Computer Systemer - Assignment 0

Nikolai vmf457

Haseeb pmj428

Sara spz377

Team 5

September 21, 2019

1 Introduction

This report provides an overview of the solution to assignment A1, which is a further development of the solution to assignment A0. The purpose of the assignment is to create a program that mimics the behavior of file (1). file (1) is a Unix tool, as given a file, checks the file type and prints it on the screen. If the file does not exist or is called without any file name, it returns an error code. The report provides a review of the implementation of this program, which has been written in C and can be run with a makefile. The report also provides a review of the tests for the program. To run the code:

$ make file

gcc -std=c11 -Wall -Werror -Wextra -pedantic -g3 -o file file.c

$ ./file <file path>

To run the tests:

$ bash test.sh

2 Implementation

Our program file is primarily built by one while loop, which uses fgetc to run the entire file and check every byte in the file. This while loop runs within one for loop, which passes each parameter. This makes it possible to give files more files than just one. The program does not use enum with saved file types, but instead calls them directly in the program itself. This is done after specific requirements with character values are changed on the created status value, which is later used to print the correct file type. The various file types the program supports so far are ASCII text, ISO-8859-1 text, data, empty, UT8 Unicode text, Little-endian UTF-16 Unicode text and Big-endian UTF-16 Unicode text. The program also initially uses another for loop, which checks which parameter (file names) has the longest name. This is used for the printing of files, which is about making sure that they are printed at the right space. The program also uses break to stop while loop if a specific file type is identified. An example of the above is the following:

```
if (num == 255) {
    if (fgetc(file) == 254) {
        status = 4;
        break;
    }
}
```

This is an excerpt of the code that shows how break is used and how status is changed. In this case, it is the implementation of the recognition of a file with the file extension Little-endian UTF-16 Unicode text.

3 Theory

3.1

The macros are defined as functions that is, they are modified to be able to take input arguments. The limits of each Macro for each UTF8 type are defined in decimals. The macros return true if the input argument falls within the limits. The limits are calculated by taking the lower bound and upper bound for the first byte in each of the 1 byte, 2 byte, 3 byte and 4 byte encodings. The bytes are converted to decimals and then hardcoded into the macros. A cleaner solution would have been to use bit masking so the unneeded bits could be filtered out. The tests for the macros return 1 since each test with its respective input argument falls within the limits of the if statement. Returning 1 is equivalent to returning true.

3.2.1 Program I

```
long p2 (arg1, arg2, arg3) {
    arg3 = arg1;
    arg3 = arg3 >> 63;
    long var = arg1;
    var = var ^ arg3;
    var = var - arg3;
    return var;
}
```

Referring to the given x86prime source code: The first two lines refer to the push instruction to the stack while the last two lines before the return command refer to the popping instruction on the command. Since this has no direct translation in C code, we can safely ignore these lines.

We see a `movq` instruction with the destination being register `%rdx` which holds the third argument , thus we assign the third argument the value of the first argument (register `%rdi`).

Next, We see an arithmetic right shift on the third argument with the result being the stored in the third argument and the shift being given by the immediate $63. Finally, there is another `movq` instruction and finally an exclusive or instruction whcih are directly translated to C as shown below. The function returns teh value in the `%rax` register which is the variable `var` in our case.

3.2.1 Program II

```
long p1(long arg1, long arg2, long j) {

    j = 0;
```

```
    long i = 0;
    long value = *(&arg2 + i);

    while (value != 0)
    {
        if (value != arg1)
        {
            *(&arg2 + j) = value;
            j++;
        }
        i++;
    }

    value = *(&arg2 + j);
    *(&arg2 + j) = 0;
    i = i - j;

    return i;
}
```

Referring to the given x86prime source code:

The first two lines refer to the push instruction to the stack while the last two lines before the return command refer to the popping instruction on the command. Since this has no direct translation in C code, we can safely ignore these lines.

First we see the `xorq` commands which are bitwise exclusive-or expressions, these simply evaluate to 0 since the operands are both the same. Thus this operation is the same as assigning a value of 0 to both variables. We know that register `rdx` holds the third argument so we assign our third argument j a value of 0. THe other simply holds a local variable which we initialize as a long with value 0 (which suffix q indicates).

Next, we see a `leaq` and a `movq` operation. The `leaq` operation loads the effective address of the source and saves that in the destination register. The `movq` operation reads the value at the address of the same register that the `leaq` operation has written to. In essence, the first operation translates to loading the address of the $i'th$ index, `(8 * (i) + arg2)`, while the second then computes the value there `M(8 * (i)+arg2)`. Since an array is simply a place in memory, in code this translates to reading the value of the $i'th$ index from an array. In the C code above, the pointer is retrieved and i is added to it, before dereferencing. This is essentially equivalent to saying `long value = A[i]`

In the same code block, we see the same instructions but this time the order of the operands on the `movq`

instruction are reversed, indicating that we are writing to the array index. Also, it can be seen that the index variable in the `leaq` instruction now lies in the `%rdx` register, which is the variable `j` in our C code.

Finally, since the `%rax` register holds teh return value, we return the variable `i`.

4 Testing

The tests are performed in such a way that the test.sh file compares the result between file(1) and file.
For these tests, it is interesting to examine the following cases:

- An empty file
- Edge cases for ASCII and ISO-8859-1
- Files that is ASCII text except the last byte
- Files that is ASCII text except the first byte
- Files that is ISO-8859-1 text except the last byte
- Files that is ISO-8859-1 text except the first byte
- UTF8, UTF-16BE and UTF-16LE tests
- How file handles incorrect file paths

For this, it has also been examined whether the program works for the tests that were made for A0. These tests, on the other hand, have not had much relevance, since none of them examined specific cases or edge coverage.

5 Conclusion

A version of a file(1) that mimics the program has now been reviewed. The program complies with the various API requirements set out in the assignment description and can know the different file types as well. There is also a review of the theory behind the program, which also shows rewriting of x86prime code to C code.