Shamal Siriwardana
John Stevenson
CSC 412
Prof. Hervé
12/22/2018

CSC 412 Final Report

Version 1:

Definitions

Grid Representation
- 0 = Empty grid
- 1 = Grid is occupied with a Robot
- 2 = Grid is occupied with a Box

Movement Clariftion
- 0 = Move North
- 1 = Move South
- 2 = Move East
- 3 = Move West
- 4 = In Place

First the program compares the location of the robot in respect to its box and moves the robot next to the box if needed. Then the box is compared against the door it's suppose to go to and then have the box pushed toward that door.

During comparison, first the box is moved East or West until it matches the door's X position and then The box is moved North and South until the Y positions are the same.

The compBotnBox and compBoxnDoor functions create a new struct of type pushData which contains the number of Spaces the box or robot needs to travel, the direction the box or robot needs to travel, the box/bot id, and in which side the bot is in respects to the box. This data is used by the move function to move the robot next to the box and by the push function, with the addition of the boxSide field being used to denote the position of the box relative to the robot, to push the box to the door.

Version 2:

In version 2 of the project, the code primarily stayed the same, with the exception of the startRobot function being turned into our robot thread function. The data being sent to the thread gave us a couple issues since just passing the local 'i' variable in a for loop doesn't work. This is because the i that gets sent is actually the memory address, and if we are still iterating that i, the value will change as the loop proceeds. Our solution was to allocate an int pointer and assign it the value of i, send the pointer to the thread function, and free the pointer once the thread is ready to terminate. The limitations include a limit on the number of boxes and robots being tied to the size of the grid, otherwise we would loop forever trying to find an empty space for it. Another imposed limitation is the random number function, it is not as random as we would have liked, often assigning the same door to everyone. We tried reseeding the random function to alleviate this, and it did spread the values out a little, but not enough to be unnoticable. The output file was also locked in this version to properly synchronize writing to the output file.

Version 3:

For version 3, again a majority of the code remained the same. Since we were updating the grid since v1, there was very little change here apart from the locks. The way we used the mutex locks was to first allocate and initialize an array of locks in a 2d array of size Col x Row. We head into the initialize function after those get created and for every robot and box that we create, we lock the lock on it. In that way each robot is responsible for both its own lock and the lock of the box it is assigned to. When we go to move the box, we attempt to acquire the lock for the grid space it is moving to, if we get it we set our current spot to empty and unlock the square, then set the box and robot grid locations. We then move on. This way a robot never has to wait on a lock for its own square, or wait on the lock for the box it is trying to move. We also further simplified the locks by using the move function inside the push function, which meant we only had to worry about locks inside the push function when we were actually in position to push the box on the next move. The problem we ran into with the locks had to do with the file lock that was still in the same place it was in the last version. We had put the file lock above the grid lock call, which meant that a robot would acquire the file lock, and if it got deadlocked, the rest of the robots would be asleep waiting on the file lock. This was easily fixed by moving the file lock call to more precise locations only when we are literally about to write to the file. A possible method for detecting deadlocks in a system like this would be to hold, in memory somewhere, at least one copy of the robot and box location lists from a particular number of cycles ago(for instance an array that gets checked against the current array every n number of passes through the draw function). You could then compare that with a current copy of the location lists(robotLoc, boxLoc) and if there are entries that have not changed, there is a deadlock. A possible way to use rollback to rectify the deadlocks might be to hold a third list from 2*n cycles ago and rollback the deadlocked ones to those positions, or even further back, or even the start location for them.