CSC 412 – Operating Systems

Final Project, Spring 2023

Saturday, April 30th 2023

Midpoint due date: May 7th, 11:55pm (explained below)
Official due date: May 10th, 8:00pm (for legal reasons)
Submissions accepted until: May. 13th, 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.

This is a C++ assignment for which you are expected to use C++ threads and locks, rather than elements of the pthread library. You project should build against the C++20 version.

1.2 Solo or group project

This project was designed to be done by a group of two students. You may do it just by yourself, but the load will remain the same.

You may do it as a group of three but some of the "extra credit" components will be mandatory: You must complete two of the extra credit options (If you want additional EC options to get a higher score, I can make up some for you). Additionally, I will require an account of who did what part of the coding (writing the report is not sufficient), as well as a Zoom meeting during which I will ask each group member to explain their contribution to the code.

1.3 About the midpoint version

Important [mandatory]: You must submit a "midpoint" version of the project by the due date. If you fail to submit the midpoint version, then your final project won't be graded. That "midpoint" version should consist of a version of the code that

- Compiles without errors;
- Performs *some* of the tasks of the full assignment (for example, implements multithreading, with each thread properly positioning its robot next to the box it's supposed to push).

Please note that the midpoint version isn't the completed project before cleanup. It really should be a version with only part of the work done. If you move fast on the final project, then submit the midpoint version earlier.

Each group should submit one (and only one) midpoint version of their code, in a zip archive named `Midpoint.zip`.

1.4 Handouts

The handout for this assignment is the archive of a C++ project made up of 2 source files and 3 header files. The code of the handout has the the same OpenGL+glut organization and behavior that we used for the last two programming assignments.

2 Part I: The Basic Problem

2.1 Box pushing

For this problem, you are going to simulate robots that must push boxes on a grid. The robots will be implemented as threads in your program. The boxes will be... whatever you decide to implement them as (or not).

In order to push a box, a robot must be standing on a grid square adjacent to the box, and push the box in the opposite direction. There are therefore only four possible directions for a robot to push a box. We will call these displacement directions *W, E, N, S* (for West, East, North, and South respectively). In Figure 1, the boxes are represented as brown squares while the robots are represented as light green squares.

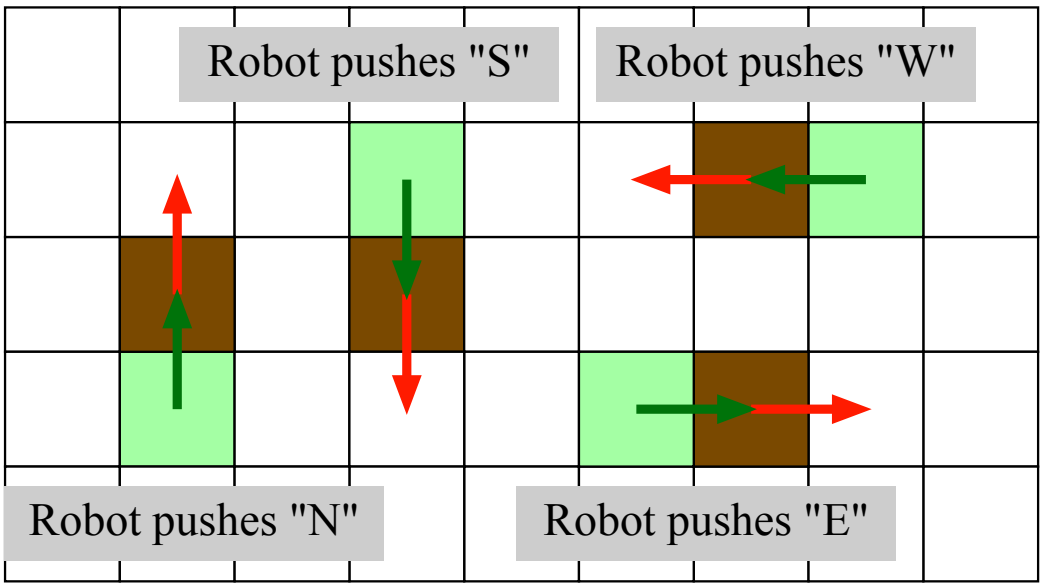

Figure 1: Directions in which a robot can push a box.

In order for a box pushing command to be valid, the square that the robot is pushing the box to must be free (occupied neither by a robot nor a box). Figure 2 shows example of invalid push attempts.

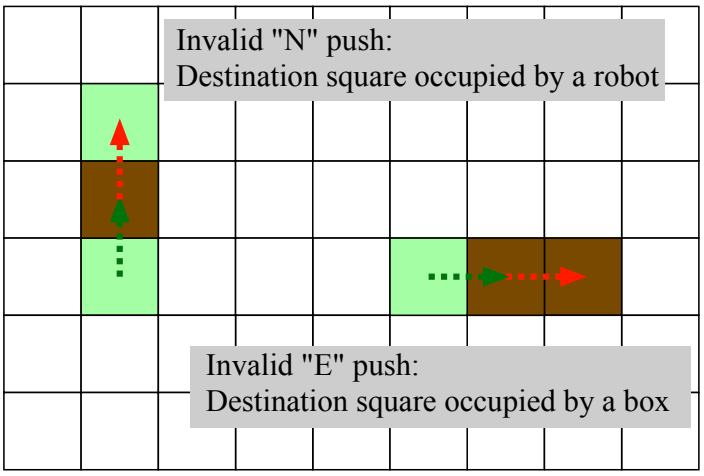

Figure 2: Invalid pushing commands.

2.2 Task assignment

At the beginning of the simulation, there will be one box assigned to each robot (in other words, to have to create as many robot threads as there will be boxes in the simulation). In addition, each robot has a destination "door" for its box. In Figure 3, each robot is shown with a color that matches that of the door it got assigned to. The objective of each robot is to proceed to its box, and then push the box to its assigned door. When the box has reached the door, it disappears and so does the robot (the thread terminates).

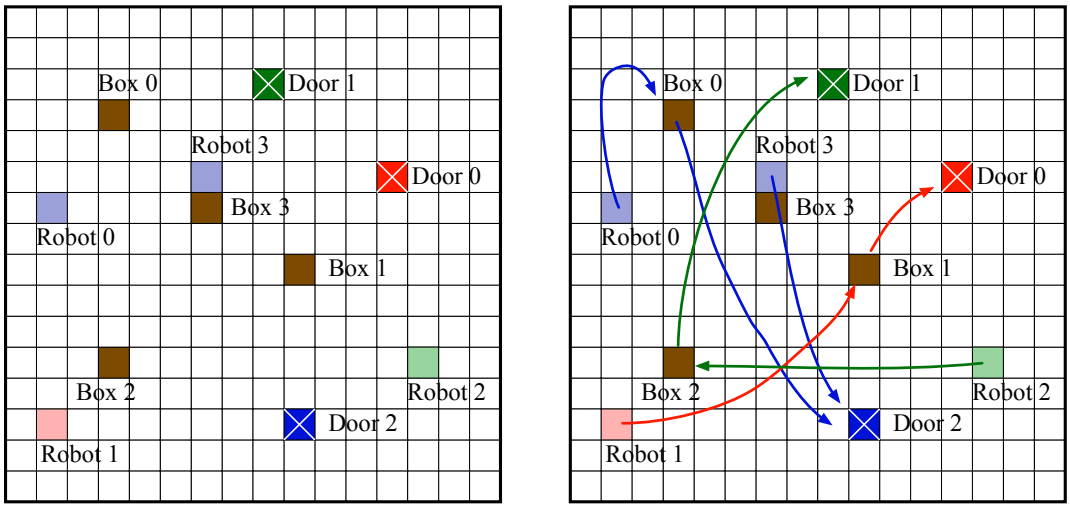

Figure 3: Robots, boxes, and doors assigned.

2.3 Programming language for the robots

The robots only know three commands:

- `move` *direction* (where *direction* is `W`, `E`, `N`, or `S`), to move to an unoccupied adjacent square in the chosen direction;
- `push` *direction* (where *direction* is `W`, `E`, `N`, or `S`), to push an adjacent box into the unoccupied square in the chosen direction;
- `end` when the robot has pushed its box to its destination and terminates.

Admittedly, we could have done it all with simply the `move` and `end` commands (the `push` could be seen as simply a particular case of `move`), but I prefer to make the box pushing more explicit (and therefore easier to debug and evaluate/grade).

To simplify the path planning problem somewhat, we will allow robots and boxes to move over doors other than their designated destination (so you don't have to write an obstacle-avoiding path). The path shown for the box in Figure 4 is therefore acceptable. This assignment is not about designing algorithms for path planning, but about synchronization and deadlocks.

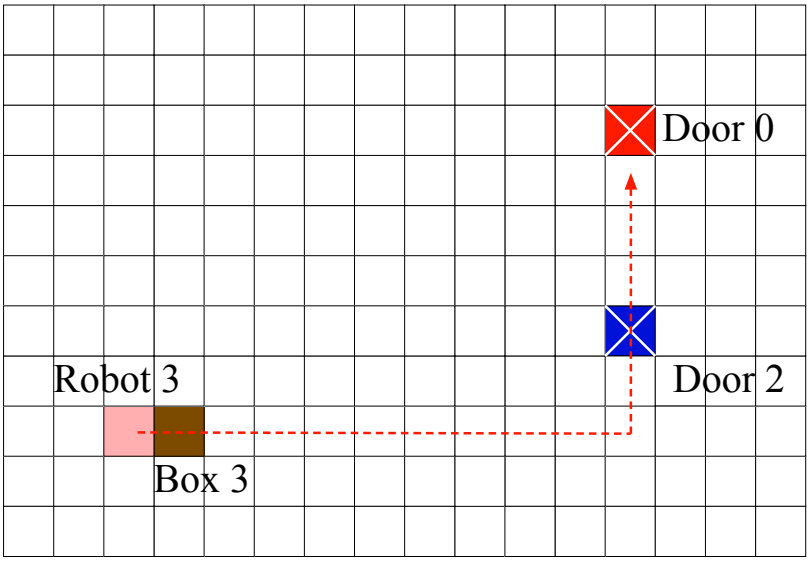

Figure 4: A box or a robot may move over the square of a door other than their assigned destination.

2.4 Input parameters of the program

Your program `robotsV`<*version number*> should take as parameters from the command line:

- the width and height N of the grid,
- the number n of boxes (and therefore of robots),
- the number p of doors (with $1 \leq p \leq 3$),

and then randomly generate:

- a random location for each door;
- an initial position of each box on the grid (be careful that boxes should not be placed on the edges of the grid, that two boxes should not occupy the same position, and that a box should not be created at the same location as a door),
- an initial position for each on the grid (be careful that a robot should not occupy the same position as a door, a box, or as another robot);
- randomly assign a door as destination for a robot-box pair.

So, the grid and robot task assignment shown in Figure 3 may have been produced by launching on the command line

```
robotsV1 16 16 4 3
```

Things are going to look prettier on screen with a square grid but I highly recommend that you do your testing with different sizes of non-square grids. It's so easy to confuse a row index and a column index in code; having a rectangular grid almost guarantees that a row-column accidental swap will result in an "index out of bounds" error that will expose your bug.

2.5 Output of the program

The desired output of the program is a text file `robotSimulOut.txt`. This file will contain:

- First line: the input parameters of the simulation (the size N of the grid, the number n of boxes, the number p of doors);
- A blank line;
- On a separate line, for each door, its initial coordinates on the grid;
- A blank line;
- On a separate line, for each box, its initial coordinates on the grid;
- A blank line;
- On a separate line, for each robot, its initial coordinates on the grid and the number of its destination door;
- A blank line;
- A list of the "program" executed by each robot to complete its task.

For example, a segment of this part of the output file might look like this:

```
robot 1 move N
robot 1 move N
robot 3 push W
robot 2 move E
robot 0 push S
...
```

As you can imagine, there are synchronization problems to solve, and risks of deadlock. We will come to that step by step.

2.6 Basic path planning: a sketch of the algorithm

I will not try to pretend that this path planning algorithm is particularly clever or efficient, but it gets the job done. Feel free to suggest and implement a better algorithm if you believe that this will improve the deadlock problem.

Looking at Figure 5 we see that the simplest path for the box consists of a series of horizontal pushes, to bring the box align vertically with the destination door, and then of a series of vertical pushes, to bring the box to coincide with the door. In the case of the illustration, we see that the difference of column coordinates between the destination door and the box (in its initial location) is $+9$, while the difference of row coordinates is -6. The box will therefore be brought to the door by applying in sequence 9 times `push E`, then 6 times `push N`[1]

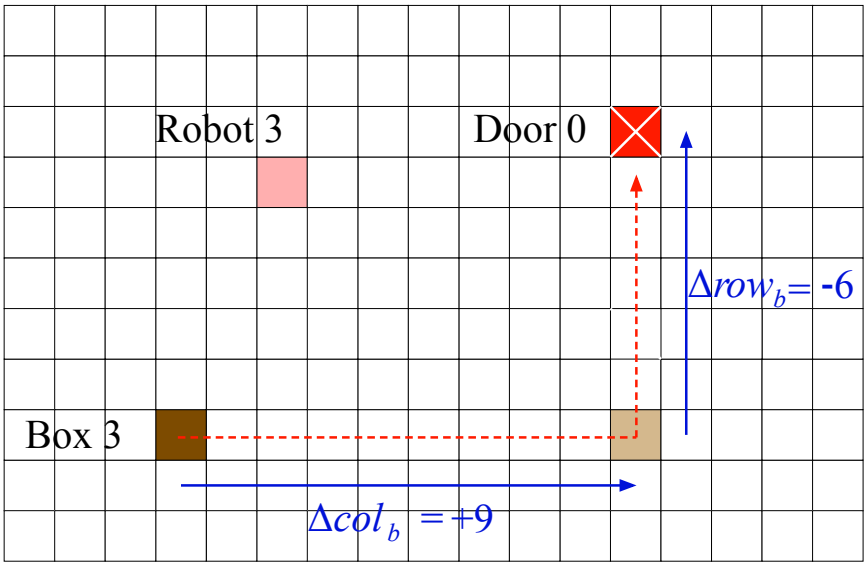

Figure 5: Moving the box to the door.

[1] You may prefer to accomplish the vertical displacement before the horizontal one, but it neither way is systematically superior. Why should it be?.

In order to be able to execute the `push` commands, the robot must position itself on the proper side of the box. Figure 6 summarizes the path followed by the robot to complete the box pushing path[2]. Since the robot is going to perform `push E` commands, it must first position itself on the left side of the box. After it has completed its series of 9 `push E` commands, it must position itself below the box in order to be able to execute `push N` commands. The complete "program"

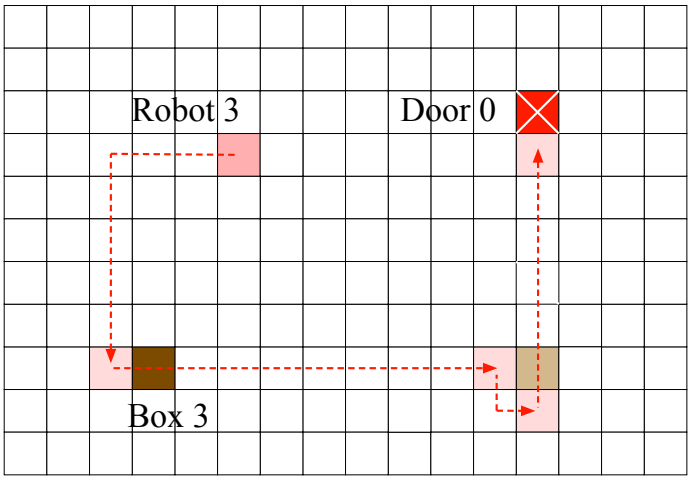

Figure 6: Path followed by the robot.

for Robot 3 to write to the output file in this example would be:

```
robot 3 move W
robot 3 move W
robot 3 move W
robot 3 move S
robot 3 move S
robot 3 move S
robot 3 move S
robot 3 move S
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
robot 3 push E
```

[2]Here again, I had the robot move to a destination by traveling first along the x direction then along the y direction. A zigzagging path going directly to the destination point would be aesthetically more pleasing, but also more difficult to implement.

```
robot 3 move S
robot 3 move W
robot 3 push N
robot 3 push N
robot 3 push N
robot 3 push N
robot 3 push N
robot 3 push N
robot 3 end
```

3 Implementation, Part I: Building and Running the Handout

3.1 The code handout

The code handout consists of 2 source files and 3 header files. There are clearly identified parts that you should probably don't mess with. Modify any of the "don't touch" sections at your own peril.

3.2 Build and run the handout

We already went over this for the past two assignments. By now you should have a working version of OpenGL/MESA and glut and know how to build the executable.

3.3 The GUI

When the program runs, you should see a window with two panes. The one on the left displays a grid showing a grid and the location of robots, boxes, and doors. I initially hard-coded a number of doors, robot, and boxes locations so that you don't stare at an empty grid, but this hard-coding should be be replaced by a random generation (using modern C++ classes and not that horrid C `rand()` function.

I remind you that a robot is paired with the door of the same index. A robot-door pair has been assigned a door they must reach. Robots and boxes are rendered with a colored contour that indicates their target door.

There are a few keyboard commands enabled:

- Hitting the 'escape' key terminates the program;
- Keys '`,`' and '`.`' are used to slow down or speed up the simulation by changing the sleeping time of the robot threads between consecutive moves..

4 Part II: Implementation, No Mutual Exclusion

4.1 A few important points about the implementation

A 2D array of `SquareType` is declared in `main.cpp`, but it is not used for anything at this point. In fact, it isn't even initialized. The grid will be necessary if you implement the "partitions" extra credit, but only for the partition generation function. You can use it or not. This is your choice.

I am going to repeat one more time what I said earlier: You must use the classes introduces in C++11 for threads, mutex locks, and random generation.

4.2 First implementation: `robotsV1`

In this version, you are going to implement each robot as a separate thread, but you are going to solve the planning problem for each robot while ignoring other robots and boxes.

As was the case in Assignments 5 and 6, if you want to join a robot thread in the main thread, for whatever reason, you can only do that if you are absolutely sure that the thread has terminated (otherwise you are going to freeze the rendering and the keyboard input), or do it 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 thread `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.

This version doesn't implement mutual exclusion for grid squares, but there is already a synchronization problem: You must use a mutex lock to protect access to the output file.

4.3 Complete implementation: `robotsV2`

In this version, your robot threads must now verify that the grid square they—or the box they are pushing—move to is free before they perform the move. You cannot use busy wait for this task: It must be done with a mutex lock. It is up to you to decide how many mutex locks should be used for this problem.

As explained earlier, you are not obliged to use the 2D `grid` array. You could decide instead to look directly at the location arrays. The decision is entirely yours. You just need to properly prevent any race conditions.

4.4 Deadlock detection

As you can imagine, as the number of robots and boxes increases, the odds of encountering a deadlock increase. Discuss in the report section of your project a strategy for detecting a deadlock in the simulation. As a starting point for reflection: How would you know that a specific thread is blocked? And the thread must really *blocked* on a lock, not just repeatedly checking if the square it wants to move to is finally free (if a thread is blocked, it cannot communicate anything back. It attempted to acquire a mutex lock and was put to sleep until the lock is released for it).

5 Extra Credit

5.1 Extra credit 1 (8 points): Replanning

If you complete this EC, you should place your complete solution in a folder named `EC1`.

When your robots are merely "stuck in traffic" rather than deadlocked, just repeatedly checking if the square it wants to move to is finally free, there is a chance for them to change their planned route to reach their goal. After a given number of trials, or after some time has passed, the robots should plan a different path to complete their task.

5.2 Extra credit 2 (15 pts): Implement deadlock detection

This EC is mutually exclusive with the previous one. The full 15 pts would be for a complete, working solution for the deadlock detection problem. You would get 10 points for handling properly a couple of particular scenarios to take care of (for example, two robots facing each other, trying to move in opposite directions).

In addition to the robot moves output files, your program should now also print out a file named `deadlocks.txt` in which it lists the robots involved in a deadlock at the time the program was exited, as well as an output of the grid at exit time.

The list of robots involved in a deadlock should be organized as follows:

- On the first line, the keyword `CYCLES`;
- Then, each of the following lines would list the indices of robots directly involved in a "waiting for" cycle;
- Then on a line the keyword `DEPENDANTS`;
- and finally, on separate lines a list of pairs of robot indices: that of a robot blocked waiting and of the robot it is waiting for.

An example of a `deadlocks.txt` file's content would be

```
CYCLES
4 1 5
2 3
DEPENDANTS
0 7
9 5
7 2
```

. . . followed by the ASCII output of the grid produced by a call to `printGrid()`.

5.3 Extra credit 3: Scripting (10 pts)

Write a script that takes as arguments:

- the width and height N of the grid,

- the number n of boxes (and therefore of robots),
- the number p of doors (with $1 \leq p \leq 3$),
- the number m of simulations to run,

runs m times the robot simulation program and numbers the output file for each simulation run, `robotSimulOut 1.txt`, `robotSimulOut 2.txt`, etc.
For this script to be feasible, you must modify the main program of your simulation so that it terminates (closes output file and exits with error code) after a given amount of time if all the robot threads haven't finished yet. Call this modified version of the program `robotsV4` and provide its source code with the rest of the project.

5.4 Extra credit 4: Sliding partitions (12 pts)

The handout contains the code of a function `generatePartitions()`, and a call to that function has been commented out in `initializeApplication()`. There is also a data type `SlidingPartition`. As the name indicates, a sliding partition can slide, along one direction. Horizontal partitions can slide along a row, while vertical partitions can slide along a column. Figure 7 gives an example of a grid before and after two partitions, one vertical and one horizontal, have moved along their preferential direction.

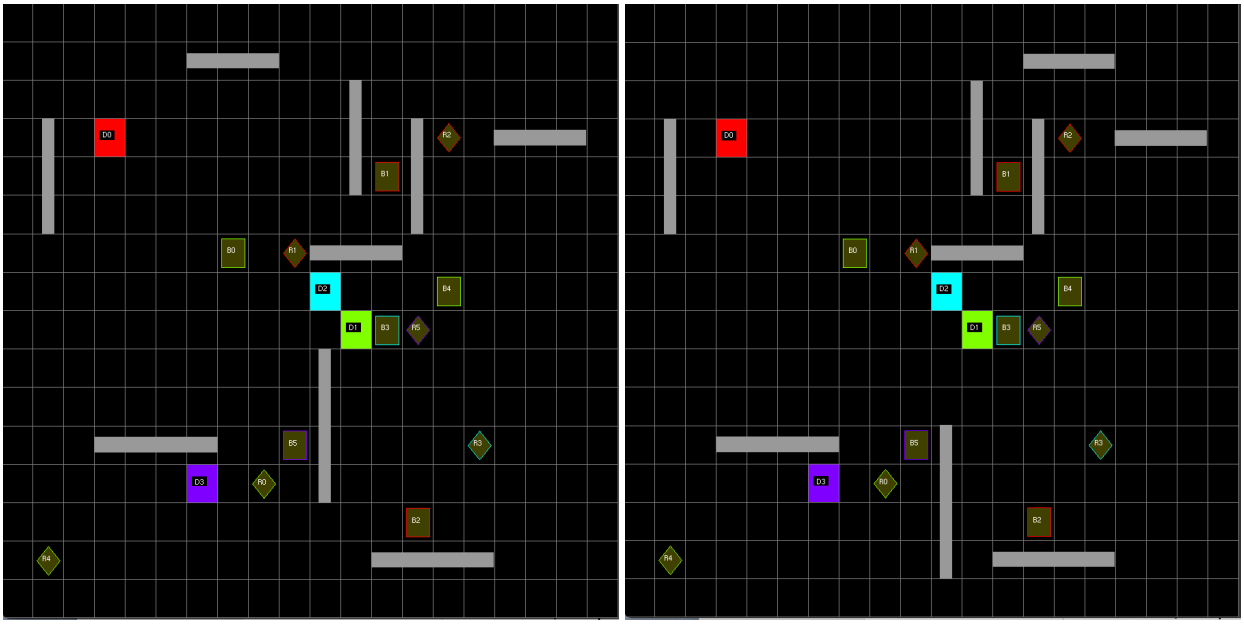

Figure 7: Sliding partitions: Before and after.

A robot is allowed to slide a partition only it it or the box it is pushing stands next to the partition. Needless to say, a partition cannot slide to a square occupied by a robot, a box, or a door.

To move partitions, we are adding one new command to our "robot programming language." For example

```
robot 3 slide 6 E
robot 3 slide 6 E
robot 3 slide 6 E
robot 3 slide 6 E
robot 3 slide 6 E
robot 3 slide 6 E
robot 3 slide 6 E
robot 7 slide 4 S
robot 7 slide 4 S
```

to indicate that Robot 3 slid a horizontal partition seven times to the East, and Robot 7 slid a vertical partition twice to the South.

Having both vertical and horizontal partitions can be complicated. So, for partial credit (8 out of 12 points) you can restrict the random generator `headsOrTails` to produce only one type of partition.

5.5 Extra credit 5: Multiprocessing with common front-end (12 points)

In this version of the project, the parent process, which runs the graphic front end, will create at least 3 child processes that will each run a separate robot simulation (with different initial random locations for doors and robots). The child processes will communicate with the parent process via a pipeline to report the state of their simulation. The user will be able to select which simulation to visualize in the graphic front end.

More details will be provided later on

5.6 Extra credit 6 (10 pts): Deadlock resolution by rollback

In the report section of your project, discuss what it would take to implement deadlock resolution for this problem. I expect something more detailed that a simple repetition of what rollback is (save previous states and return to a state prior to deadlock)

6 What to Hand in

6.1 Source code

The source code should be properly commented, with consistent indentation, proper and consistent identifiers for your variables, functions, and data types, etc.

6.2 Note on the report

In the report, you will have to document and justify your design and implementation decisions, list current limitations of your program, and detail whatever difficulties you may have run into.

7 Grading

7.1 For solo or duo project

- Version 1 completed and performs as required: 20 pts
- Version 2 completed and performs as required: 30 pts
- good code design: 10 pts
- comments: 15 pts
- readability of the code: 10 pts
- report: 15 pts

7.2 For trio project

- Version 1 completed and performs as required: 15 pts
- Version 2 completed and performs as required: 20 pts
- First selected EC completed as required: 10 pts
- Second selected EC completed as required: 10 pts
- good code design: 10 pts
- comments: 15 pts
- readability of the code: 10 pts
- report: 15 pts

(Yes, I know that it adds up to 105 pts).

Penalties (partial list:)

- Use pthread library instead of C++ thread and mutex: -20,
- Use `rand()` instead of the C++ random classes: -10,