CSC 412 – Operating Systems

Final Project, Spring 2019

Wednesday, May 1st 2019

Due date: Friday, May 10th 2019, 10:00pm (**submissions accepted until May 13th, 11:55pm).**

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,
- a bit of deadlock management.

1.2 Handouts

The handout for this assignment is the archive of a C++ project made up of 2 source files and 2 header files.

1.3 Solo or group project

This project was designed to be done by a group of two or three students. You may try to do it solo, but highly discourage this. In fact, I *actively* discourage it by making some extra credit sections mandatory for solo developers (in the spirit of "since you think you're so good, then why don't you do that too?").

2 Part I: The Basic Problem

2.1 The world

For this problem, you are going to simulate travelers in a maze searching the single exit point. Figure 1 shows what the environment looks like. The travelers are the wiggly colored lines. their maze is made up of solid walls (drawn in brown color) and of sliding partitions drawn in gray. The single exit point on the maze is indicated by the white square with a black cross over it.

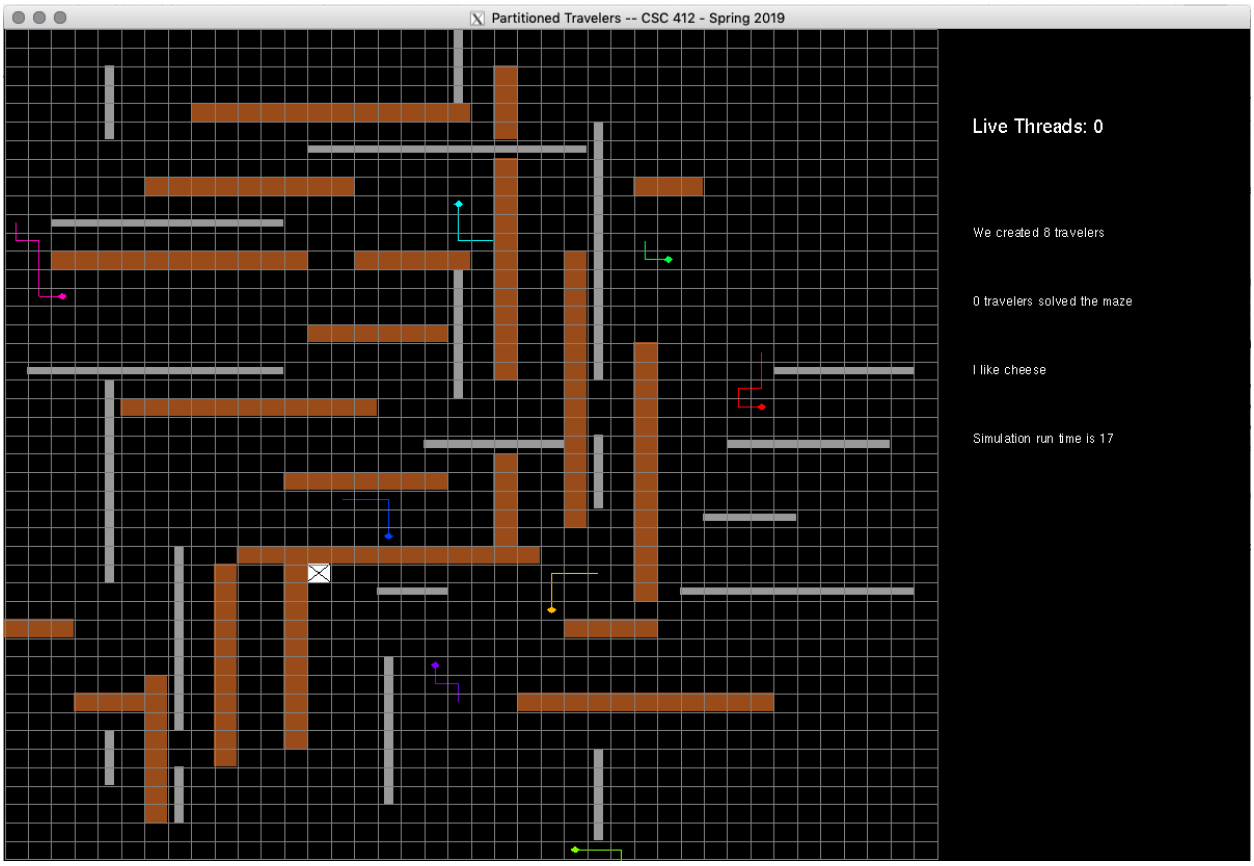

Figure 1: A screenshot of the graphical front end.

2.2 The task

The task is the same for all travelers: Make it to the exit point, and terminate their execution.

2.3 How travelers move

Travelers can only move vertically or horizontally (no diagonal move for this assignment). Upon reaching the exit, the traveler simply leaves the simulation. If a traveler can make it to the exit simply by walking around the various obstacle, then that traveler had an easy problem (that traveler still has to handle a synchronization problem over access to the grid squares).

In some situations, however, your traveler may need to move a sliding partition out of the way. Partitions only move along their main direction. Needless to say, you will need to handle some synchronization problems when you slide a partition.

Travelers start their life a single-segment creatures, but grow segments as they travel. After it has made N moves (N is an input parameter of the program), a traveler will grow a new segment. You have to make sure that your travelers don't self-intersect.

2.4 How to move the partitions

This part is completely up to you. The grid only stores the type of object found on the grid. So, if you find that the square at (row, col) is occupied by a vertical partition, you will need to search the partition list to determine which partition it is. After that, moving the partition will involve updating the data for this particular object as well as the state of the grid.

2.5 Input parameters of the program

Your program should take as parameters from the command line:

- the width and height N of the grid,
- the number n of travelers,
- the number of N moves after which a traveler should grow a new segment. If this parameter is not entered, then it will get a default value of `INT_MAX` (meaning that travelers will stay single-segmented).

and then randomly generate a random location for each traveler (you may of course reuse the code of my handout of this purpose).

3 Implementation

3.1 Version 1: Single-threaded version, no sliding partitions

I don't want you to rush trying to implement directly the full problem, because too many of you will end up with a mess of nonworking code. Therefore, I am going to *impose* some intermediate version, to make sure that you have the basic part of the program working before you add in multiple threads and mutex locks.

In this version, you are going to run with a single traveler, and try to get your traveler to the exit. Make sure that your path planning works and that your traveler avoids the walls and partitions, but doesn't attempt to move the partitions. You will also need to make sure that your code doesn't conflict with the glut callback functions so that the display refreshes properly.

3.2 Version 2: Multithreaded, with no synchronization

In this version, you are going to create multiple travelers, each running on a separate thread. Our travelers still don't move the sliding partitions, and we allow the travelers to run over each other, so there is no need for synchronization yet.

Some of you may be tempted to use a fancy path planning algorithm (say, A*) to plan a path for the travelers. Obviously, it is possible to use such a planner, but keep in mind that in this problem (not in this version, but in the next ones), the environment will change while the simulation is running, so you will need to do dynamic replanning if you find that your initial plan is not valid anymore.

As was the case in Assignment 6, if you want to join a traveler thread in the main thread, for whatever reason, you can only do that after the `glutMainLoop()` call at the end of the main function. The reason for this is that in a GLUT-driven program, the handling of all events and interrupts is left over to GLUT. If you insert a `pthread_join` call (basically, a blocking call) anywhere, you basically block the graphic front end. Keyboard events won't be handled anymore and no more rendering will occur. There is no way around that as GLUT *must* run on the main thread. It is a the price to pay for it being such a light-weight, portable, and easy to use library.

3.3 Version 3: Multithreaded, with a single lock

This version simply add synchronization over the location of the travelers, both in their data structure and on the grid. Whatever piece of code in the program needs to access either data structure needs to get the lock first and release it after it has accessed the data. Yes, this also applies to the rendering code. Try to keep the critical section as short as possible.

3.4 Version 4: Enable partition sliding and multiple locks

Now that you have your threads moving around without running over each other, we can enable partition sliding. In addition, we are going to add one lock for each traveler, to access that traveler's data. These new locks come in addition to the global lock used to control access to the grid.

3.5 Version 5: Synchronization with multiple gridlocks

In this final version, you need to implement one lock per row and one lock per column for the grid (not one row per square).

4 Extra Credit

4.1 Progressive disappearance (4 pts)

Mandatory for solo developers
When a traveler's "head" reaches the exit, the traveler doesn't disappear immediately, but one segment at a time.

4.2 Extra credit (10 pts): Detect deadlocks

To implement this one, you will need additional data structures and a bit of journaling of the actions of your travelers.

4.3 Extra credit (5 pts): Deadlock resolution

Mandatory for solo developers
I don't expect you to implement a complete, working solution for the deadlock problem, but rather a discussion of the problem and an outline of how you would solve it.

4.4 Extra credit (20 pts): Deadlock resolution implementation

I don't really expect to see that one, but I give the option to try.

5 What to Hand in

5.1 Source code

The source code should be properly commented, with consistent indentation, proper and consistent identifiers for your variables, functions, and data types, etc.

5.2 Note on the report

In the report, you will have to document and justify your design and implementation decisions, list current limitations of your program, and detail whatever difficulties you may have run into.

5.2.1 Limitations of the program

It is important to be able to identify things that your program cannot do, or cannot do well, and it's crucial that you identify situations that can make it crash. Even if you lack the time to solve these issues, it is important to identify problems. Think of it as a sort of "todo" list for future revisions.

5.2.2 Difficulties you ran into

By this I don't mean "my car was for repair this week." Rather, I mean coding problems that forced you to think of some clever solution.

6 Grading

- Version 1 completed and performs as required: 10%
- Version 2 completed and performs as required: 10%
- Version 3 completed and performs as required: 20%
- Version 4 completed and performs as required: 10%
- Version 5 completed and performs as required: 10%
- good code design: 10%
- comments: 10%
- readability of the code: 10%
- report: 10%