Assignment A0

Daniel Blyme Grundtvig, Emil Weel Sørensen

10. September 2017

1 Compilation

We've reused the MakeFile provided by the course, so it is used to compile both the *file(1)* program and the *test.sh* shell script.
They are compiled by navigating to the file locations in your terminal, typically using the 'cd' function. They are then compiled by calling 'make file' and 'make test' respectively.
The file program can now be run by calling '$./file < FILE\_PATH >$'. And the automated tests can be run by calling '$./test$'.

Tests

We used the provided test.sh script to test our *file(1)* implementation, We manually created some baseline testfiles, and then auto-generated a dozen of both ascii-text files and "random-text" files using basic bash functions.
For the ascii set, we had aleady been provided with two files. One that just contained text and one which contained the same text but with a newline char at the end.
These files were not very large, only a sentence long, so we created the file "$large\_ascii.input$" which is a generated file, which contains the number PI with the first 4000 digits.
This can be changed, by editing the "Scale" value in the bash script.
We then created 12 files only containing characters from the ASCII subset defined in the exercise, by using the shell functions '$/dev/urandom$' and '$base64$' [2] to get random data, but only from base64. These were simply used in a loop which created the file and increased the size of the file exponentially.
These tests were all marked as succesful except for the first file, which because of it's small size is considered a "very short file (no magic)". This is not a type we acknowledge in our program. Similarly, we created an exact copy of the "$large\_ascii.input$" but appended a character, _372, from outside the provided definition of ASCII characters and called it "$large\_non\_ascii.input$".
Again we used a loop to easily create multiple files of increasingly large size, but this time we used the *shred* [1] function to get entirely random data.

When the file size is small enough for this type, sometimes the original file program will assign the files with a different type than data, which is due to the entirely random nature of the bytes. This is why we have made these files start off being larger than their ASCII counterparts, as those weren't relevant.
We then created empty files using the functions in four different ways. The three of which were actually empty with a size of 0 bytes and marked as such by the program. But the file ”$empty_echo.input$”, created using the 'echo' function was marked as an ASCII text file by our program but a ”very short file (no magic)”. This happened, because even though the file looks empty upon inspection it actually has the size of 1 byte and contains a new line character which makes it an ASCII text in accordance with the assignment.
Then we tested the output when file tried to determine a non-readable file, which produced the correct output in accordance with the assignment. This however differs from the original implementation, which is why the test is marked as ”failed”.
The last test was done on a non-existing file, which again provided the correct output but was different from the original implementation.

Implementation

Our implementation is very simple, we check first how many arguments are present, and continues if there are exactly one argument. This is intentional, so the user does not think, that we will be checking multiple files at a time. This could be implemented, but it is outside the scope of this assignment.
We then use the fseek function to find the end of the file, and the ftell to determine the size of the file. This size is then used to allocate a char array which will hold all the data from the file.
After each file manipulation function, we check for IO erros, and exits with apropriate error messages if found. We then read the file in the char array.
Each char is cast to int and checked to be within our provided subset of the ASCII table, and if any char is found to be not of this subset, the program will recognize the file as a data file.
We are currently using an int (bool) to switch between ASCII and data but if we are to extend the program, we will be using a enumerator as it would be able to hold an arbitrary number of file types.

Ambiguities

We are not entirely sure what is meant by ambiguities in this context - however an ambiguity could be that the provided test script tests the output of our file program against the originals output, but in many cases they will differ because the original program recognizes more file types than ours.

This could be intentional however, so that these "failed tests" can be made succesful in an extended implementations.

References

[1] StackOverFlow Forums. Is there a command to write random garbage bytes into a file? URL: `https://stackoverflow.com/questions/3598622/is-there-a-command-to-write-random-garbage-bytes-into-a-file`.

[2] SuperUser Forums. How to create a random .txt(human readable text like ascii) file in linux. URL: `https://superuser.com/questions/692175/how-to-create-a-random-txthuman-readable-text-like-ascii-file-in-linux`.