CS 451/551 Project 2: Multithreaded Steganography (version 2020.01)

February 25, 2020

DUE DATE: March 22 at 11:59PM

1 Introduction

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. This is followed by an example which should help understand the requirements. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To write a relatively simple, but non-trivial multi-threaded Rust program
- To get you to start to think about performance implications of your code.
- To provide the starting point to help you evaluate and improve performance.

1.2 Background

Trends in computing have increasingly moved towards horizontal scalability. This is true both in terms of distributed systems (e.g., cloud computing) as well as single machine computing (e.g., multi-threading). The single machine situation has reached the point where it is almost more difficult to find a single core device than a multi-core device. From a practical perspective, this

means that performance critical code must be designed and implemented to take advantage of using multiple CPU cores or it will be significantly limited. To that end, in this project you will build a multi-threaded version of the `steg` program from project 01.

1.3 Functionality

In Project 01, we wrote a program that can hide a message within an image file in such a way that the difference in the image file was not really visible to the naked eye. One limitation we had is that because of the way we encode the data, one byte of message is encoded into 8 bytes of image data. In a practical sense, this means that we were limited to encoding relatively small messages. To address that issue, this project will allow for encoding a message of *any* length. This is accomplished by splitting the message into different parts across several images.

Imagine that you have a message that is 100 bytes long and a single image that has 10 bytes "usable" for hiding the message (i.e., pixel bytes). A 100 byte message will require a minimum $100 \times 8 = 800$ bytes of storage. We could theoretically pack the entire message into 80 copies of the single image, however, recalling the requirements from Project 01, we used the null character (ASCII `255`) to mark the end of a message, and thus we would really need $80 + 1$ copies a minimum.

Thinking a little bit further, we might realize that the size of an image might not evenly divide the size of a message, and thus there will be some left over bytes of data that can't be used to encode the message. E.g., if your message requires 10 bytes and your image is 12 bytes, there are two extra bytes that do not contain a part of the encoded message. Indeed, this is the reason that we had the requirement of terminating encoded messages with a null character in Project 01.

If we extend this to the idea of spreading a message out across multiple images, it should be apparent that the problem will persist: we can't simply use every byte in the image to encode a part of our message, we still need to use the terminating null character in every image.

With this said, it leads to a relatively straight forward solution: given *at least one* image, we can encode a message of any length by breaking it up into pieces, and storing each piece in a different image. For example, if we are given one image to encode into, we would simply produce multiple copies of that image, with each one storing a different portion of the message to be hidden.

Once we have encoded our message, the next problem is decoding it. If we encode as described above, it is relatively obvious that we can extract a message part from each image, however, we are left with the question of how to reassemble the message in the right order. For this project, we will handle this via file names. I.e., if given a directory of files each containing a portion of a larger hidden message, the first part of the message will be encoded in the lexicographically "smallest" file name. E.g., "0000000.ppm" will contain a part of the message that comes before the part contained within "0000010.ppm"

Astute readers will notice that, for the most part, this problem is trivially parallelizable. In other words, the only real challenge in terms of implementing such a system non-serially is in partitioning the data. If you have you divided your message up into N parts, it is trivial to actually encode those N parts independently. The same can be said with respect to decoding the message, where the only non-independent comes when it's time to reassemble the message in the correct order.

And indeed, considering that we need 8x the bytes to store our message (plus whatever is needed to terminate a message within an encoded image) we have an opportunity to increase performance by exploiting this data independence: if our message is split into N parts, we can use $M \leq N$ threads to get increased performance.

2 Requirements

Create a new cargo project called `project02`, and change the name of the outputted binary to be `multi-steg` Submit a **gzipped** version of your project directory, such that `cargo run` in the decompressed file will build and execute your program. **Make sure that you run `cargo clean` before compressing your directory and submitting!**

It should be possible to invoke the `multi-steg` program with upto two arguments:

- When invoked with zero arguments, the `multi-steg` program should simply output a usage message on standard error.
- When invoked with two arguments, the `multi-steg` program should use the number of threads specified in the first argument ot unhide the message concealed in the images located in the second argument, and print the decoded message to `STDOUT`.
- When invoked with four arguments, the `multi-steg` program should use the number of threads specified in the first argument to hide the message specified located in the file specified by its second argument within the images in the directory specified by the third argument and in the PPM image file named by its first argument and write the resulting images to the directory specified by the fourth argument.

From the above, two major questions arise.

What to do if the message is to big to fit in the images in the directory passed in? If the message you are trying to hide is too big to fit within the images in the supplied directory, you simply reuse the images.

What to do if there are images in the input directory when hiding or unhiding? Your code should be able to detect if images it is trying to encode to our decode from are valid PPMs. If the files are *not* valid PPMs, your program can log the issue to `STDERR`, but it should *not* crash, or panic, or exit in anyway. Your program should simply skip those files.

2.1 Benchmarking

This is *required* for graduate students and will result in extra credit for undergraduate students.

In addition to the working program, all Graduate Students are *required* to provide a short report that demonstrates the performance of their program. At minimum, this report *must* show the overall performance of your program as you vary the number of threads and also the throughput of your program (i.e., how many bytes per second it can process.) Ideally you would measure this across a variety of axes. E.g., how does it vary with the number of threads you use? Is there a linear relationship with the size of the input message, or the size of the images being used to hide the message, or the number of images used?

Additionally, there are other things that can be measured, for example, you can use benchmarking to discover "hot spots" in your code that are slowing things down due to the frequency they are executed. By repeatedly benchmarking and evaluating the results, you might discover new ways to optimize your code. Any such benchmarking and optimizations should be noted in the report.

2.2 Allowed Libraries

There are precisely *three* external crates you are allowed to use in this project:

1. `Criterion.rs` (`https://github.com/bheisler/criterion.rs`) is a benchmarking library. You are not *required* to use it to benchmark your code; there are a variety of ways to do this. However, I am using it to benchmark my code.
2. `log` (`https://github.com/rust-lang/log`) which can help make debugging your code a bit easier than using `println!()` and `eprintln!()`.
3. `libsteg` an early version of which you have been provided the source code for. Although you can use this, I would advise against it in favor of the development of your *own* `libsteg`. You are welcome to liberally steal from the code I provided, however, if you do make your own external crate, be sure to include it in your submission (in which case your `project02.tgz` should include two directories, the one containing the `multi-steg` implementation as well as your crate).

The use of any crates not on the above list (and that includes manually using any dependencies brought in by the above crates) will result in a zero on the project.

2.3 Example Program

You have been supplied with a binary of my solution to the project. There are two compiled versions, one for Linux and one for macOS. Apologies to Windows users, but I don't have access to a Windows machine.

You have also been supplied with a `bible.txt` which is some version of the Bible I got from project Gutenberg (noted at the end of the file). This file is much bigger than can fit in any of the sample images from project 01.

You can also make it bigger if you want to test larger files:

```
# this will create a file 2x as big as ‘‘bible.txt’’

cat ../data/bible.txt ../data/bible.txt > ../data/bible-2x.txt
```

Below are some example runs of my solution.

```
# use 10 threads to encode the message located in ‘‘../data/bible.txt’’
# using the images located in ‘‘aux/in’’
# store the result in ‘‘out’’

./multi-steg 10 ../data/bible.txt aux/in out

##########
# message.len() = 4452516
##########
**********
* Checking input directory for valid PPMs
Found valid PPM: "aux/in/rose.ppm"
Potential input file not a valid PPM: "aux/in/rose-blue-flower-rose-blooms-67636.png": BadHe
Potential input file not a valid PPM: "aux/in/rose.png": BadHeader("Bad Magic Number: \u{89}
Found valid PPM: "aux/in/rose-blue-flower-rose-blooms-67636.ppm"
Found valid PPM: "aux/in/marguerite-daisy-beautiful-beauty.ppm"
Potential input file not a valid PPM: "aux/in/marguerite-daisy-beautiful-beauty.png": BadHea
Found valid PPM: "aux/in/dahlia-red-blossom-bloom-60597.ppm"
Found valid PPM: "aux/in/garden-rose-red-pink-56866.ppm"
Potential input file not a valid PPM: "aux/in/logo.png": BadHeader("Bad Magic Number: \u{89}
Potential input file not a valid PPM: "aux/in/garden-rose-red-pink-56866.png": BadHeader("Ba
Potential input file not a valid PPM: "aux/in/dahlia-red-blossom-bloom-60597.png": BadHeader
Found valid PPM: "aux/in/logo.ppm"
Potential input file not a valid PPM: "aux/in/pexels-photo-86243.png": BadHeader("Bad Magic
Found valid PPM: "aux/in/pexels-photo-86243.ppm"
##########
# There are 7 valid input images
# aux/in/logo.ppm is 640 x 480 pixels, 921600 bytes
# aux/in/pexels-photo-86243.ppm is 525 x 350 pixels, 551250 bytes
# aux/in/garden-rose-red-pink-56866.ppm is 526 x 350 pixels, 552300 bytes
# aux/in/rose-blue-flower-rose-blooms-67636.ppm is 528 x 350 pixels, 554400 bytes
# aux/in/rose.ppm is 70 x 46 pixels, 9660 bytes
# aux/in/marguerite-daisy-beautiful-beauty.ppm is 538 x 350 pixels, 564900 bytes
# aux/in/dahlia-red-blossom-bloom-60597.ppm is 467 x 350 pixels, 490350 bytes
----------
# total 3644460 unique bytes in storage available to store 4452516 bytes of message
# we can store 0.1023146239115143% of the message before needing additional storage
$ Finished generating jobs (4452516 characters to be encoded)
$ Launching image encoding jobs...
$ Spawning thread #1
```

```
$ Spawning thread #2
$ Thread #1 performing encode...
$ Thread #1 encoding image #0000000000
$ Spawning thread #3
$ Thread #2 performing encode...
$ Thread #2 encoding image #0000000001
$ Thread #3 performing encode...
$ Thread #3 encoding image #0000000002
$ Spawning thread #4
$ Spawning thread #5
$ Spawning thread #6
$ Spawning thread #7
$ Spawning thread #8
$ Spawning thread #9
$ Spawning thread #10
$ Thread #5 performing encode...
$ Thread #5 encoding image #0000000004
$ Thread #5 encoding image #0000000014
$ Thread #6 performing encode...
$ Thread #6 encoding image #0000000005
$ Thread #7 performing encode...
$ Thread #7 encoding image #0000000006
$ Thread #8 performing encode...
$ Thread #8 encoding image #0000000007
$ Thread #9 performing encode...
$ Thread #9 encoding image #0000000008
$ Thread #4 performing encode...
$ Thread #4 encoding image #0000000003
$ Thread #10 performing encode...
$ Thread #10 encoding image #0000000009
$ Thread #3 encoding image #0000000012
$ Thread #10 encoding image #0000000019
-- snip --
$ Thread #1 encoding image #0000000060
$ Joined on thread 1
$ Joined on thread 2
$ Joined on thread 3
$ Thread #8 encoding image #0000000067
$ Joined on thread 4
$ Joined on thread 5
$ Joined on thread 6
$ Joined on thread 7
$ Joined on thread 8
$ Joined on thread 9
$ Joined on thread 10


# Use 10 threads to decode the message hidden in the files located in
```

```
# the ‘‘out’’ directory.
# Also redirect STDOUT to /dev/null, which will make sure it’s not printed.

./multi-steg 10 out 1>/dev/null

$$$$$$$$$$
$ Finding valid PPMs in input directory
$ Spawning thread #1
$ Spawning thread #2
$ Spawning thread #3
$ Spawning thread #4
$ Spawning thread #5
$ Spawning thread #6
$ Spawning thread #7
$ Spawning thread #8
$ Spawning thread #9
$ Spawning thread #10
$ Thread #4: Found valid PPM: "out/0000000026.ppm"
-- snip --
$ Thread #10: Found valid PPM: "out/0000000000.ppm"
$ Found 68 valid paths
$$$$$$$$$$
$ Decoding data...
$ Received 7 message parts
$ Total message parts received: 7
$ Received 6 message parts
$ Total message parts received: 13
$ Received 6 message parts
$ Total message parts received: 19
$ Received 7 message parts
$ Total message parts received: 26
$ Received 7 message parts
$ Total message parts received: 33
$ Received 7 message parts
$ Total message parts received: 40
$ Received 7 message parts
$ Total message parts received: 47
$ Received 7 message parts
$ Total message parts received: 54
$ Received 7 message parts
$ Total message parts received: 61
$ Received 7 message parts
$ Total message parts received: 68


# Decode the message hidden in the files at ‘‘out’’
# Compare the unhidden message to the expected output
```

```
./multi-steg 10 out | diff ../data/bible.txt -

# no output indicates the decoded message and the expected
# input are identical
```

3 Grading

Undergraduates:

- An earnest effort was made to complete the project.
  - **20 points**
- Program compiles.
  - **30 points**
- Your program can properly encode messages using multiple threads.
  - **20 points**
- Your program can properly decode messages using multiple threads.
  - **20 points**
- Invalid PPM files in input directories do not affect your program.
  - **10 points**
- You provide a benchmarking report.
  - **20 points extra credit**

Undergraduates:

- Program compiles.
  - **30 points**
- Your program can properly encode messages using multiple threads.
  - **20 points**
- Your program can properly decode messages using multiple threads.
  - **20 points**
- Invalid PPM files in input directories do not affect your program.
  - **10 points**
- Benchmarking report.
  - **20 points**

4 Hints

- It took me a solid weekend to complete the project. I was *not* able to complete it in a single session.
- You need to *carefully* think about how you are going to solve this project. The way I went about it was to first complete a single threaded version that was able to encode/decode using multiple images. Then, I wrote code that spawned a single thread that performed the actual encoding/decoding. Then I modified the code to use a variable number of threads.
- My program is in the 750 lines of code range *not including* the `libsteg` code.
- I made use of regular style threads (where you `join` on them) as well as a multiple producer/single consumer channel.
- There are many useful methods in Rust's standard library useful for dealing with files in this project. In particular, you should probably take a look at the `Metadata` struct (`https://doc.rust-lang.org/std/fs/struct.Metadata.html`).
- Be cognizant of the fact that hiding a message will result in output that is $\geq 8x$ than the given input. The sample `bible.txt` available is 4MB and results in 34MB of encoded images.