Final Project

CSC 412: Networks and Operating Systems Fall 2023

Aaron Bun and Andrew Dionizio

Version 1 – Version 5
Data Structure / Algorithms:

- Traveler Movement
  - In order to determine where the traveler moved, we created a separate function called getAvailableDirections(). This function receives the head of the traveler, it will then insert the current square into a set of grid squares and mark the current square as visited.
  - The algorithm used in this function will explore up, down, right, and left of the current traveler head and determine which directions are valid to move to. Valid squares are determined if the explored direction on the grid is either a free space, an exit, or partitions in later version of the code. This function will return a vector of all possible directions.
  - We then choose a random direction from the vector of all possible directions. After we are choosing a direction, the traveler will attempt to move to that square.
  - We've noticed only the head of the traveler moves to a new space; thus we move the head of the traveler last after changing all the segments according to the algorithm below. When we actually move the head of the traveler in the handleObstacleCase(), we determine what kind of square we move to.
    - If a free space is chosen, we set that space to a traveler space on the grid.
    - If an exit space is chosen, we set the new segment to that space.
  - After attempting to move to the space, the traveler will finally change the segmentList[0] to the new segment determined in the handleObstacleCase()
- Travelers Segment Algorithm

```
-   For each segment for a traveler[i]:
-     if growSegment:
-       store end segmentList.back()
-       set segment[n] = segment[n-1]
-       move segment[0] in new direction
-       pushback(store end)
-     else
-       set segment[n] = segment[n-1]
-       move segment[0] in new direction
-
-   set segment[0] = newSegment
```

- Partition Movement
  - In version 4 and 5 where partition movement were enabled, a partition was an available direction to move in. However, to determine a partition as a valid direction to move in, we have to determine if a partition is stuck. In addition of creating a parititionList, a partStatus vector was also created containing partitionStatus struct. Each struct contains if a partition was stuck to the left, up, right, or down (these were booleans). It also contains the number of times a partition moved.
  - To determine if a partition was stuck, a separate function was used.
    - Vertical Partition – check if one above a partition is a free space or check one below a partition is a free space.
    - Horizontal Partition – check if one to the left of a partition is a free space or check one to the right of the partition.
  - Anything that wasn't a free space makes that specific partition stuck in that specified direction.
  - When actually moving the partition, we flipped a coin to determine to move up or down if a vertical partition, or left or right for a horizontal partition. We then move each block in the partition by one in the specified direction.
- Traveler Getting Stuck
  - There are several conditions which allows a traveler to get stuck. Since a traveler moves randomly, there is a chance for a traveler to trap itself in its own segments. To handle this specific case, we set the keepGoing condition for the thread to false and the traveler will terminate.
  - Other conditions allow the traveler to get stuck, for example, this may include a traveler going down a one-way path and having no way to get it. So, the thread will terminate like mentioned before.
  - Traveler collision can happen within our code. If two travelers happen to crash into each other head on and there are no other available directions, both of the travelers will terminate.
  - When travelers collide with each other in a way where there are no possible direction to move in for one traveler, we temporarily make the traveler go to sleep. We give a traveler five attempts to find an available direction. This allows enough time for another traveler to move out of the way and allow the original traveler to search for directions and move again. In no available direction appear within the five attempts, the traveler will terminate.

Synchronization (Version 3 - 5)

Version 3 (single lock) : Version 3 was a very simplified version of synchronization. Due to only having one lock for all the travelers and the grid, only when accessing the data, we lock and unlock when we are done accessing the data. Specifically for our implementation, the lock was obtained whenever the traveler segments were modified like in the traveler segment algorithm mentioned above. Additionally, any spot on the grid which gets physically changed, we lock and unlock the data after the grid square changed. For example, this may be when a traveler segment leaves that grid square and then changes that spot to a free space.

Version 4 (Traveler Locks) : This version implements one lock per traveler and one lock for the grid. Like the previous version we keep the lock locations the same, however, any time we change a grid square, we lock and unlock the global defined grid lock. Any time a traveler changes its data, like moving its segments, we lock the traveler lock and unlock when done. These two locks ensure the protection for a specified traveler and no other thread can overwrite their data. However, the traveler locks are another defense of protections. We designed our code already to choose directions that are not in the direction of another traveler. So, there are no instances where one traveler will run over another traveler. This means no other traveler will overwrite another traveler's segment data.

Version 5 (Square Locks) : In version 5, a lock was created for every grid square. In this version we also decided to keep the traveler locks to prevent any overwritten data just in case. Specifically, we choose to keep every lock for each square initially unlocked. We thought this would be okay because when generating the grid and the travelers, everything is frozen. We only want to lock and unlock grid squares when a traveler is moving to and from other squares. At the beginning of every thread function, we lock all positions on the grid in which a traveler segments occupy. We calculate the available directions and update all the segments in the segment list. After we are done moving a traveler, we then unlock all the locks on the grid which the traveler occupied originally. When the traveler attempts to move again, we lock the square locks for the current state of the traveler's segment .

Deadlocks

Problem: Dead locks were most present in version 5 of our code. After observing the model, we discovered only one occurrence of a deadlock in the program. The technique we used to lock and unlock the grid square locks involved locking all grid squares that a traveler occupied. We would perform our calculations to move the traveler (we also lock the traveler so it can change it segments). After moving the traveler completely, we unlock the traveler. When the traveler looks for its next destination, it locks and reperforms its calculations. However, because we are locking at the beginning of the threaded function and then locking when its done moving, we never consider the new segment square that's determined during moving the head of the traveler. So technically, the head of the traveler is moved outside the locked area. Thus, this shared square outside the locked region can possibly be taken by another traveler. However, during testing, this occurrence almost never happened.

Detection : In order to detect this kind of deadlock, one simple strategy is to check the traveler index of who moved to that certain shared square. In would make sense if a traveler moved to its own square it choose to move to. If we performed a check in which another traveler index occupied the square the current traveler was trying to move to, we can print a statement that says a dead lock occurred at that square and exit the program.

Possible Solution: Since another traveler can possibly take the space of another traveler, a possible solution is to avoid the shared square in the first place. When determining the new segment for the head of a traveler, it would probably be best to do an additional search to check the squares that are around the shared space to determine if a locked area(s) are around that shared square two travelers are trying to move to. We can instead halt the traveler and have the other traveler take the space, or choose another direction to move in if possible and perform the same check.

Problems/Limitations

Background : When testing the movement and interaction of partitions with the traveler, we found a design flaw within our partition design. A partition only becomes an available direction to move to if only one of the directions is not stuck. So, if a partition can move up, but it can't move down, it's a valid partition the traveler can interact with. However, due to the AI of the traveler, the traveler can trap itself within its own segments. Additionally, if the traveler happens to trap itself within its own segments and the only available direction is the direction of a partition, we have to terminate the traveler. Furthermore, if the partition can freely move on either side, this can cause an infinite loop of the partition moving side to side. The traveler can only move in that direction, so there's no free space for the traveler to move to.

Limitation/Solution : Thus, we had to give each partition a max threshold of how many times it can move (we set this to 1000). Thus, if a partition hit 1000 moves due to a stuck traveler, it will terminate the traveler that hit it last. Another small limitation we discovered during this, is if a traveler interacts with the partition with at max threshold value, it will terminate the traveler who interacts with the partition last. So technically, any traveler can terminate due to a partition hitting its max number of moves. We thought this would be fine because most runs of this program will never come close to 1000 moves per partition, unless stuck.

Another limitation happens to be with our threads, each thread has a delay of 100 milliseconds between updating the traveler movement. If no delay is added to the threaded function, the traveler will instantly render on the front end. To see the traveler updating and rendering on the front end, the delay is needed.

Version 5 Specific Error (kind of) : Our version 5 is not compatible with the front-end code that renders the locks. We are not sure why this happens, but we assume that it happens due to the way we lock the gird squares once at time for spaces in which the traveler occupies. The front-end code attempts to check locks, so we felt like the front-end locks and never unlocks again.

Difficulties Encountered

Initially, rendering the traveler correctly was a difficult task in version 1. Due to the directions defined when creating the segments are "backwards". When moving the traveler, you would have to move it in the opposite direction of the generated segments. However, I learned that the front-end code renders the direction opposite of the traveler head's movement direction. When rendering the code for the first time, the traveler and its segments were mirrored. To fix this issue, we simply just switched the direction of the head last to render the traveler correctly. This was slightly confusing at first to see the traveler correctly moving, but rendering in the mirrored direction.

A major struggle involved moving the partitions in version 4. When coming up with a design for moving the partitions, we always considered a partition as an available direction to move in. However, we realized that not all partitions can move on the grid. So, after some time, we had to check if a partition was considered stuck in some direction. For instance, if a partition couldn't move up due to the border of the grid, a wall, or anything that wasn't a free space, it was considered stuck in that direction. The answer was to not consider stuck partitions as available directions for the traveler to move to.

Another difficulty encountered during this project was testing the travelers. Whenever we implemented a solution to a specific problem. We would have to run it a bunch of times to see if we actually fixed the problem. It would've been in our favor to create a test traveler in some part of the grid and specifically test its behavior. However, this was never done. If we had just implemented a test traveler, we could've save some time. Our testing methods were not an ideal approach, but it was worth mentioning.

Extra Credit

4.1 Progressive disappearance implemented in version 4.

4.2 Deadlock detection strategies was mentioned in the deadlock section under detection.

4.4 Deadlock resolution strategy was mentioned in the deadlock section under solution.

4.6 Take Control of a traveler implemented in version 4.

4.7 Take control of any traveler implemented in version 4.

4.6 Collision-activated sliding partitions implemented in version 4. Both 4.6 and 4.7 implementations have working sliding partition interaction.

4.9 Multiple Levels implemented in version 4.

Note** Every extra credit version takes advantage of the callback functionality of glutMainLoop. So, all key's pressed events are handled in the handleKeyboardEvent().