CSC 412 – Operating Systems

Final Project, Spring 2022

Wednesday, April 27th 2022

Due date, midpoint version: Friday, May 6th 2022, 11:55 pm
Official due date, final version: Wednesday, May 11th 2022, 11:55 pm
Project accepted with no penalty until: Sunday, May 15th 2022, 11:55 pm

1 What this Assignment is About

1.1 Objectives

In this final project, you will combine together several things that we saw this semester:

- pthreads,
- mutex locks,
- memory sharing,
- pipes.

1.2 Handouts

The handout for this final project (besides this document as a pdf file), consists of 2 data sets (a small one and a larger one).

1.3 Solo or Duo?

This project was designed to be done by a group of two students, but if you want to do it solo, I won't prevent you (but there won't be any load reduction either).

2 The Basic Problem

2.1 Aggregation of satellite images

The concept of the problem is that we are collecting partial pictures taken by different satellites[1] and are supposed to aggregate them in a few different map images.

[1] **Please** don't waste time search online for "merge satellite images" or any such nonsense. We are dealing here with a *very* abstracted and simplified version of the real problem. Any code you might find online is going to be *way* more complex than called for, and a general nightmare to integrate to this assignment.

2.2 User interface

Whether the user interacts with a program or with a script is completely up to you, as long as it implements the desired behavior. For this reason, i will henceforth refer to "the system," and it will be up to you do decide which part of that "system" is script and which is C++ program executable.

In any case, you must provide a script to build the various executable components of the system, at the locations where your system expect to find them (hint: a hard-coded path on *your* personal desktop is not a good location).

From the point of view of the user, the system watches a folder where capture data files, possibly thousands of them, will be dropped, not in a single time. The system will also take map output commands (see Subsection 2.6) from the standard input.

Finally, the system should accept a shutdown command whose execution will result in the proper termination (with no crash or memory-related error printout) of all the components (pipes, shared memory, threads, and processes) of the system.

2.3 Configuration file

The system is launched with a single argument: The path to a configuration file providing the following information (in this specific order):

- The path to a "watch folder."
- The path to an output folder where aggregated image maps will be output.
- The number of maps to be aggregated.
- The height (number of rows) and width (number of columns) of each image map.

For example, I should be able to run your system on my computer with the following contents for my configuration file:

```
/Users/jyh/CSC412/Final/Watch /Users/jyh/CSC412/Final/Output/ 3
300 400 1500 2000 1000 200
```

In that case, I would be aggregating "satellite" data to produce three maps: the first one being 400 pixel wide and 300 pixel high, the second 2000 pixel wide and 1500 pixel high, and the last one 200 pixel wide and 1000 pixel high.

Note: If the watch folder or the output folder don't already exist, your system should create them. As usual when dealing with folder paths, I remind you to be careful with that path-ending slash. As with any path, I also remind you that your system must be able to accept any valid path.

2.4 Capture data

Each image captured by a satellite is a small rectangle, but some of the pixels may be missing (either because of excessive noise or because the image was produced by a small rotation (to align it with the maps' axes).

Each satellite comes under the form of a single text file with the `.dat` extension. The first line of the file gives (in this specific order):

- The index (1–n) of the map that the image should be integrated to;
- The row and column coordinates for the upper-left corner of the image fragment in the map (somehow, the "satellite" has computed the coordinates of the fragment relative to the map);
- The height (number of rows) and width (number of columns) of the image fragment.

The remainder of the file simply lists the pixel information (listed row-by-row, column-by-column) for all pixels in the fragment. For each pixel, the data file lists four numbers (`unsigned char`):

- The usual red, green, and blue channel values;
- A fourth number that indicates whether the color data for that pixel are valid (value 1) and should be integrated to the map, or invalid (value 0) and should be ignored.

Note: Fragments won't be smaller than 4×4 (such a size would really only be encountered for debugging/testing purposes) and could be up to a hundred pixel high/wide.

2.5 Data integration

The different snaphots provided by the satellites will overlap, so you will end up getting multiple measurement for the same location on the map. You cannot simply overwrite the latest value stored them. Rather, because each individual measurement is noisy, you should compute an average of all the values received for that pixel location (so far). Naturally, invalid pixel data should not be counted in the average.

2.6 Map output

A request for map output will provide, besides the desired map's index, the red, green, and blue components of a "substitution color" to use in the output map at the location of pixels where no valid data has been recorded yet:

```
map <map index> <subst. red> <subst. green> <subst. blue>
```

The map requested will be output as a TGA image file to a designated output folder, under the following naming scheme:

```
map_<map index>_<time stamp>.tga
```

2.7 Architecture of the C++ program

You are going to implement three different versions of the system, each using a different architecture based on pipes, threads, or shared memory. I don't give you any details or directions on the way you should proceed to implement each version. You should be able to figure ir all on your own based on past assignments, labs, and code handouts.

In all versions of the system, you must make sure that none of the processes or threads involved hangs inappropriately. For example, a map process should not hang because another map process is slow to process the information arriving to it.

3 Version 1: A Dispatcher, and One Process Per Map

3.1 Specifications

In this version, a dedicated dispatcher process creates one process per map. Upon receiving or reading (depending whether this process also performs folder watching duties) the path to a satellite data file, the dispatcher will forward this information to the appropriate map. The dispatcher will also forward map output commands to the appropriate map process.

3.2 Extra credit 1.1 [4 pts]

Instead of computing the aggregated value as an average of the pixel values collected at a location, compute a median.

3.3 Extra credit 1.2 [8-25 points]

3.3.1 The problem

In this revised version, the sensors collecting the map fragments have a bit of uncertainty on the correct location of the fragment: It comes with an uncertainty range. The revised configuration file contains the following information (in this specific order):

- The path to a "watch folder."
- The path to an output folder where aggregated image maps will be output.
- A pair of integers giving the vertical (row) and horizontal uncertainty of all fragment locations.
- The number of maps to be aggregated.
- The height (number of rows) and width (number of columns) of each image map.

For example, I should be able to run your system on my computer with the following contents for my configuration file:

```
/Users/jyh/CSC412/Final/Watch /Users/jyh/CSC412/Final/Output/ 4 5
3 300 400 1500 2000 1000 200
```

The new uncertainty parameters tell us that all fragment locations are given with a ± 4 vertical uncertainty and ± 5 horizontal uncertainty. In other words, a map fragment with the following second line:

```
79 220 82 65
```

could in fact have its upper-left corner located at any row index in the range $[75, 83]$ and column index in the range $[215, 225]$. In other words, instead of a single possible location for the upper-left corner, there would be $9 \times 11 = 99$ possible locations. The "best" location for a given fragment

would be the one giving the best agreement with the other fragments with which it overlaps. A fragment that doesn't overlap with any other fragment should be placed at the location reported by the sensor.

Note: For reference, this is a simplified version of the `image registration problem`. The full-fledged problem has to deal with rotation and possibly a bit of scaling as well.

3.3.2 Partial credit for a discussion and sketch of solution [8 pts]

Discuss how you could solve this problem. Explain what data structures would be involved in your solution. To get full credit for this part, you need to get as close as possible to the point where a competent programmer, given enough time, could start implementing the solution from your outline.

3.3.3 Full credit for a complete implementation [25+ pts]

Impress me. :-)

3.3.4 Data for this EC

I will provide a modified dataset for this EC over the weekend.

4 Version 2: A Single Multi-threaded Process

4.1 Basic structure

This is the least "system programming-oriented" version, as it is implemented as a single, complex, monolithic application.

In this version, there is a single process that allocates the three maps and creates threads dedicated to map aggregation and other threads dedicated to folder watching duties (either directly, or by communicating with a script), and handling of user commands. In the basic form, the process would create a new thread for each satellite data fragment to integrate to a map.

4.2 Beware of race conditions

As mentioned earlier, different satellite data files may correspond to overlapping rectangles within the same map, which leads to race conditions. You must take care of that problem. Another race condition is encountered when a request for map output arrives. Whatever solution you end up with, you should present the reasoning that went into its design (such as the compromise between performance and complexity).

4.3 Extra credit 2 (10 pts): Thread pool

Instead of creating a new thread each time a new data file needs to be process, create in advance a pool of threads. Idle threads should put themselves on a waiting list and should be awaken by the dispatcher thread when a new task arrives.

5 Version 3: A Dispatcher, Using Shared Memory

In this version the dispatcher should setup shared memory segments (at least one per map) and launch a process for each map. When a request for a map output is received, the dispatcher process is the one that should produce the output image.

6 What to Hand in

There will be a 5 pts penalty for multiple submissions (under different names) by the same group. That penalty will apply to all members of the group, regardless of who gets blamed for posting the "wrong" project[2], so make sure that you know who is tasked with submitting the final version.

6.1 Midpoint version

Each team **must** submit a "midpoint version" of your final project (one submission only per group). What I mean by "midpoint version" is an early version of the program that at least partially implements one version required for the assignment.

- It must compile;
- It must run without crashing in normal operation (i.e. it's OK if error handling is not there yet);
- It must produce an output (it's OK if the image produced is all black, as long as it has the right dimensions)

There is no need to provide a report with the midpoint version.

6.2 Final version

Your submission should include:

- Your report;
- The Doxygen-produced html documentation.

[2] It happens ever semester and it's really irritating. I grade a FP and then discover another submission by another member of the same group, and of course *that one* is the right one, the other was a mistake, and the grading time on the first project was a complete waste.

- Source code: Each version should be presented in a separate folder named `Version1`, `Version2`, etc.
- If you wrote bash scripts for this assignment, then they should be placed in a folder named `Scripts`, and be setup to run from inside this folder.
- If you implemented any of the EC sections, make sure to say so in the report, and to indicate for which version of the project you implemented the EC.

6.3 Source code

The source code should be properly commented, with consistent indentation, proper and consistent identifiers for your variables, functions, and data types, etc. All your custom data types (all instance variables and all functions) and all free functions should have full Doxygen documentation. Please note that if you decide to implement classes (with instance functions), as opposed to "dumb storage C-style structs" for this assignment, then I am going to grade your code for proper OOP implementation.

6.4 Doxygen documentation

If you implement classes (I don't see any particular reason would you would have to)m, make sure that you export documentation for all members of your classes, not only public ones.

6.5 Note on the report

Your report should identify the team members. In the report, you will have to document and justify your design and implementation decisions (in particular regarding synchronization), list current limitations of your program, and detail whatever difficulties you may have run into.

The report should provide a short "user manual" explaining how to run the different versions of your project.

7 Grading

- Version 1 completed and performs as required: 15 pts
- Version 2 completed and performs as required: 15 pts
- Version 3 completed and performs as required: 15 pts
- good code design: 10 pts
- comments: 10 pts
- general quality of the code: 15%
- Doxygen documentation: 10%
- report: 10%