srw812), Nicolai N (dwx635) og Andreas J (cmn945)

September 19, 2021

Introduction

In this report we will give our solutions to the problems presented to us in assignment A0. Our task was to implement our own version of the Unix tool File(1), to understand the function and test our solution.

Implementation

The core idea of solution is that by determining the first character, we can loop through all characters in a file and compare them to the first character and then updating the first character as we are looping. Before the first character is checked the file type is set to UNDEFINED. Our implementation consists of a function $file\_type()$ which determines the file type from input stream.

First of all, we try to determine if the file is empty by setting the pointer to the end-of-file and comparing this to 0. We use certain thresh-holds to determine if the first character is either ASCII, ISO-8859-1, or UTF-8 as that is the easiest way to differentiate between different file types. In regards to checking for UTF-8 characters it was a little more complicated as UTF-8 characters can consist of up to four bytes.

In order to check for multiple bytes we move the pointer to the next byte. If the function fails to determine the next byte as UTF-8, we move the pointer back. In order to check the bytes, we use the bitwise logical operator & and compare it to a binary number within the range of UTF-8 encoding since the bytes are required to follow a pattern as presented to us in the assignment text. Moreover, if the function fails to determine the file type based upon our limits we return the file type, data.

How to run the program

We expect exactly two arguments from the input stream. The first one is a call to our compiled version of file.c and then the file we want to check.
To run our program one has to open the terminal and navigate to the folder where file.c is located. Next, write the command "**make file**" which compiles the file.c to an executable file. Now, write the command "**./file** *file_name*" where *file_name* is the file to be checked. If "./**file** is ran without a file to determine the program will return EXIT_FAILURE. If exactly two arguments are given the program will print the file type and return EXIT_SUCCES.
To run the tests, *bash test.sh* can be run, which will print the test results to the terminal. the test file will compile file for you and thus it is not needed to write *make file* before testing.

Testing

We tested our file by extending the hand-out test.sh script, which compares the result of our file with the systems-installed file(1). This way we can look for any variations of test output. We tested for the following:

- Empty file
- ASCII
- ISO
- ISO that could have been utf8, but 1 byte was wrong
- ISO with ASCII chars
- UTF8 for each possible and then all possible amount of bytes
- UTF8 for each possible amount of bytes with an ISO char
- all possible encoding types with each UTF8 amount of bytes
- bytes between 126 and 160
- the byte 127(0x7F)
- all encodings plus a byte between 126 to 160 with a special case for 127 as well

One of the provided tests in the test.sh script fails as it contains a NULL byte. Because of this file(1) considers this a data file, however, based on our interpretation of the assignment, we have implemented it so that any and all bytes in the form 0xxxxxxx are considered to be valid UTF8 chars, where it seems file(1) does something different, however we decided to stick with our interpretation of the assignment.
The first tests for each encoding type test whether each encoding is detected without any real possibility of overlap of encodings.
For ISO we also tested what happened if it could look like a UTF8 char but a single of the bytes(not including the first byte which is integral to making it look like a UTF8 char) was ISO.
For ISO we also tested whether it worked if one char was ASCII

For our tests where the result for our implementation of file should return data, it is important to note that most of the tests(except for 1) fails. This is because the file(1) used to compare against can get results which are outside of the scope of this assignment and in this case generally returns "Non-ISO extended-ASCII text"(though there is a special case where result was different which we go in depth with below) which is not an encoding we were made to implement.

For testing data we first tested valid UTF8 chars with an ISO char ending the file, and with roughly the same idea we tested also putting in ASCII chars in between the UTF8 char and ISO char
To round it off we tested bytes between 126 and 160. With our implementation they should always return data. Here we noticed something peculiar specifically with 127. It was the only one where file(1) also returned data to us, so we tested 127 separately and it returned a peculiar encoding type: *"RDI Acoustic Doppler Current Profiler (ADCP)"*
With the last tests in mind we decided that we should also test all the encoding types with a random value 128-160 and with 127, and it returned as expected, with only 127 returning data.

Test Limitations

Our tests only cover cases where there is not any error. We have, by hand, tested cases where a wrong number of arguments is given, a file is unreadable or does not exist. However if there is an I/O error in the middle of the file, this is something non-trivial which we have not tested here.

Conclusion

We have implemented our version of file that can check and differentiate between files that are: empty, ASCII, ISO and UTF8 encoding. All parts of API specified in the assignment text are supported. We believe our implementation is of sufficient quality.

Theory

Boolean logic

We make a truth table to determine if the expression is correct or wrong. Truth table for expression $(A \hat{} B)\&A = (A\&\tilde{} B)$

A	B	(A ^ B)	(A ^ B) & A	(A & ~ B)
0	0	0	0	0
0	1	1	0	0
1	0	1	1	1
1	1	0	0	0

We can conclude that the expression is correct by looking at the truth table. When A is 1 and B is 0 both sides of the equation sign returns 1 and on all other occasions they return 0.

Bit-wise logical operators and representation

- x « 3 returns x multiplied by 8 since shifting left once is the same as multiplying by 2. Therefore, by left shifting 3 times is the same as multiplying by 8.
- x ^ 6 returns 0 (true) if x is 6. Bitwise operator XOR can be used to check if number is equals to 6 since the operator maps to 0 for like bits.
- ($\sim$x + 1) & 0x80 returns 0(true) if the x is lesser than or equal to 0 using the bitwise $\sim$ and & operators.
- !(x ^ y) returns 0(true) if x and y are different, and 1(false) if they are equal. The XOR operator returns 0 if two are numbers are the same, so we just negate that using !.

Floating point representation

The advantage of having denormalised/subnormal numbers in IEEE 754, is to smooth the gap between the smallest number and zero. It does so by using fractions and this reduces loss of precision when experiencing underflow.