CSC 412 – Operating Systems

Programming Assignment 06, Fall 2020

Thursday, November 12th, 2020

Due date: Monday, November 23rd, 11:55pm.

1 What this Assignment is About

1.1 Objectives

This assignment is different from the last one in the sense that you get a complete working program to work with, and need to add functionality. The objectives of this assignment are for you to

- Get some practice at modifying an existing code base, and in particular develop a "feel" for how much of the existing code base you must understand in detail and how much only requires very superficial, global understanding;
- Use third-party external libraries in a C++ project;
- Use named pipes to establish communications between a `bash` script and processes running a C executable.
- Implement different forms of multithreading;
- Synchronize access to shared resources using either a single, global mutex lock or multiple mutex locks.

This is an individual assignment.

1.2 Handout

The main handout for this assignment is a small graphic application that updates and displays the states of a cellular automaton's cells. The code provided should build on Linux, mac OS, and Windows.

2 Part I: Building and Running the Handout

2.1 The code handout

The code handout consists of 2 source files and 1 header file. Normally, you shouldn't have to make any changes to the header and source file dealing with the "front end." Even within the `main.cpp` source file, there are clearly identified parts that you should probably don't mess with. Modify any of the "don't touch" sections at your own peril.

2.2 OpenGL and glut

This handout, as-is, requires the OpenGL and glut libraries. OpenGL is a library for doing 2D and 3D graphics. OpenGL is blissfully OS-unaware: It has no notion of time. It is equally ignorant of the things called windows, menus, mouse, and keyboard. All it does is render a scene into a buffer. If that buffer happens to be attached to a window, then you will see a rendering of the scene on your display, but OpenGL would be equally happy to render into a buffer attached to the file system or to an output device such as a printer.

Almost every GUI library (including the native GUI library of your OS of choice) provides functions to assign a window's buffer as the place for OpenGL to do its rendering into. Keyboard, mouse, and timer events can be handled either through the GUI's API or through regular system calls. The problem is that native GUI libraries are not portable and that multi-platform libraries (e.g. Qt or wxWidgets) are fairly complex.

This is where glut (which stands for GL utility toolkit) comes in. It was created as a minimalist, very simple library for providing some OS services to OpenGL:

- create and resize windows;
- handle keyboard events;
- handle mouse events;
- handle timer events;
- create drop menus.

A major advantage of glut is that almost exactly the same code can be built and run on different platforms, with only slight cosmetic difference. What differs between different platforms is the location of the headers and binaries for OpenGL and glut. This can be solved by using a header that defines these paths based on the OS and development environment for which the program is built. Some open-source libraries build on top of glut to offer more advanced GUI widgets such as buttons, sliders, file selectors, etc. An example of such a library is pui, which is distributed as part of the plib library.

One minor disadvantage of glut is that the interfaces built with glut (including the ones built on top of glut) are ugly[1]. However, glut was never meant to build commercial-grade applications, just small demos and programming assignments for courses, so this drawback is fairly minor.

The main drawback of glut is that, on Mac and Windows at least, it is still a 32-bit library, now deprecated. It should be replaced by freeglut, which is theoretically 100% compatible with the original glut, the only difference being the name of the header file and binary[2]. On Linux, freeglut has completely replaced glut, so that nothing has to be changed in the code nor in the paths.

[1] This may not be obvious to Linux users, because they don't particularly stand out as being significantly worse than other Linux apps, but on Mac or Windows, the difference is striking.

[2] This being said, in recent versions of mac OS, i have had serious problems with freeglut, to the point that I have decided to revert to Apple's deprecated version of GLUT).

2.3 Install MESA and FreeGlut on Ubuntu

MESA is a open-source implementation of OpenGL, the only free option on Linux. MESA and FreeGLUT come installed by default on Ubuntu, so that you can run pre-built OpenGL, but in order to build a MESA project, you need to install the *developer* version of both libraries.
Make sure first that your Ubuntu install is up to date, and then execute:

- `sudo apt-get install freeglut3-dev`
- `sudo apt-get install mesa-common-dev`

2.4 Build and run the handout

After you have installed MESA and FreeGlut, you should be able to build the handout:

```
g++ -Wall main.cpp gl_frontEnd.pp -lGL -lglut -o cell
```

Once you start adding threading code to the handout, you will need to link with the pthread library. So, the build command will become

```
g++ -Wall main.cpp gl_frontEnd.pp -lGL -lglut -lpthread -o cell
```

You may also include headers that are part of the math library. In that case you would execute

```
g++ -Wall main.cpp gl_frontEnd.pp -lm -lGL -lglut -lpthread -o cell
```

Of course, feel free to change the name of the executable. What you cannot change much, though, is the order in which you list the libraries to load, if you want to avoid linker errors[3].

2.5 The GUI

When the program runs, you should see a window with two panes. The one on the left displays a grid with white dots switching on and off. This dots corresponds to the different cells of a cellular automaton and the back/white color corresponds to the alive/dead state of each cell. The right pane displays some text messages, such as the number of "live" computing threads.

There are a few keyboard commands enabled:

- Hitting the 'escape' key terminates the program;
- Hitting the + and − keys speeds up or slows down the simulation;
- Hitting the spacebar "resets" the grid's cells to a random state;
- Hitting `1`, `2`, or `3` changes the rule used by the cellular automaton;
- Hitting `c` or `b` toggles on/off the color mode;
- Hitting `l` toggles on/off the drawing of lines delimiting the rows and columns of the grid (only useful for smaller dimensions).

[3]The handout builds fine on mac OS, with clang++ instead of g++, but OpenGL and glut are *frameworks*, not just regular libraries, on mac OS, so you need to do `clang++ -Wall -std=c++17 -stdlib=libc++ main.cpp gl_frontEnd.pp -lm -framework OpenGL -framework GLUT -lpthread -o cell`. I will trust Chris and Mikel to give you proper directives for building on Windows.

3 Part II: The Cellular Automaton

3.1 Cellular automata

3.1.1 Concept

Cellular automata (CA for short) were developed as a model for the way cells or organisms develop and die. The basic idea is that if you are all alone in a desert you die, and if you are too crowded you die as well. In other case you can keep going or, even better, prosper and create new life.

The most famous cellular automaton is the Game of Life of British mathematician John Horton Conway:

`http://zh-yue.wikipedia.org/wiki/%E7%94%9F%E5%91%BD%E6%A3%8B`
`http://www.conwaylife.com/wiki/Cellular_automaton`

The reason why cellular automata are so interesting for mathematicians (and programmers) is that with extremely simple rules it is possible to produce very complex dynamic behaviors such as patterns that repeat over time (called "oscillators"), "glider" patterns that move over the screen (Figure 1), etc. Some of these patterns can be quite large, like the ones nicknamed "spaceships."

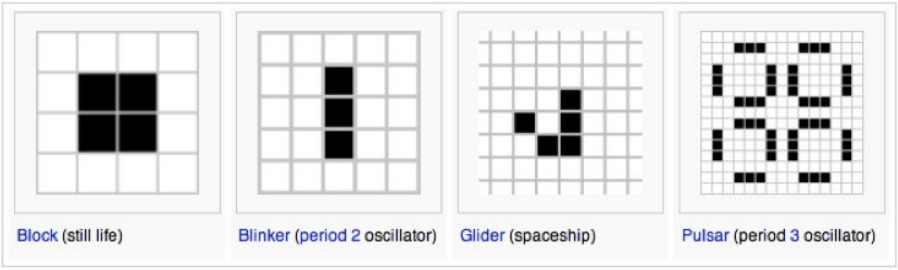

Figure 1: Examples of repeating patterns in the Game of Life (from Wikipedia)

3.1.2 How it works

In their simplest form, cellular automata are represented as rectangular grids of "cells" that are either dead or alive. At each new generation of the cellular automaton, each cell looks at all of its immediate numbers and counts how many of them are alive. A live cell that has too few live neighbors will die (desert). So will a live cell with too many live neighbors (overcrowding). Otherwise a live cell will stay alive. Life can restart at a "dead" location if the right number of live neighbors are present.

Different cellular automata differ in the rules that specify under what conditions a cell will die or come to life. The rules for CAs are generally presented in the form Bxxx/Syyy. For example, the classical Game of Life is B3/S23. What does this mean?

- A dead cell with exactly 3 live neighbors will get **B**orn again (alive).
- A cell that is alive will **S**tay alive if it has either 2 or 3 neighbors alive (if it has 0, 1, 4, 5, 6, 7, or 8 neighbors alive, then the cell will die).

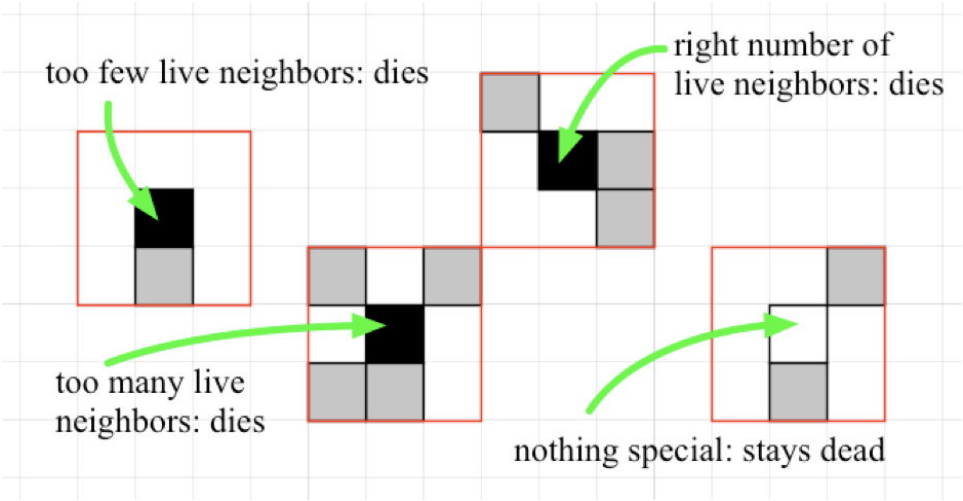

Figure 2: How cells live or die in the Game of Life.

Figure 2 shows some examples of cell birth/death. In the figure I have drawn "alive" cells in black or gray, and "dead" cells in white. We will do the reverse in our application because it will look prettier when we start using color to represent the "age" of live cells.

Note: The basic CA have only two states for their cells, but it is of course possible to define more states, and rules that take this into account. For example, the CA known as "Brian's Brain" has cells that can be "alive," "dead," or "dying" (a cell does not go directly from "alive" to "dead" without one generation in the state "dying," which is a form of "second chance" algorithm).

3.2 Let's look at the code

You shouldn't need to modify anything in `gl_frontEnd.cpp` or `gl_frontEnd.h`. Even in `main.cpp`, some parts are marked as "no need to touch," or even "don't touch!"

3.2.1 The "glut main loop"

You need to understand a few things about glut before you start doing things that will break the program. In an OpenGL+glut program, we define a bunch of "callback functions" that will be called directly by glut to process some GUI-related events (e.g. timer event, mouse clicked event, window resizing event, etc.). Once the callback functions have been defined (I do that in `gl_frontEnd.cpp` and you shouldn't need to modify it), the program relinquishes all control to glut with the call to `glutMainLoop()`.
In a multi-threaded application, glut insists on being the main thread. This means that you cannot create a thread and have that thread make the call to `glutMainLoop()`. Your main thread creates other threads and then it is the one that makes the call to `glutMainLoop()`.

3.2.2 Why the cellular automaton "works" in the handout

The code of the handout include an ugly hack that you will **absolutely** need to remove once you add threading to your code. In order to get the cellular automaton to keep updating, I make a call to `oneGeneration()` from within my timer callback function. Once you properly implement your threads, that call to `oneGeneration()` must be removed. Otherwise you will get all sorts of strange behaviors.

3.3 Version 1: Multithreaded CA with short-lived threads

3.3.1 Modify the main function to take arguments

In the handout, the number of rows and columns of the grid are defined as constants. You should modify the code so that the program now takes three arguments: the width (number of columns) and height (number of rows) of the grid, as well as the desired number of computation threads of the program. You should verify that your program indeed received three arguments, that these are strictly positive integers, with the width and height larger than 5, and than the number of threads is not larger than the height of the grid.

Style-wise, this means that the constants `NUM_ROWS` , `NUM_COLS`, and `MAX_NUM_THREADS` won't be constants anymore and should be renamed, from the current all-caps-separated-by-underscores forms to lowercase with internal capitalization, since they are going to become regular variables.

3.3.2 Multithread the program

The simplest way to do this is to assign a horizontal band of the grid to each thread, which means that you will have to assign a "start row" and "end row" to each of your threads. Make sure that you split the work as evenly as possible and don't miss rows.

In this version, your threads are going to last for a single generation. At each generation of the CA, you create the update threads, and you join them at the end of the generation. Although this is a bit clumsy, this solution has two advantages. First, it keeps your threads synchronized, since there is no way for a thread to get ahead of the others by working on the next generation before its "siblings" have completed their work. Second, because you are going to do a lot of allocating and freeing, this will be a good stress test for the program, as memory leaks will be exposed.

3.3.3 One last thing: speedup and slowdown

Finally, for this version to be complete, you need to figure out a way to enable the speedup and slowdown key controls. This does not mean adjusting the delay of the callback to the timer function (which should be set to a value around 10 to 30 milliseconds. Rather, you want to adjust the sleep time between generations.

3.4 Version 2: Multithreaded CA with a single mutex lock

3.4.1 "Permanent" threads

In this version, we want to create our update threads only once, and then let them keep updating their part of the CA until the program terminates. This is going to pose two problems that you will

need to address and fix.

3.4.2 Problem 1: Keep the threads in sync.

Now that you are going to have multiple computation threads that you don't join between generations, how are you going to know that the "next grid" has been completely computed and that we can now safely swap the grids and launch the computation of the next generation? I will let you ponder this one on your own and come up with your own solution.

3.4.3 Problem 2: Terminate properly

One problem with the multithreaded version of the CA is that when the main thread decided to exit, there is a good chance that you will get a bad signal from some of the threads. What you want to do here is that you want to make sure that all your thread have stopped messing with the grid before the program terminates.

3.5 Version 3: Randomized multithreaded CA with a grid of mutex locks

3.5.1 An in-place implementation

This proceeds of the same idea of the randomized bubble-sort implementation that we saw in class. Here we are saying: Instead of a dumb execution in artificial strict order, row-by-row and column-by-column, why not let the update be more natural and random? In this version, each of the computation threads would randomly select a cell to update and would perform the update *in place*. In other words, there wouldn't be anymore a "next grid," as all calculations would be done directly on the "current grid." This also means that all cells won't be anymore all at the same generation. Some cells will "age" faster than others if they are lucky, or unlucky, depending how you see it.

3.5.2 Synchronization issues

In this new, randomized and in-place version of the cellular automaton algorithm, we now have a race condition on all the cells of the grid. We could simply use the global lock that was used to synchronize shared access with the rendering thread, but this would be boring.

Instead, we are going to use a 2D array of mutex locks, one for each cell of the grid (so, essentially, we replace one grid, the "next grid," by another: the array of mutex locks). In order to update the sate of one cell, the computing thread will need to acquire the nine locks of the 3×3 square centered at that cell.

It is really important that these locks are always acquired in the same order[4]. Here, because all threads run the same code, this should not be an issue. Let's say that you acquire the locks in order to-to-bottom and left-to-right.

Now, synchronizing access to the entire grid with the rendering thread is even more critical than before. Here again, I let you think about it an propose your own solution. I will just give you a hint: Does this look somewhat like one of the classic problems?

[4] You will see why when we rush over the chapter on Deadlock.

3.6 Extra credit?

I can't think of any doable extra credit at this point. If I can think of some, I will post them directly on the assignment page. In any case, I will give extra credit for any cool idea of very nice implementation of the standard assignment.

4 Part II: `bash` Script(s)

4.1 Build script

The script `build.sh` builds all the versions of the assignment that you completed.

4.2 Control your program from a bash script (extra credit: 5 points)

The program of the handout uses a very rudimentary GUI to specify some of the parameters of the cellular automaton. Using keystrokes, the user can select the cell reproduction rule, speedup or slowdown the simulation, switch the display from black and white to color, and also terminate execution.

What I want you do is to allow the user to perform the same adjustments to the cellular automaton by giving commands to the script. The purpose of this task in the assignment is not to improve the user experience for this particular program but simply to add one more tool to your toolbox, even if the task we use it for here is borderline silly/useless.

4.2.1 What `script1.sh` should do

The script should (not necessarily in this order):

- Get as arguments the width and height, and number of threads of the cellular automaton process to launch;
- Setup a named pipe to communicate with the cellular automaton process, using the command syntax defined in the next subsection;
- Launch a cellular automaton process, specifying the desired width and height;
- Communicate with the CA, sending commands grabbed from the standard input.

4.2.2 Syntax of the commands

After the script has launched the cellular automaton process, it should take commands from the user (and send these commands to the process using a named pipe). The syntax of these commands is:

- `rule` *¡rule number¿* selects the automaton's cell reproduction rule. For example, `rule 2` selects the "coral growth" rule.
- `color on` enables the color mode;
- `color off` disables the color mode (reverts to black and white);

- `speedup` increases the speed of the simulation;
- `slowdown` reduces the speed of the simulation;
- `end` terminates execution of the cellular automaton of the process and of the script.

Invalid commands (including rule selection commands with an invalid rule number) should be properly rejected.

4.3 Pure command interpreter (extra credit: up to 8/12 points)

Note: If you completed `script1.sh`,then `script2.sh` can bring up to 8 more points. If you skip `script1.sh` and complete directly `script2.sh`, then it is worth up to 12 points.

Modify your script so that it becomes a pure command interpreter and even the launching of a cellular automaton process is done via a script command. This means that your script could now launch and control multiple cellular automata. The command to launch a new cellular automaton would be

- `cell` *width height*

Each new cellular automation process would receive an index (starting with the first process at an index of 1) and display that index in its window's title.

All other commands to be interpreted by the script should now be prefixed by the index of the cellular automaton process to which they should be addressed:

- `3: speedup` means that Cellular automaton 3 should increase its speed;
- `2: end` means that Cellular automaton 2 should terminate execution;
- etc.

Invalid commands, including commands meant for a process that hasn't been created or one that has already terminated, should be rejected.

5 What to Hand In

5.1 Organization of the submission

Create a new folder named `Prog06`. In that folder, place

- The `Scripts` folder
- subfolders named `Version1`, `Version2`, and `Version3` that contains complete source and header files for the different versions that you implemented;
- your report as a `.pdf` file.

5.2 Grading

- The C program performs the task specified (execution): 50 points
  - Version 1: 20 points
  - Version 2: 20 points
  - Version 3: 10 points
- C++ and code quality (readability, indentation, identifiers, etc.): 20 points
- Building script: 5 points
- Report: 20 points
- Folder organization: 5 points