CSC 412 – Operating Systems

Programming Assignment 05, Fall 2020

Thursday, November 29th, 2018

Due date: Tuesday, November 10th, 11:55pm.

1 What this Assignment is About

1.1 Objectives

The objectives of this assignment are for you to

- Establish communication between processes using pipes;
- Work with a small third-party library (here, to load and write TGA images);
- Learn to produce a DLL and then build your program against this DLL rather than against source file;
- Write a script that continuously monitors a "drop folder;"
- Oh, and yes, we finally switch to C++!

This is an individual assignment.

1.2 General idea of the complete system

The general idea of the system is that your script is going to monitor a "drop folder" where the user can submit two kinds of files: TGA images files or text files containing lists of tasks to perform on existing image files (e.g. rotation, cropping, etc.). The script will move image files to a designated data folder. When encountering a list of tasks, the script will dispatch these tasks to a process that will get them executed by the appropriate utility program. As usual, this assignment will include different versions, as we work toward the final system.

1.3 Handout

Besides this document, the handout for this assignment is a zip archive containing the code of a small C++ library for loading and writing images in the TGA file format, and performing elementary operations on these image, as well as a set of images in the TGA file format.

1.4 Reminder on specifications

When I tell you "name the program/script/function" `XYZ` and put it in a folder named `ABC`, then I expect you to do it exactly the way I say, and you will get points taken off if you don't. There are plenty of places in these assignments that you can make any design and implementation choices you want, but when I give specs, they are to be followed.

Feel free to think of me as your "point-haired boss." I am fine with that. When you will be out there in industry, you will be expected to follow company policies, no matter how bone-headed they may be or seem to be. These are my policies. Follow them or lose points.

2 What to Do, Part I: TGA Image Files and Library Building

2.1 The TGA file format

The TGA (Targa file) image format is one of way-too-many image formats that you may run into. Other formats provide better compression or support for a wider range of colors, so why use this format at all? Because the format for *uncompressed* `.tga` image files is the only truly multi-platform format that is easy to read and write. We could use a more complete library such as `freeimage` or `ImageMagick`, and maybe we will do just that in a future assignment, but, as long as the image file used stores data uncompressed and does not contain any comments, the code supplied here will do the job.

2.2 The code supplied

The code supplied consists of the shell of a small image processing library, stored into the folder `ImageLibrary`, organized into four folders

- `include` contains the library's header file;
- `src` contains the implementation of functions for creating, reading and writing images, or performing elementary operations on them;
- `applications` contains source code for small standalone utilities performing elementary images operations (using the library);
- `lib` will contain the DLL that you are going to build.

You don't need to do anything with the code of the library, just build it, so please don't waste time trying to understand how the code works. If you run into problems or bugs, just report them to the silly developer for support.

2.3 A bit of information about color images and rasters

I am not sure that you are going to need know anything about this in order to complete the assignment, but I would rather have this information here than have to post a revision that some students will miss, or a post on the discussion boards that for sure almost nobody looks at.

Pretty much all libraries dealing with image and video data[1] manipulate image data under the form of a 1D "raster," as shown in Figure 1.

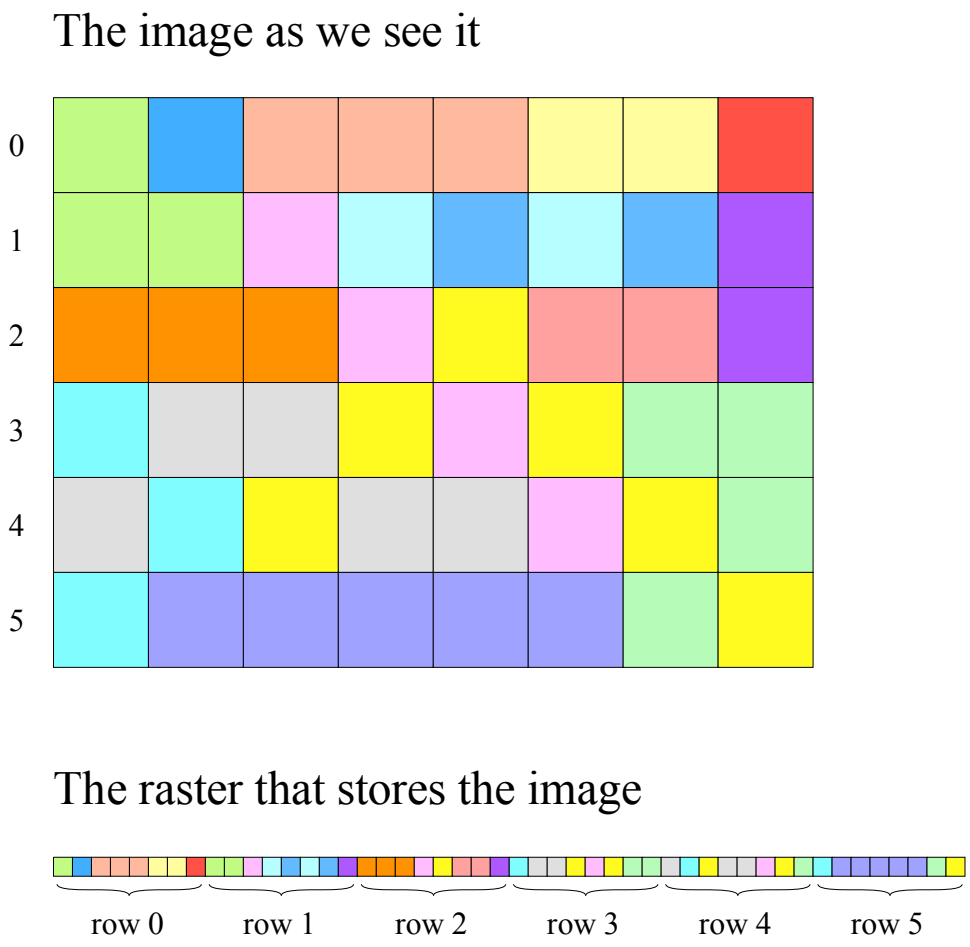

Figure 1: An image stored as a 1D raster.

So, if our image has m rows of n columns and each pixel is stored on k bytes (for a total of $n \times k$ bytes per row and $m \times n \times k$ bytes for the entire image), then the pixel at row i and column j in the image can be addressed at index $i \times n \times k + j \times k$ of the 1D raster.

In the uncompressed TGA file format, an image file stores each pixel as 3 unsigned bytes (range 0 to 255) encoding the red, green and blue color channels. When we load an image in memory, however, it is more convenient to store each pixel on an even 4 bytes. Besides allowing image processing and other graphic applications to use this additional byte to encode *transparency* (or the "α channel," as it is called), this also means that a pixel occupies the same space as an `int`, so that we can view our 1D raster either as an `int*` pointing to a pixel or as an `unsigned char*` pointing to a color channel of a pixel.

[1] An important exception is Apple's old QuickTime library and the `libquicktime` open-source replication of the API for a tiny bit of QuickTime. I mention this here because we may have an encounter with `libquicktime` before the end of the semester.

Note that different applications may store the bytes of the color channel in different orders. The most common orders are `argb` (alpha, red, green, blue) and `rgba`. Throughout this assignment, and future assignments dealing with images, we will use exclusively the `rgba` format.

What this means is that if I want to access the color information of the pixel at row i and column j of the same $m \times n$ image, then I could get a pointer to this pixel's color channels, each seen as an `unsigned char` by writing

```
unsigned char* rgba = (unsigned char*)raster + 4*(i*n + j);
```

Now, I can view this pointer as an array and access directly my color components:

- `rgba[0]`: red channel,
- `rgba[1]`: green channel,
- `rgba[2]`: blue channel,
- `rgba[3]`: alpha channel.

Alternatively, I could see my pixel as a 4-bye `int` that I access through an `int*` pointer by writing

```
int pixel = *((int*)raster + i*n + j);
```

Now I can access my color channels by extracting the different bytes of the `pixel` variables. There is just a small problem: the byte order. Different CPU architectures have a different way to encode `int`, `float`, etc. Or rather, the encoding is the same, but the byte order is different. Intel processors (and their AMD clones) are "small-endian," which means that their least significant byte is stored first, and their most significant byte is stored last.

2.4 How to build the applications "by hand"

There are 6 independent utility programs that you can build separately. Assuming that you are in the folder `Handout/ImageLibrary/applications` folder, you can build, for example, the `flipV` utility with the command[2]

```
g++ -I../include flipV.cpp ../src/ImageIO.cpp ../src/ImageIO_TGA.cpp
../src/utilities.cpp ../src/RasterImage.cpp -o flipV
```

The `-I` option lets you indicate to `g++` where to look for header files of your project. You could give multiple locations where to look for header files, but in this project we only need our library's header files.

[2] If you are on a Mac, then the default compiler is clang++ and you need to specify the dialect of C++ that you want to use, as well as the standard library to build against. The command is a bit longer:
`clang++ -std=c++17 -stdlib=libc++ -I../include flipV.cpp ../src/ImageIO.cpp ../src/ImageIO_TGA.cpp ../src/utilities.cpp ../src/RasterImage.cpp -o flipV`

2.5 Build `ImageLibrary` as a static and a shared libraries

What I want you to do now is to build the `ImageLibrary` code as a library that you can use to build new programs against. Also, in future assignments, you may be asked to distribute your library (**not** your source files) to another student who will use it to develop their project. You would then be graded for the performance of your library and the level of support you provide to your "customer," so it's important that you pay attention to what you are doing here.

Luckily, there are some very good tutorials online on how to build and use libraries. For example, the following two pages should give you a good starting point:
`http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html`
`https://blog.feabhas.com/2014/04/static-and-dynamic-libraries-on-linux`
After you have built your shared and static libraries, make sure that you can still build your test program against the static library, instead of against the image I/O source code. Running a program built against the shared library requires installing the shared library in the build path for the system. We will mess with that another time. For this assignment, I just want to make sure that you can build either kind of library.

3 What to Do, Part II: Dispatcher Programs

This week again, we are dealing with a dispatcher program that will receive from the script lists of tasks under the form of job files, and will then dispatch these tasks to the utility programs. Once more, we are going to develop two versions of the dispatcher: without and with pipes. In both versions, the dispatcher program will need to read a job file, so we will start with this part of the work.

3.1 Syntax of a task in a job file

A job file contains a least one command. Each command appears on a separate line and could be one of the following:

- `flipH` *image name* produces a copy of the image that has been flipped horizontally;
- `flipV` *image name* produces a copy of the image that has been flipped vertically;
- `gray` *image name* produces a copy of the image that has been converted to gray-level values;
- `crop` *image name* x y w h: writes to a new image file the rectangle of the input image with its upper-left corner at (x, y), width w and height h;
- `rotate` <*rotation*> *image name*: writes as a new file the image resulting from applying to the input image the rotation specified, where <*rotation*> is one of `l` (left rotation by 90 degree), `r` (right rotation by 90 degree), `ll` or `rr` (rotation by 180 degree).
- `end`: orders the dispatcher program to terminate its execution.

Note 1: Yes, no "number of lines" field on this file format, since you usually ignore them anyway and prefer to read "while not at the end of the file yet."

Note 2: If an `end` task appears in the middle of a job file, the following tasks should not be performed.

Note 3: The entries in the job file only give the *name* of the image files, not a full path, and they don't indicate where the output image should be written either. This will be discussed next.

3.2 Dispatcher program, Version 1

3.2.1 Creation and communications with the bash script

This program is launched by the script and will get among its arguments a list of tasks (whether as a list of strings or as the path to a job file is completely up to you). I leave it up to you to determine the arguments expected by your program, but, presumably, it should also get the path to the folder where to find the image files and that to the folder where to write the image produced. Upon completing the last task on the job list the dispatcher should just exit.

3.2.2 Communication with the utilities

This part is nearly identical to what we did in Lab 06: The dispatcher program should simply launch a utility process (through a combination of `fork()` and `execXY()`) to perform the appropriate task.

3.3 Dispatcher program, Version 2

3.3.1 Creation and communications with the bash script

This program is launched by the script and will keep communicating with the script through at least one pipe, until it receives an `end` task. I leave it up to you to determine the arguments expected by your program, but, presumably, it should get at some point the path to the folder where to find the image files and that to the folder where to write the image produced. Similarly, I leave it up to you to define what kind of pipes will be used to communicate between the bash script and the dispatcher program, who creates the pipe(s), and the format of what gets passed on the pipe(s).

3.3.2 Communication with the utilities

This part is identical to that of Version 1.

3.4 Dispatcher program, Version 3

3.4.1 Creation and communications with the bash script

There should be no major difference between this version and the first one, regarding launch and communications with the script.

3.4.2 Communication with the utilities

In this version we are going to add one specialized dispatcher layer between the "main" dispatcher and the utilities. In other words, there will be a dedicated "rotation" dispatcher that only handles rotation tasks, a dedicated "crop" dispatcher that only handles rotation tasks, etc. These are resident dispatchers that are launched by the main dispatcher and keep waiting for tasks sent to them by the main dispatchers, via pipes.

Again, the details of what arguments are passed to the specialized dispatchers when they are created, how the communications pipes are set up, by whom, and the format of the messages passed along these pipes are completely up to you.

One last thing: When the main dispatcher receives an `end` task, it should pass it along to the specialized dispatcher and wait for them to terminate before it can end itself exit.

3.5 Extra credit

3.5.1 Extra credit 1: Logging (6 points)

Your dispatcher process should maintain a long of all the task executed, indicating which were successfully completed (returned 0) and which didn't (returned an error code).

3.5.2 Extra credit 2: Detailed logging (2 more points)

In cases where the utility couldn't complete its task, the dispatcher program should give plain-text causes for non-completion (e.g. "file not found").

4 What to Do, Part III: Scripting

4.1 Location

Make sure that all your scripts run properly from the folder named `Scripts` placed at the root of the `Prog05` (or `Handout`) folder

4.2 Script 1: The Builder

Your script `script01.sh` should take in a single argument: The path to the `ImageLibrary` folder. The script should first build the dynamic and static libraries (in the `lib` subfolder. After that, it should build the `gray`, `crop`, `flipV`, `flipH`, `rotate`, and `comp` utility programs wherever you run the script from (I don't specify a location for the executables you build). Finally, your script should build the different versions of the dispatcher program(s) that you have completed.

4.3 Script 2: The watcher

Your script `script02.sh` will be launched with the following arguments:

1. The path to a "drop folder" to watch;

2. The path to a folder where to find input images (henceforth referred to as the "data folder");

3. The path to a folder where to store the output of the image processing processes (henceforth referred to as the "output folder").

Your script should verify that the three paths are different and create the folders if they don't already exit. Once the folders have been created, the script shall continuously monitor the drop folder for any new file and will take action for two types of files:

- Image file in the TGA file format, that is, with a `.tga` extension: Move the image file to the data folder.

- Job files (with the extension `.job`) containing a list of image processing tasks to perform on some images of the data folder (specifications for job files are given in Subsection 3.1). Your script should send this file to the dispatcher program (Version 1, as seen in Subsection 3.4), and move the job file to a `Completed` folder when the dispatcher program reports that all tasks have been completed.

Any other type of file should simple be ignored.

Note 1:
To implement the "folder watch" functionality, you should probably look at the `inotifywait` system call.

Note 2:
Be careful not to keep "finding" the same job file over and over.

4.3.1 Extra credit: up to 8 points

Implement a logging system for the drop folder. This involves creating a `Logs` subfolder inside the output folder and moving all completed job files to that folder (3 points). For 3 more points, add a time stamp prefix to the name of the file (e.g. if the job file myJob.job arrived on November 10th at 9:13:48 AM, then write the logged file should be `2018-11-10-09.13.48`) I am not imposing a particular format for the time stamp, but the format should be compatible with lexicographic ordering, so that the log files are listed in the according to their date. For 2 more points, add a "completion time" stamp as a prefix to each task line in the job file.

4.3.2 Extra credit (4 points)

Different path *strings* could in fact refer to the same folder, through the magic of symbolic or hard links. Verify that the *folder* is different, not just the string.

4.4 Script 3: Piped version)

Your script `script03.sh` will be launched with the following arguments:

1. The path to a "drop folder" to watch;

2. The path to a folder where to find input images (henceforth referred to as the "data folder");

3. The path to a folder where to store the output of the image processing processes (henceforth referred to as the "output folder").

It will have the same outer behavior as `script02.sh`, with one major difference: This time, the dispatcher program is `dispatcher2` (see Subsection 3.3 or `dispatcher3` (see Subsection 3.4), which instead are "resident" dispatcher, in the sense that they don't quit after they have completed all the tasks of a job file, and instead wait for more job files. Your script must therefore establish pipes to be able to send tasks to the dispatcher program.

4.5 Extra credit Script 3: The explorer (8 points)

Your script `script03.sh` should take as inputs the path to a "reference" image file and the path to a "search" folder. Your script should find all TGA images stored in the search folder *or any of its subfolders*, and call the `comp` utility to verify if any image in the search folder is identical to the reference image. Whenever an identical image is found, your script should print the path to this image to the standard output.

5 What to submit

5.1 Report

You should submit a report in which you explain the design of your multithreaded programs and the communications between server processes and script, and between threads of a process. Identify any current limitation of your code: Can it crash? Under what circumstances? Do you make any assumptions in addition to the ones allowed in this assignment?
The report should be submitted as a `.pdf` file (Adobe Acrobat document).

5.2 Organization

The root folder should be named `Prog05`.

In that folder, you should put your report and three subfolders

- Version 1: contains the 3 source files and the script for Version 1;
- Version 2: contains the 3 source files and the script for Version 2;
- Tasks: contains a few examples of job files that you used to test your system.

The shell script and C source files are text files with their proper file extension (under no circumstance should they be pasted into a Word document[3]).

[3]This extends to future communications with us: Do not send us source code copied into a Word document. This has never been a correct way to exchange source code.

5.3 Grading

- Execution: 60%
  - C code: 40% [25% v1, 15% v2]
  - script code: 20 [12% v1, 8% v2]%
- Quality of code: 25%
  - C code: 10%
  - script code: 5%
  - javadoc-style comments: 10%
- Report: 15%