Teddy Jackson

Code report Final

I'm going to talk specifically about the last version, since that carries over most of my earlier design decisions. For pathfinding, I used djikstra's pathfinding algorithm. It's efficient, and easy to make happen on a grid using a queue. The only time I change the grid used for pathfinding is in the function which moves partitions. Due to the limited use of this function, I have no specific safeties in place for accessing the data. Other than that, the grid is read only, so there is no risk for travelers accessing it concurrently.

For the actual movement, I have the traveler read the pathfinding grid, and move to the lowest valued adjacent tile. The tiles are valued higher the further they are from the exit, with the exit itself being valued at 0. If the traveller finds that its lowest value neighboring tile is occupied, it will do one of three things. If there is another neighboring tile of equal value which it can move to immediately, it will take that route instead. If there is only one neighbor that is of lower value, then it will (for a number of calls specified by maxStall) wait in place for the spot to be cleared. Usually the spot will be vacated, and the traveler will continue on its way. There are two main cases where this will not happen. Either the traveler is bumping against a different, dead traveler and the path is blocked, or the traveler is blocking itself. In either scenario, once the traveler has waited for the aforementioned number of function calls, it will pick an available direction to move in. It will prioritize moving in the direction it last moved, which I have found to be useful as it can lead travelers to alternate routes. I use the traveler object to communicate between various function calls when it comes to the decision of where it should move. An example of this is the "wouldStall" boolean, which is used to communicate information from a "safeMove" call.

As far as design difficulties, I had one major problem with my pathfinding. For some reason it was only working properly with the first two directions, but not the second two. Instead

it would wait for the full length of the maxStall period and then move once. Not only that, but it was only doing this in specific board states, so only when another traveler was at the corresponding side. This led to a few scenarios where two travelers would be side by side in an opening, and both would be waiting for a spot that was open and uncontested. I spent a while on this one and couldn't figure it out, so instead I implemented a boolean that ticked back and forth every function call. Every call, the first two directions that are accessed swap, so north & east, then south & west, then next call vice versa. Not an elegant solution, but it means that at max, a traveler will be waiting for an open spot for one move max.

Another issue I had briefly is that when I started multithreading, I had already made my travelers avoid each other, so I had to change the behaviour and let them overlap. The issue I had though was that for an hour I was looking into LLVM errors that kept coming up. My bad i guess, my eyes were bigger than my stomach.

For limitations, I had an unexplained segfault on version 5 while testing it, but I was unable to recreate it, which is obviously concerning considering it was a segfault. Other than that, I don't check inputs, so it assumes that you know what you're doing. Also, because of the way things were worded in the assignment handout, it is implied both that travelers should be able to move partitions, and that travelers moving partitions to get to the exit is not an expectation of the code as it is. This means that for version four and five, where it asks for moving partitions, I bound the functionality to spacebar, with the option to change partitions by hitting the ‘m’ key. The way I designed the partition sliding, I intended to make travelers automatically slide partitions, but to be honest I’m really burnt out, so I’m settling for this.

Extra credit on next page.

EXTRA CREDIT:

I have implemented progressive disappearance in all versions of my code.

How would I detect deadlock, and what conditions could lead to deadlock.

Also maybe how to fix it.

The main reason deadlock happens is that two processes lock eachother out of information they need. The major issue here being that each process has information locked down, and needs more. The way I designed my code, I was able to minimize the amount of dependence processes have on multiple pieces of information. When evaluating a move, each thread will at max have possession of one grid lock at a time. This eliminates the opportunity for the most obvious form of deadlock to happen, since processes won't be holding on to a grid, waiting for another. As for how to detect deadlock, create a timer based around locking of each piece of information. Make the locks timer reset every time it unlocks. If it is locked for too long - which given how quickly code this lightweight runs, it wouldn't be long - then you can be confident that you have a deadlock. Additionally if it's the above scenario, so two or more processes locking each other out, then you could find out which threads are blocking each other by tracking the lock holders. You could give one of them priority by forcing the other threads to give up their lock. If you know that the processes will have only 2 locks to acquire, then you could put the first one on a timer, so if the process cant get the second lock in a certain time frame, then it must give up the first lock and try again.