A0 - CompSys

Nikolaj Krarup, ltf688,
Simon Wissendorff Seheim, fzt420
Emil Vedel Thage, prn150

September 2023

Introduction

For this assignment, we've been tasked with coding aspects of the unix command `file` in c. We've followed the requirements to the best of our abilities but not all logic questions have been answered.

Implementation

Our file implementation has one main functions, determineFile. determineFile reads a valid filestream and checks each bit with a while-loop. If fread() never reads anything, the loop won't be entered and the file will be recognized as empty. Each byte is checked to be either a ascii- or iso-character, and if the byte is neither, the loop breaks. Then the functions checks if the file is a UTF-8. The pointer in the file is reset to the beginning to prevent false positives and the file's bytes are again checked. UTF-8 has to be done in an entirely different loop since each UTF-8 character can be contained in multiple bytes and cannot be determined by just checking one byte at a time. UTF-8 characters can at most be 4 bytes long and follow a specific pattern to recognize the different byte lengths as specified in the requirements. As such, the distinct part of each byte sequence is checked for each character and if a match is found, it's checked if the next couple of bytes match the rest of the pattern.
Bools keep track of what type the file has and are changed according to the checkers inside determineFile. The bools are checked at the end one at a time, with ascii first and UTF-8 last. If no bools are true, it's a datafile. DetermineFile returns an Enum based on the bools. We use enum to easily add more types and make it easier to print the type into the terminal. The printFileType takes advantage of this and print the type to terminal. Based on the Enum printFileType gets as an argument, it will print a string from a constant array matching the Enum. If a type was successfully determined, file returns EXIT_SUCCESS
Some error handling has also been done according the requirements. We check for file readability and number of arguments given. A readability error is printed using the `errno` variable which it set when `fopen` fails. This error returns EXIT_SUCCESS, while the wrong number of arguments error returns EXIT_FAILURE.

Testing

Since our program is made following the standard `file(1)` utility it is quite easy to test. We have made our tests upon an existing `test.sh` file provided by our teacher. The test file calls our implementation of the file guesser and compares it to the Unix command line utility `file(1)` using the shell command `diff(1)`. Since `file(1)` may report a super-string of what our own `file` reports, the testing program uses the `sed` command to remove anything that `file(1)` might add to the string following the file type. Since we unfortunately didn't have the time to implement whether or not the file contained line terminators, this makes it possible to test files without `\n`.

We used shell command `printf` to create and input data into files. For testing an empty file we simply printed an empty string into a file. For each relevant text encoding we tested the bytes belonging to the set, with and without line terminators. We also tested different edge cases, like the first and last byte belonging to the set, and for UTF-8 we tested characters with 2, 3 and 4 bytes. For testing the `data` type, we simply did the opposite, testing bytes outside the sets of each text encoding. However we encountered a problem trying to test the bytes with decimal values 128-159 i.e., the bytes between ASCII and ISO. Our own implementation read the file as a `data` file, since it didn't belong to any of the text encoding sets. However the `file(1)` returned the string `"Non-ISO extended-ASCII text"`. For this reason we deleted these tests. The reason for these bytes not belonging to ASCII or ISO is because they might indicate a UTF file since the bytes begin with 10....

Testing the error part of our program proved a bit more tricky. We used the `chmod` command to create a secrete file. We also had to tinker a bit with the `sed` command to get the syntax of `file(1)` to match with the syntax requested by our assignment. Since the testing loop runs the program for every `.input` file inside the directory `test_files`, creating a test for a non existing file was not possible. So this we tested by hand, by typing non existing file paths as arguments for the program. The same goes for the error handling when calling the program without an argument. Another thing the shell program does not test is the actual exit code of the program. The `test.sh` only tests the string written to stdout, but not the actual return value of the main function. This we tested by hand using the `echo $?` command.

Limitations and Potential Problems

As stated in the previous section we did not have the time to create a full replica of the `file(1)` shell command. More specifically we have not implemented a way for the program to determine whether the file contains any line terminators. However we do believe this would be quite simple to implement, due to the open and expandable structure of our code.

Questions

Boolean logic

The expression is wrong. If A is false and B is true, the left side becomes false and the right side true, thus they don't equal each other for every case.

Bit-wise logical operators and representation

Multiply x by 8

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return -1;
    }
    int x;
    x = atoi(argv[1]);
    x = x << 3;
    return x;
}
```

Moving the bits of x 3 times to the left is equivalent to timing it by 3, since moving it 3 times $= x \cdot 2^3$

x is equal to 6

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return -1;
```

```
    }
    int x;
    int check;
    x = atoi(argv[1]);
    check = !(x ^ 6);
    return check;
}
```

By checking if everything is different and then negating it, we know if x is the same as 6.

Less than or equal to zero

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        return -1;
    }
    int x;
    int zero;
    int less;
    x = atoi(argv[1]);
    zero = !(x ^ 0);
    less = (x >> (sizeof(int) * 8 - 1)) & 1;
    return zero | less;
}
```

(x » (sizeof(int) * 8 - 1)) finds the significant bit, if it's 1, it was negative and & 1 is true, if the significant bit was 0, then it would be 0. Zero is checked the same as equal to six. If either zero or less is true, then x <= 0 and it returns 1 (true).

Floating point representation

The IEEE 754 floating point format usually have numbers represented in a normalized form. This means the number before the decimal point is not zero. In IEEE 754 this follows the form $v = (-1)^s \cdot M \cdot 2^E$ where $E = EXP - Bias$. This however, proves troublesome when working with numbers very close to 0, which is where denormalised numbers has an advantage. Here $E = 1 - Bias$, which allows for very small numbers to be handled more acuratly. If we did not have denormalized numbers, all very small floating point numbers would just become zero, which would result in a big loss of precision.

Conclusion

In conclusion, we've implemented our own version of `file` that can check for, empty-, ascii-, iso-8859-1, and UTF-8 files. We've followed the API requirements, we've tested the program to the best of our ability, and we've all learned a lot about C.