CSC 412 – Operating Systems

Final Project, Fall 2023

Wednesday, December 6th 2023

Deadlines — Team announcement: Tuesday, December 12th, 6pm
Midpoint version: Sunday, December 17th, 2023
Due date (for legal reasons): Wednesday, December 20th, 2023
Submissions accepted with no penalty until: Saturday, December 23rd, 11:59pm.

1 What this Assignment is About

1.1 Objectives

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

- threads,
- 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 three source files and two header files.

1.3 What you should stay clear from

I have marked spots that you should not mess with. Generally, you shouldn't have any reason to modify the rendering code. I believe that this code is a bit simpler than that of the last assignment, in the sense that I have tried to move to `main.cpp` the functions that you may want to edit. So, you shouldn't have to do much, if anything at all, in `gl_frontEnd.cpp`.

I implemented the `Traveler` type as a C-style struct (of course, in C++, this kind of struct is just a class with all its members public and implicit default semantics for constructors, destructor, and copy/move operators. If you want to do some OOP and implement the `Traveler` type as a more complete struct or class with a real constructor and some instance functions, feel free to do so, but this is not one of the objectives of this assignment.

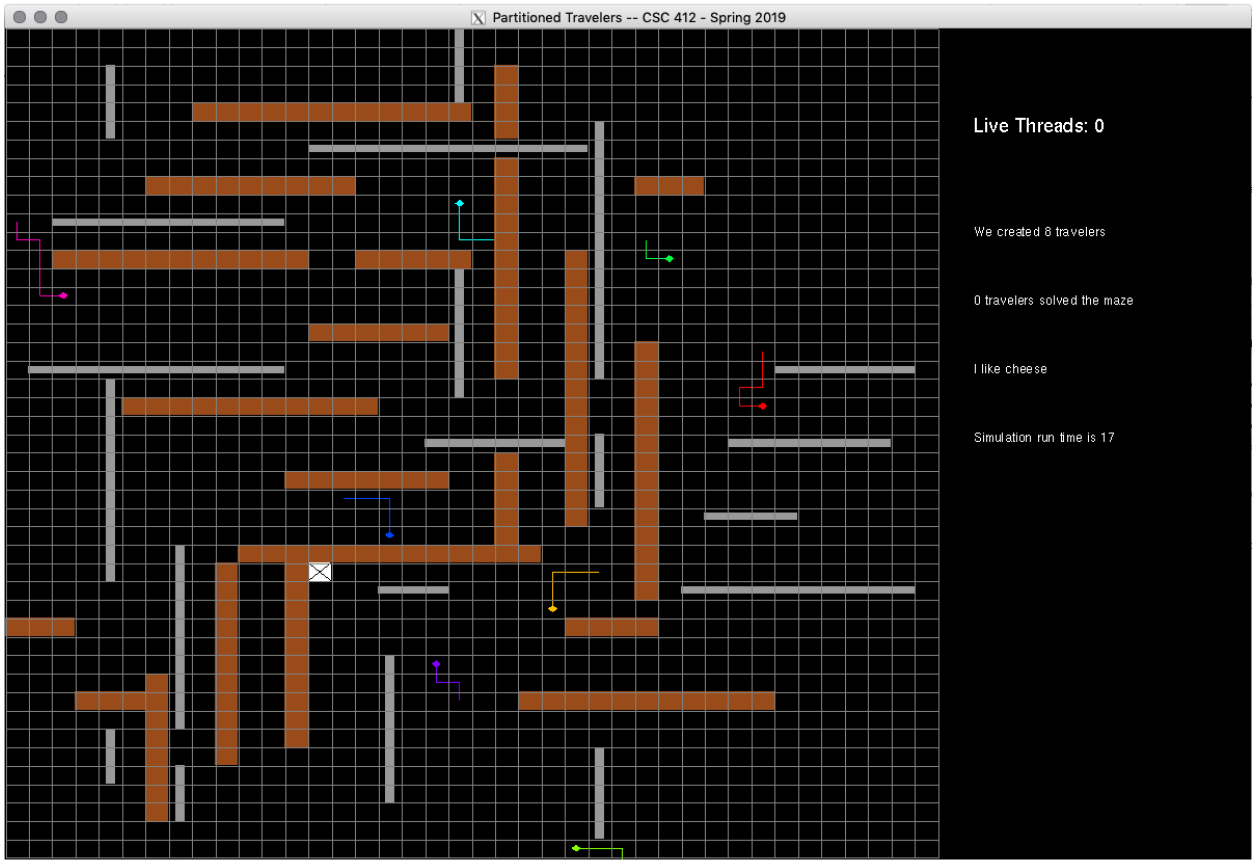
1.4 Solo or group project

This project was designed to be done by a group of two or three students. Groups of three will simply have to complete a bit more work (things that could be done for Extra Credit become mandatory). You can work alone, but the requirements will remain the same as for a group of two.

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.



```
Live Threads: 0

We created 8 travelers

0 travelers solved the maze

I like cheese

Simulation run time is 17
```

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, a 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.

2.3.1 Tail growth

Travelers 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.

2.3.2 Traveler tails and motion direction

If you look at the `Traveler` data type, you will see that it stores a vector of `TravelerSegment` structs. A `TravelerSegment` in turn is defined by its location on the grid and a direction N/S/E/W in which it points.

In Figure 2, I show an example of a multi-segmented traveler (8 segments), and I indicate the value of the `dir` field for each segment. The "head" of my traveler (indicated by the larger red disk) is pointing to the EAST direction.

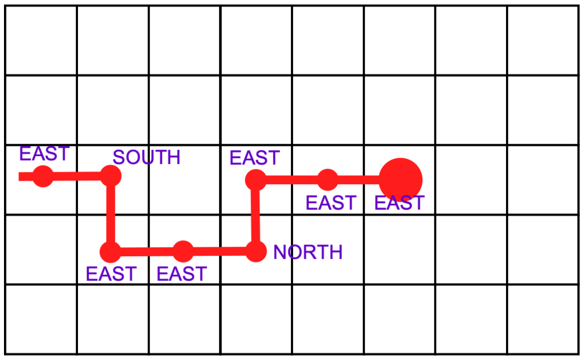


Figure 2: A multi-segmented traveler.

Now let's say that my traveler moves in the direction in which its head is pointing (EAST). Then after its move it would look as shown in the top-right of Figure 3. If this move just happens to coincide with a time where the traveler grows a new segment, then it would look as shown in the bottom-right view.

If the traveler changes direction, first it cannot inverse the direction in which it's headed. So, our heading-EAST traveler cannot move in the WEST direction. We have already seen it move EAST. The other two possible directions would be NORTH or SOUTH. Figure 4 shows what

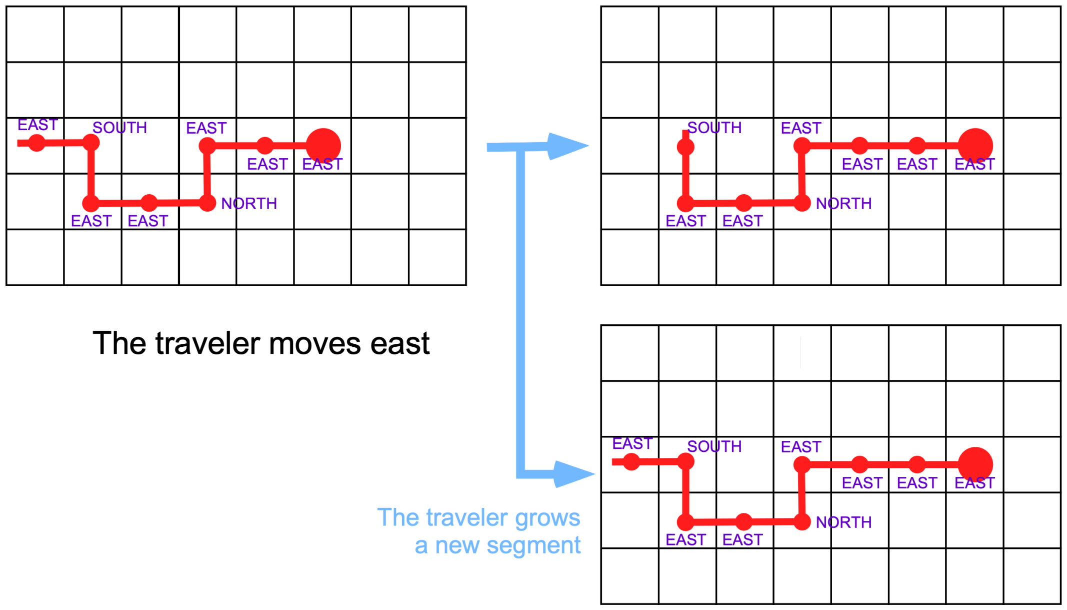


Figure 3: The same traveler after one “EAST” move.

happens in the case of a move to the NORTH, again both in the case of a regular move (no segment growth) and of a move with segment growth.

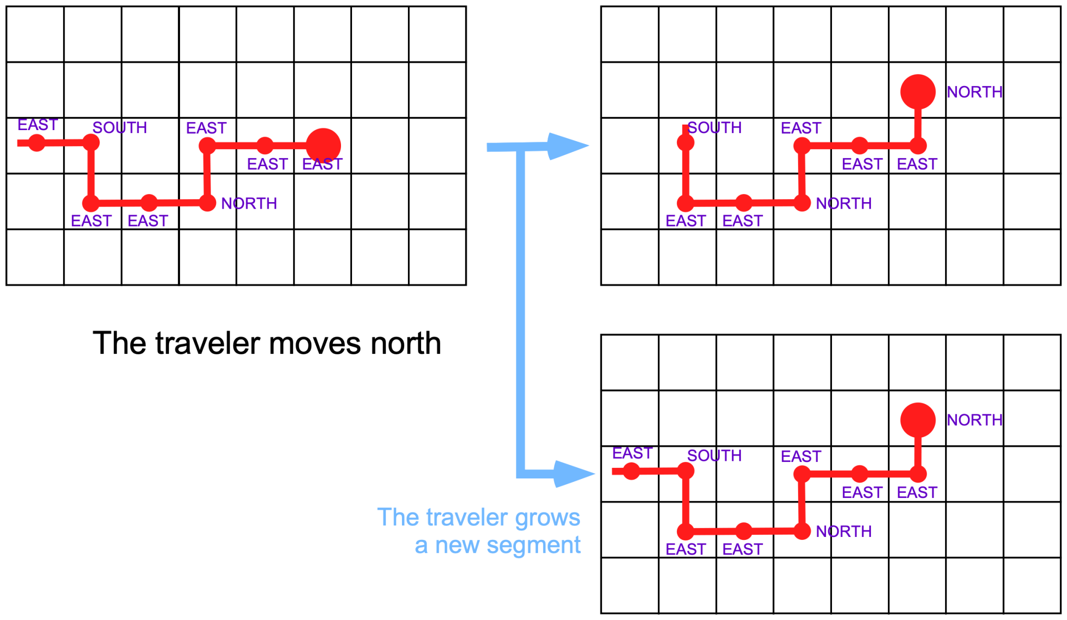


Figure 4: The same traveler after a direction change and one “NORTH” move.

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 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

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.

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

In this version, you are going to run with a single traveler thread, 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 define and use one lock per grid square..

3.6 Additional work for groups of three

Some of the work marked as "extra credit" for solo or pair work not only does not bring any additional points but is actually **mandatory** for groups of three. These pieces of additional work are:

- Subsection 4.1: Progressive disappearance.
- Subsection 4.2: Deadlock detection strategy.
- Subsection 4.9: Multiple levels.

3.7 A note on implementation options

If you look at the code, you will see that it looks very similar to that of the last programming assignment, but with a slightly different arrangement. You will also see that you have different options for handling displacements of travelers and partitions. For both kinds of objects, there is a grid square type (`TRAVELER`, `VERTICAL_PARTITION`, `HORIZONTAL_PARTITION`), and then there is also a struct data type (`Traveler` and `SlidingPartition` respectively).

This means that we are dealing with redundant representation of information. You don't *have to* update all these representations of the sate of the state of travelers and partitions, and you are free to use whichever you want to check for collisions in the workspace: Do you check segments of the traveler against grid squares occupied by a partition or against the list of blocks that make up a partition? Do you use grid squares with value `TRAVELER` instead? This is completely up to you. Just keep in mind that the rendering uses:

- grid square values, not the `SlidingPartition` structs, for the partitions,
- the lists of `TravelerSegment` of `Traveler` structs, not the state of the grd squares, to render the travelers.

Unless you decide to tackle replacing the rendering code, you must therefore update the information necessary for the rendering code of the handout. But this doesn't mean that you can't decide to use grid square values to check for and prevent traveler-traveler collisions, or the `SlidingPartition` structs to move your partitions. Again, I provide options. Pick the one that works best for you.

4 Extra Credit

4.1 Progressive disappearance (4 pts)

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

4.2 Extra credit (4 pts): Deadlock detection strategy

Mandatory for groups of three

To get full points for this EC section, you would need to provide a thorough discussion of the conditions that could lead to a deadlock (please note that a traveler that wrapped itself into a coil and cannot move anymore is not really engaged in a deadlock but in an AI failure) and what you would need to add to your program to successfully detect such deadlocks.

4.3 Extra credit (6 pts): Deadlock detection implementation

Having collected the EC points on the previous item (unless you are in a group of three), implement your solution and assess success 9or absence thereof).

i am still going to give most of the EC points if your strategy seemed reasonable but you observe that it doesn't actually solve the problem, and then make a proper assessment of what went wrong and what you could have done differently, with more time.

4.4 Extra credit (5 pts): Deadlock resolution strategy

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.

4.5 Extra credit (15 pts): Deadlock resolution implementation

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

4.6 Extra credit (3 pts): Take control of a traveler

The user of the program should be able to use the usual `w-a-s-d` keys to control the displacements of traveler thread 0. That thread would still be submitted to the same synchronization constraints as the regular "autonomous" threads (which implies that these EC points are only valid for a version of the project of 3 or higher).

4.7 Extra credit (3 pts): Take control of any traveler

In this version, the user of the program can hit the index (0-9) of a thread to take control of the corresponding traveler, without waiting for the traveler previously controlled to make it to the exit. When the user selects a new traveler to control, the traveler that was user-controlled before that should revert to autonomous mode. Here again, these EC points are only valid for a version of the project of 13 or higher.

4.8 Extra credit (5 pts): Collision-activated sliding partitions

When a user-controlled traveler collides with a sliding partition, the partition should slide in one of the two directions it is allowed to (random selection).

4.9 Extra credit (8 pts): Multiple levels

Mandatory for groups of three

In this version, the travelers don't terminate anymore when they find the exit and "go to sleep" instead. When all travelers have found the exit, the game doesn't end. Instead, a new maze and new locations for the travelers are generated, after which the travelers are awaken and the game restarts.

4.10 Extra credit (up to 8 pts): improved AI

The focus of this assignment is the proper synchronization of access to grid and Traveler data, *not* the planning of a path leading each traveler to the exit. In other words, as long as your travelers don't pass through walls, partitions, or each other, it doesn't matter if they end up stuck in a corner.

This being said, if you want to spend a bit of time giving more "brains" to your traveler threads by improving their planning, I will give some extra points for that. Full EC would be for an AI that always finds a path when one is possible.

5 What to Hand in

You should make only one submission per group. 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[1], so make sure that you know who is tasked with submitting the final version.

Your submission should include:

- A report identifying all team members.
- Source code: Each version should be presented in a separate folder named `Version1`, `Version2`, etc.
- 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.

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.

Since most of the code is handout, I am not asking for Doxygen comments nor Doxygen-produced documentation.t

5.2 Note on the report

In the report, you will have to document and justify your design and implementation decisions, in particular regarding threads and synchronization, 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: 15%

[1]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 wasa mistake, and the grading time on the first project was a complete waste.

- Version 4 completed and performs as required: 13%
- Version 5 completed and performs as required: 12%
- good code design: 10%
- comments: 10%
- readability of the code: 10%
- report: 10%