David Shawver & John Parkhurst

CSC 412

Professor Herve

December 21, 2021

**Final Project: Robot Box Pushing**

In this project we were tasked with developing a program that would run multiple threads to send a robot on a course towards a box where it would push the box towards the destination. In addition, we were tasked with creating three different versions of the program. Although we were able to successfully achieve this task, there were obviously many complications in doing so. As it stands, all of the versions of that we were tasked with creating have been created and appear to run with the GUI and without error. We are using Windows Ubuntu 20.04 and g++ 11 to compile. The program both displays the correct GUI output and file output in both versions. We also managed to complete the Scripting Extra Credit.

A summary of the **design choices** is as follows. We will not be going over the details of the path planning functions, please refer to the Doxygen documentation.

* For part 1, we first determined the path planning for all robots, then iterated thorugh each robot printing its moves to file, then sent a combined vector with all of the robot’s moves to the GUI. The GUI then did one move per robot by updating the global arrays prior to sleeping and updating the display until all of the moves had been completed.
* For part 2, each thread computed it’s path first, then iterated through a it’s list of commands and wrote to file while at the same time updating the global location arrays. The GUI was run on the main thread. One thread was split off for each of the robots. Another thread was split off of the main thread in order to join all fo the child threads while the main thread runs. Each cell corresponding to each robot/box pair of the global location arrays had a single mutex lock, and there was also a single mutex lock for writing-to-file.
* Part 3 is the same as part two except we added a mutual exclusion locking display grid locking system in addition to implementing the reader writers solution pattern for reads and writes to the global location arrays for robots and boxes. When moving to a cell on the display grid, each cell in the grid received it’s own mutex lock which needed to be obtained prior to a cell moving to that location. This prevented two cells from occupying the same location on the grid. Prior to attempting to move to this cell though, we had the robots check if it was occupied on the global location arrays first. But in order to do so, it needed to avoid a race condition and needed to acquire a lock. But having each robot acquiring the global array position read/write lock so frequently just to check if a cell was occupied seemed like it would be a problem for performance, so we implementned a reader writers solution. The function that was just checking for occupancy would read the global arrays (and then the thread would busy wait if they were occupied) just as a reader, meaning only one of the readers needed to have the lock. We used a global reader counter to keep track of how many threads were simultaneously reading, and if it went below 1, then the reading/writing lock for that global array cell was released, making it so any thread/robot that determined it needed to write and was waiting for the lock could acquire it. Once it was finished, the next reader or writer would have a turn. Thus robots could serve as both readers and writers, whereas the GUI would only be a reader. In summary, when the robot was a writer we made the robot acquire the writing lock for that robot/box as an individual, when it was a reader we let it pass through and write to the vector as long as there was at least one other reading robot that had the lock. This arrangement seemed to work pretty well, with no performance issues.
* Other matters of design: The commands lists were vectors of pointers to “pairs” of enums. This data structure was chosen so we could easily resize, keep everything in dynamically allocated memory with storage duration across functions, and to prevent having to differentiate different data components. We experimented using a “Robot” namespace, just for fun, and made many of our methods and variables part of the “Rthread” struct. This seemed to make certain data more accessible when we needed it. We generally designed the program around the handout code that we were provided. We used dynamically allocated sub-vectors and pairs that would hold the robot commands, since this seemed like an easy way to keep the storage duration the way we wanted it as the command lists passed through the various path planning functions. We used enums as well, which made translating n-ary choices such as directions, types of moves, and other modifiers easier to work with.

**Limitations of the program:** While all of the functionality that was requested by this assignment exists in our programs, there are a couple of things that we would have made better about it if we had more time. We decided to implement a Pthread join to collect the threads after they completed their execution, but since they were collected by a single thread in a for loop sequentially, this made it so threads that were finished did not decrement our numLiveThreads variable until threads that were earlier than them sequentially had also completed. In a future version, we could have made the threads detachable or implemented some kind of timeout loop, but we did not have time. Although we cleaned up as many memory leaks as we could, since we used dynamic allocation of variables so frequently, we did definitely did not get all of them. The program uses too many pointers, and it would have been a fun exercise trying to do things differently while maintaining the functionality.

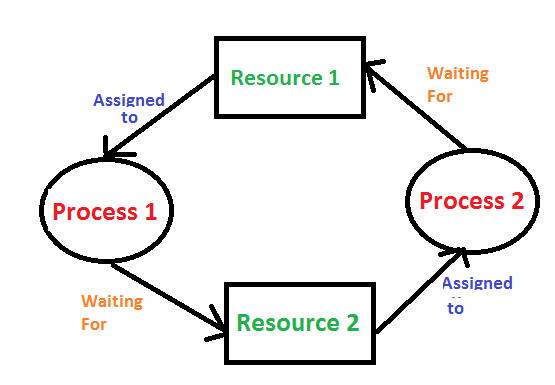
**Difficulties Encountered:** Thiswas an extremely challenging assignment and we encountered many difficulties, described below:

* **Difficult segmentation faults** such as one caused by improperly popping off the front of a vector when it was empty. This took several hours to figure out. We did not know that a vector would still try to remove an item that was empty.
* **Figuring out how to do the path planning.** This took countless hours. It took a long time to figure out how to identify the location from where to push the vector, to add in the behavior moving instead of pushing on the last move, to figure out how to turn, and to figure out which “dogleg” path to take to avoid a collision with the box when traversing to the starting push location. For the latter part, we made a very long if statement that looked at whether the box was going to be a point between the starting point of the robot and the starting box pushing position, and then went the other way to get there if it was.
* **Coordinating on Github.** We had some difficulty working on the same branch and combining our work while effectively resolving conflicts. This was annoying, but was a good learning experience.
* **Figuring out how to coordinate the threads with the GUI.** The GUI aspects of this project could have used some additional explanation. Fortunately, David went to office hours and had some questions answered to save us some trouble with this.
* **Other random problems with C++:** There were many other random problems that we encountered, such as understanding multiple levels of pointers, the extern keyword, and the behavior on our program on a linux machine vs. a macintosh.
* **Figuring out the synchronization design:** We went with the reader writers solution, and we are not sure that it was necessary. But it took some time to figure out and to implement.
* **Navigating through the program:** This is a large and complex program, and we attempted to use bookmarks to navigate through it. But the bookmarks would get deleted when we changed branches on Github. We still have not figure out the most optimal way to navigate through code as complex as this.

**Section 5.2: Deadlock**

Deadlock is the state when multiple processes cannot function anymore due since the resources that they need being occupied. If you look at the diagram below it illustrates this exact scenario where process 1 needs resource 2, but it is assigned to process 2 waiting on resource 1 to open up which is currently assigned to process 1 bringing it back in a big loop.

In terms of our project deadlock occurs on the grid when two robots or boxes are trying to access the same space but the other is blocking their path locking them up permanently.

The way to detect this would be to make another global array that shows the space towards which each robot is trying to move. Then have an algorithm that uses the three global arrays (deadlock whereGoingToArray, robotLoc, and boxLoc) to see if there is a reciprocal occupancy/need relationship with respect to the two robots and any box they are each pushing. If Robot A’s is in cell 1,1 and wants to move to cell 1,2, but can’t because Robot B is in cell 1,2 and Robot B wants to move to cell 1,1 but can’t because of Robot A, then this reciprocal relationship ensures deadlock, and the algorithm would return those robots as being in deadlock.

**Extra Credit Section 5.3: Deadlock Rollback**

There are a couple of different solutions to this problem such as rollback. In rollback, when we detect deadlock we enter the processes into a queue, and then terminate each one and check for deadlock again to determine if it resolved this. We could have used a queue in particular so this we can keep going in order of first in first out. Killing the processes in turn will break the endless loop we encounter with deadlock.

Another potential solution would have been to program the robots to detect when they are entering within a certain radius of another robot or box, and have them update their path to take a different route. This would have required more path planning work than we had time for.

Finally, you could have a robot communicate with another robot in it’s using a pipe to tell the robot to move to the side. Again, we did not have time for a solution such as this.

Bibliography

“Deadlock Detection and Recovery.” *GeeksforGeeks*, 8 Nov. 2021, https://www.geeksforgeeks.org/deadlock-detection-recovery/.