CSE30 - Lab #3

Introduction

The focus of this assignment is file input/output and bit manipulation in C and ARM Assembly. The assignment also emphasizes a sound understanding of the ARM procedure call standard. Click on this link to create your repo and get the starter code for the assignment: Lab 03 starter code

If clicking on the link does not work, copy and paste the url on your browser:

`https://classroom.github.com/group-assignment-invitations/ef565b99b5593e8488300f29c3c2ba5e`

If you are working with a partner, both of you will have to click on the above link. However, only one of the partners has to create a joint repo. The other partner only needs to join the repo following the instructions provided by the above link. If you are working in a pair create your repo following the naming convention:

<partner1firstname-lastinitial>-<partner2firstname-lastinitial>

For example if the partners were Joe Smith and Lily Monroe, the naming convention should be JoeS-LillyM. Note that the partner whose first name appears earlier lexicographically should appear first in this naming convention. Choosing LillyM-JoeS as the repo name would be incorrect in this case. Do NOT include the keywords partner1 and partner2 in your repo name. If you are working individually follow the naming convention:

<firstname-lastinitial>

Once you have create a new repo, or joined an existing repo, you will have access to the starter code and you can start working on the assignment. Remember the general guidelines emphasized in class: compile and run your code often. Develop your code using test driven development (TDD). Submit to gradescope often. Following these guidelines allows you to be in control of the outcome of this assignment!

This is an assignment in two parts - For the C part submission, you must turn in your submission for Section 1 (Warm up) as well as the functions from Section 2.2 and 2.3.
The ARM part constitutes the remaining sections and has a later due date. Please check the assignment calendar on the class website for all due dates.

This assignment is subject to the style guidelines from PA1.

The C part of the assignment can be completed on ieng6 or on the Rasperry Pi hardware/emulation environment. The ARM part should be completed and tested on the Raspberry Pi, or on a Raspberry Pi emulation environment.

1 Warm Up: File I/O and Bit Manipulation

So far, you have worked on programs that read and write data into memory and interact with standard output. However, often times inputs to your program may come from other sources, such as over the network or from a file! These sources of inputs/outputs are known as streams. The C programming language implements its own interface for accessing data streams(such as files) using FILE pointers and a set of stream I/O functions. We will now look at how to use these functions to get input to your program from files and also write the output of your program into files. We will do this through a simple warm up exercise that counts the frequency of occurrence of characters in a file. The following will be a useful resource as you try to complete the warm up exercises.

`http://www.tutorialspoint.com/cprogramming/c_file_io.htm`

Please also read Chapter 22, Sections 22.1 -22.5 of the King book.

1.1 Counting Characters

In this section you will write a program that reads the contents of a file and computes the number of times each character appeared in the file, we will call this the frequency count of the character. By default, each char is represented using 1 byte following the standard ASCII encoding scheme. For instance, when you write `char c = 'a';`, the value stored in 'c' is actually the ASCII value for 'c' which is the integer 97.

But how is the number 97 stored in a file? Files reside in memory (specifically on the hard disk of your computer). As always, the real representation of all data is binary. So, in a file, the number 97 is stored as a one byte binary number, which is '0110001', same as how it would be stored in memory.

What are the steps involved in writing your program?

1. Open an input file in binary read mode using the I/O stream function `fopen()`. Note that, for this part, you are only implementing `char_freqs`, which already takes a file pointer as an input. You may assume that the file pointer passed in is either NULL or an file pointer to a file that is already open.

2. Read one byte at a time from the input file until the end of file is reached. For every byte read increment the frequency count of that character. See `fgetc()`.

3. Store the frequency count of each char into an array of 256 ints, where each index of the vector corresponds to the count of the ASCII character corresponding to that index number. For example, index 97 should correspond to the count for the ASCII character 'c' in your input file.

4. Close the input file using `fclose()`. For the sake of consistency, we do not require that you close the file in `char_freqs`. In a full program, we expect you to close any files that you open, in the same way we expect you to free any memory that has been dynamically allocated (more on full programs in section 1.2).

Write your code in the file `warmup.c` provided to you. In `warmup.c`, implement the function:

```
unsigned int* char_freqs(FILE *fp);
```

This function takes as input a pointer to a file. If the pointer is not NULL, the function should return the count of each possible symbol in the file as an integer array of size 256. If the input file pointer is NULL, your function should return a NULL to indicate failure.

1.2 Bit manipulation

In this section you will read a sequence of integers provided to you in an inputfile and write them back to an output file after reversing the endianness. This means that if the integers were stored in Little Endian format in the input file, they will follow a Big Endian format when written to the output file, and vice versa. As before, your implementation is divided into two files: `reverse_endian.c` and `warmup.c`.

Implement the function `int reorder_bytes(int)` in the file `warmup.c`. The declaration of the function is provided to you in `warmup.h`. This function reverses the byte ordering of the input value. For example if the input value is `0xAABBCCDD` (4 bytes of hex), the output value should be `0xDDCCBBAA`. If the contents of the file are the string "ABCDEFGH", then the output file should be the string "DCBAHGFE".

A useful tool when doing this conversion is a **bit mask**. A bit mask is a carefully bit pattern that allows extracting a desired set of bits from a number using the appropriate bitwise operation. For example if I want the

least significant byte of the value (`0xAABBCCDD`, I would perform a bitwise 'AND' between the number and the bit mask 0xFF).

Implement the main function in the file `reverse_endian.c`. The main function should perform the following operations:

1. Open the inputfile in binary read mode and the output file in binary write mode using the I/O stream function `fopen()`. Note that the file names are passed as arguments to your main function.

2. Read from the input file one integer (4 bytes) at a time and appropriately call the function `reorder_bytes()` to obtain an integer with the reverse byte ordering. Write this integer into the output file. See `fread` and `fwrite`. Repeat the process until there are fewer than 4 bytes to be read in the input file. If there are fewer than 4 bytes, don't write those bytes to the output

3. Close the input file using `fclose()`

Compile your code into an executable called `reverseendian`, which should be run as follows:

```
./reverseendian inputfile outputfile
```

You are free to handle the case where fewer than two arguments are passed, in any way. However, your program should not segfault under any circumstance.

If you have extra time, we highly recommend that you write the ARM version of the function `reorder_bytes()`.

Testing Tools

A useful Linux tool for comparing two files is `cmp` and `diff`(see the manual page for `cmp` by typing `'man cmp'` in an open terminal, and for `diff` by typing `man diff` into an open terminal).

Another useful tool for performing sanity checks is `hexdump` (see the manual page for hexdump by typing `'man hexdump'` in an open terminal).

And now, to the main part of the assignment! Remember practice makes perfect. So, keep going!

2 Hiding and retrieving secret messages

So far, we have worked with 8-bit binary representations of character symbols, also known as the extended ASCII encoding scheme. In 8 bits we can represent 256 unique symbols. However, if we were working with a reduced set of symbols, say just the letters of the alphabet and punctuation marks, we need fewer bits to encode those symbols. In this section you will work with one such reduced set that comprises of only 64 symbols. Therefore, a 6-bit encoding scheme is sufficient.

Encoder

The encoder should read characters from a file containing a secret message and convert each character into its 6-bit binary representation in the provided encoding scheme. The message using the new encoding scheme should be stored to file composed of ASCII 0's and 1's. Each bit of the encoded message should then be implanted into one of the bits of a randomly generated character byte. In the test files provided to you, we have implanted bits into the third least significant bit. This two layer encoding scheme essentially hides your secret message from plain sight. The final encoded message should be stored in a file.

Decoder

Given the file containing the encoded message, you will write code to reverse the above process by applying a two layer decoding scheme. First, you have to extract each bit of the secret message from each byte of the encoded message. You should know where each bit resides, **it's the bit at the same index where the bits were implanted in the encoded byte**. For test files, the index in which we implanted the bit is indicated in the encoded file name (Ex: for code_trollface_0.txt, the bit is implanted in index 0). Then examine the extracted bits, six at a time in order to map them to the corresponding character symbol. The decoded message should also be stored in a file. If your implementation is correct the decoded secret message and the original secret message will be identical.

Implementation

Now that you understand the end to end behavior of your program, its time to think about how you would design your code. This involves breaking down the encoder and decoder parts into sub-tasks or functions. A first step when designing your code is to come up with the exact function signatures, describing the inputs and outputs and also what each function is expected to do. To help you along, we have given you the function signatures. However, in later upper division courses like CSE 100, you have to do this crucial step on your own. So, take a moment to go through the function signatures that are described below and relate them to the encode/decode process. We will rigorously unit test your implementation of these functions, so should you! Be sure to not change any of the header files or function signatures, otherwise your code will not work with our test code.

You are required to implement both the encoder and decoder in C, and re-implement only the decoder in ARM Assembly.

2.1 C Implementation

The specific files that relate to this section are provided in the skeleton code. These are:

- `common.h` and `common.c` : These files consist of code and global data that will be shared by both the encoder and decoder. **You should not modify these files or implement any new code here**, rather use the functions given to you in your encoder and decoder. `common.c` contains a `char*` array called `MAPPING` that contains the 64 symbols that could be encoded. The 6 bit encoding of these symbols corresponds to the symbol's index in that array.

- `encoder.h`: This file consists of all the functions that must be implemented as part of the encoder. **Do not modify it.**

- `encoder.c`: Your code for the encoder goes here. You have to implement all the functions declared in `encoder.h`. In addition, you may declare and implement any helper functions for the encoder here. Do not implement your main function here.

- `decoder.h`: This file consists of all the functions that must be implemented as part of the decoder. **Do not modify it.**

- `decoder.c`: Your code for the decoder goes here. You have to implement all the functions declared in `decoder.h`. In addition, you may declare and implement any helper functions for the decoder here. Do not implement your main function here.

- `tester.c`: Your code for testing end to end functionality goes here. Note that you may create your own test files to unit test the encoder and the decoder. **You do not need to turn in this file or any additional test files that you create.**

- Makefile: We have given you a sample Makefile. You may compile your code for encoder/decoder by typing `make` into the command line. Modify it as necessary. **You do not need to turn this file in.**

- `refencode_ieng6 / refdecode_ieng6 / refencode_rpi / refdecode_rpi`: These are "reference" implementations that satisfy the functional requirement of this assignment. You can use them to verify your results. Usage:

  ```
  ./refencode_[ieng6/rpi] [input filepath] [ASCII binary filepath]
  [output filepath] [index to implant]
  ```

  ```
  ./refdecode_[ieng6/rpi] [input filepath] [ASCII binary filepath]
  [output filepath] [index to extract]
  ```

  The executables should produce a file that contains the ASCII "binary" encoding/decoding whose name is specified by the ASCII binary filepath, and a file that contains the fully encoded/decoded file whose name is specified by the output filepath and whose implanted/extracted index is specified by the last argument. The index is a number between 0 and 7 with 0 referring to the least significant bit and 7 referring to the most significant bit.

  Note: *_rpi were compiled on an Raspberry Pi environment (The ieng6 RPI environment), and *_ieng6 were compiled on ieng6. Because of the different architectures in which these executables were compiled, you will not be able to run these

executables in the wrong environments. Moreover, in different environments (e.g. in Mac OSX), you may generate different randomized chars and different encodings than is provided in the sample test files. We recommend you compile on ieng6 or on the rpi if comparing fully encoded files.

- `test_files`: This directory contains encodings for 5 files: allchars.txt, astley.txt, harambe.txt, onedoesnot.txt, and trollface.txt. The fully encoded files are encoded with an index indicated by file name (Ex: code_trollface_0.txt is encoded by implanting into index 0, the least significant bit). You may test your code using these files as inputs, and compare these files to their encodings (bin_*.txt, code_*.txt) and their decodings (bin_de_*.txt, and the original file should match the decoded files). For more exhaustive testing, we recommend you generate your own test files as well.

Note that all other code in that directory is related to the warm up exercises.

2.2 Encoder

Below is the list of functions that you will implement as part of the encoder. More detailed descriptions of these functions are also available in `encoder.h`

- `char* encodeChar(char c)` - This function takes a char c as input and returns a (6 character) array that represents the 6 digit code for that character. The 6 digit binary code is simply the index of the char in the MAPPING array (represented in binary). You should use the REVERSE_MAPPING array in this function to get the binary code for the character, initialized by some call to createReverseMapping(). For example, if the code for the input character c is 3, then the returned char array (let's call it arr), should be "000011", with arr[0]='0' and arr[5]='1'. Notice '0' and '1' are characters and not numbers.

- `char implantBit(char c, int bit, int index)` - This function takes a char c and int bit and int index as input. It then sets the bit at the input index of c to be the input bit and returns the result. Note that the least significant bit is index 0, while the most significant bit is index 7.

- `void textToBinary(FILE *in, FILE *out)` - This function takes a FILE handle 'in' as input and reads the file, character by character. It then encodes each char into a 6 character "binary" char array (by calling encodeChar). The resulting character arrays should be written to the output file handle 'out'.

- `void binaryToCode(FILE *in, FILE *out, int index)` - This function takes a FILE handle 'in' as input (corresponding to an ASCII "binary" encoded file) and reads the file one char at a time. Each char read will be an ASCII '0' or ASCII '1', and either 0 or 1 will be implanted into randomized chars generated by rand()%256. The appropriate bit is then implanted into the bit at the input index of the randomized chars (by calling implantBit). Write the result into the output file handle out. **DO NOT EDIT OR REMOVE THE LINE** `srand(1);`

- `void encodeFile(char* input, char* bin, char* output, int index)` - This function reads in a file from the specified input path and outputs a binary encoding to the specified bin path and a fully encoded version of the input file to the specified output path. This function should simply

open the necessary files, call the above helper functions in the correct sequence, and then close the necessary files.

2.3 Decoder

Below is the list of functions that you will implement as part of the encoder. More detailed descriptions of these functions is also available in `decoder.h`

- `int extractBit(char c, int index)` - This function takes a char c and int index as input. It then extracts the bit at the input index from the char c and returns it. The least significant bit is index 0.
- `char decodeChar(char *b)` - This function takes a 6 character array b as input and returns the corresponding char from MAPPING that is indexed by the binary ASCII string b. If b is the char array "000011", with b[0]='0' and b[7]='1', then the decoded char should be MAPPING[3].
- `void codeToBinary(FILE *in, FILE *out, int index)` - This function takes a FILE handle in as input (corresponding to an encoded file) and reads the file, char by char. The bit at the input index of each char is extracted (by calling extractBit). For each character, if the extracted bit is 0, output ASCII '0' to the output file. If the extracted bit is 1, output ASCII '1' to the output file.
- `void binaryToText(FILE *in, FILE *out)` - This function takes a FILE handle in as input (corresponding to an ASCII "binary" decoded file) and reads the file, 6 chars at a time. Each 6 chars (all ASCII 0's and 1's) should be read into a char array and decoded into its corresponding character symbol (by calling decodeChar). The resulting chars would be output to the FILE handle pointed to by out.
- `void decodeFile(char* input, char* bin, char* output, int index)` - This function reads in a file from the specified input path and outputs a binary decoding to specified bin path and a fully decoded version to specified output path. This should simply open the necessary files, call the above helper functions in the correct sequence, and close the necessary files.

2.4 ARM Implementation

In addition to implementing all the functions listed in the previous section in C, you will additionally implement just the functions of the decoder in ARM assembly. The skeleton code for these functions has been provided in the following files:

- `extractBit.s`
- `decodeChar.s`
- `codeToBinary.s`
- `binaryToText.s`
- `decodeFile.s`

Implement the functions `extractBit, decodeChar, codeToBinary, binaryToText` and `decodeFile` in the respective .s file. The description and signatures of these functions are the same as those specified in the previous sections. When testing your code, make sure you don't attempt to compile two copies of the same functions into the same executable. Instead the assembly functions should be used instead of your functions in decoder.c.

2.5 An example outline

Suppose our input file, called in.txt, is just the character "j". We'll walk through the process of this encoding and decoding scheme in this outline to give you an idea of what the behavior of your code should be. The idea is that we need a two layer encoding scheme. For simplicity, we assume the randomly generated chars for this example are the ASCII lowercase letters "c" to "h", generated in order. Note that "c" is ASCII 97, so its binary representation is 01100001. We'll also assume that the index you are implanting into is 1 (the second least significant bit). **NOTE: The randomly generated chars will vary by architecture, but should produce the same random chars using the same seed and architecture. ALSO, your code must be able to handle implanting in any index bit of a char (0 through 7).**

1. First we want to encode input file in.txt into binary using our `textToBinary` and `encodeChar` functions, and output the result to some output file, let's say binen.txt. `textToBinary` should read each character of an input file, character by character, and convert each character to its 6 digit binary encoding using `encodeChar`. The encoding is given by the given MAPPING array in common.h, where the corresponding char's index corresponds to its binary encoding. For example, "j" is index 9 in MAPPING, so its encoding, to be placed into the output file, is "001001" (the unsigned 6 bit binary representation of 9). The output file should be composed of ASCII 0's and 1's (when opened using some word processor, it should appear as a sequence 1's and 0's). The result is illustrated below.

   input file: $\boxed{j}$ $\longrightarrow$ ASCII "binary" output file: $\boxed{001001}$

2. Next we want to encode ASCII binary file binen.txt even further, by using our `binaryToCode` and `implantBit` functions, and output the result to some output file, which we'll call out.txt. `binaryToCode` should read each character of an ASCII binary input file, character by character, and implant the corresponding bit at the index of a randomized char. In `binaryToCode`, there should be a line srand(1); which generates a "random" seed. You will generate "random" numbers using `char c = (char)rand()%256;`. The random numbers you generate should be different for each char you read in from the binary input. We'll assume

that you generate the chars "c", "d", "e", "f", "g", "and "h" in that order, and implant the read in "bits" into the second least significant bit of the generated chars. The index of the least significant bit is assumed to be 0. **NOTE: We do NOT use MAPPING or the 6-bit representations of the randomly generated chars here. We instead just implant bits**.

(a) The first random char generated is "c", ASCII char 99. It therefore has a binary representation 01100011. The first character we read from the input file is "0", so we need to implant 0 into the 2nd LSB of "c". What this means is that the 2nd LSB of "c" should be replaced with the corresponding bit of whatever we read in from binen.txt. In this case, we read in "0", so we need to make the 2nd LSB of "c" 0. The result would then be 01100001 = "a". The following steps will skip this explanation and give the original random char, the bit to implant, and the resulting char.

(b)

original char: $\boxed{d\ (01100100)}$

(bit to implant at index 1: $\boxed{0}$) $\longrightarrow$ resulting char: $\boxed{d\ (01100100)}$

(c)

original char: $\boxed{e\ (01100101)}$

(bit to implant at index 1: $\boxed{1}$) $\longrightarrow$ resulting char: $\boxed{g\ (01100111)}$

(d)

original char: $\boxed{f\ (01100110)}$

(bit to implant at index 1: $\boxed{0}$) $\longrightarrow$ resulting char: $\boxed{d\ (01100100)}$

(e)

original char: $\boxed{g\ (01100111)}$

(bit to implant at index 1: $\boxed{0}$) $\longrightarrow$ resulting char: $\boxed{e\ (01100101)}$

(f)

original char: $\boxed{h\ (01101000)}$

(bit to implant at index 1: $\boxed{1}$) $\longrightarrow$ resulting char: $\boxed{j\ (01101010)}$

The result is illustrated below.

ASCII "binary" input file: $\boxed{001001}$ $\longrightarrow$ fully encoded output file: $\boxed{adgdej}$

3. Next, we want to be able to decode similarly formatted encoded files. Since out.txt is fully encoded, we'll use that as an example. We will first translate it to our first decoding layer using out `codeToBinary` and `extractBit` functions, and output the result to some output file, let's say binde.txt. `codeToBinary` should read each character of an input file, char by char, and extract the bit of that char from the specified index using `extractBit`. In `extractBit`, if the bit at the specified index is 0, we return 0, and if the bit at the specified index is 1, we return 1. We then output an ASCII "0" to the output file if the bit extracted was 0, and ASCII "1" if the bit extracted was 1. We assume we know that the code is implanted at index 1 for out example, and extract the bits accordingly. The resulting output file should be exactly the same as binen.txt, since those are exactly the bits implanted previously. The result is shown below:

   input file: $\boxed{adgdej}$ $\longrightarrow$ ASCII "binary" output file: $\boxed{001001}$

4. Finally, we fully decode the file, using our `binaryToText` and `decodeChar` functions. We output the result to some output file, let's say decoded.txt. `binaryToText` should read the input file, 6 chars at a time, and convert char array to its fully decoded char using `decodeChar`. The encoding is given by the given MAPPING array in common.h, where the corresponding char's index corresponds to its binary encoding. In our example, the first and only 6 chars we read are "001001", so the corresponding index is 9, and the char decoded is at MAPPING[9]. The result is shown below:

   ASCII "binary" input file: $\boxed{001001}$ $\longrightarrow$ fully decoded output file: $\boxed{j}$

3 Submission

Commit and push all your code on github. Make sure you do this often. Make sure you also turn submissions in to gradescope.

3.1 C Part submission

Turn in your submission for Section 1 (warm up) as well as the ten functions specified in Section 2.2 and 2.3. Make sure that your submission compiles and has no memory leaks. You should turn in the following files.

- `warmup.c`
- `reverse_endian.c`
- `decoder.c`
- `encoder.c`

Gradescope will have two assignments for the C part. One is PA3_Warmup, and the other is PA3_C. Turn in `warmup.c` and `reverse_endian.c` into PA3_warmup. Turn in `decoder.c` and `encoder.c` into PA3_C. Additionally, you have the option to turn in a zip file containing the required C files in the respective assignments.

3.2 ARM Part submission

For the ARM part, you must turn in all of the functions as specified in section 2.4. Make sure your code compiles on the Pi or on a pi emulation environment. **Code that doesn't compile will receive 0 credit.** The files to be turned in for the final submission are:

- `extractBit.s`
- `decodeChar.s`
- `codeToBinary.s`
- `binaryToText.s`
- `decodeFile.s`

Turn in all `*.s` solution files into PA3_ARM. Additionally, you have the option to turn in a zip file containing the required `.s` files in to PA3_ARM

3.3 General Submission Guidelines and Tips

You may submit your homework as many times as you'd like, but only the final submission will be recorded. We will use our own **Makefile**, so while you may make modifications to the supplied Makefile for your own purposes, we will not be using it for grading.

Late assignments will not be accepted, so make sure to turn in your work by the deadline!