CSC 412 – Operating Systems

Final Project, Fall 2021

Saturday, December 7th 2021

Due date: December 21st, 11:55pm (**re-**submissions accepted until Dec. 26th, 11:55pm).

1 What this Assignment is About

1.1 Objectives

In this final project, you will combine together several things that we saw this semester:

- pthreads,
- mutex locks,
- a bit of deadlock management.

1.2 Handouts

The handout for this assignment is the archive of a C++ project made up of 2 source files and 2 header files.

1.3 Solo or group project

This project was designed to be done by a group of two students. You may do it as a group of three but some of the "extra credit" components will be mandatory (more on this later).

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.

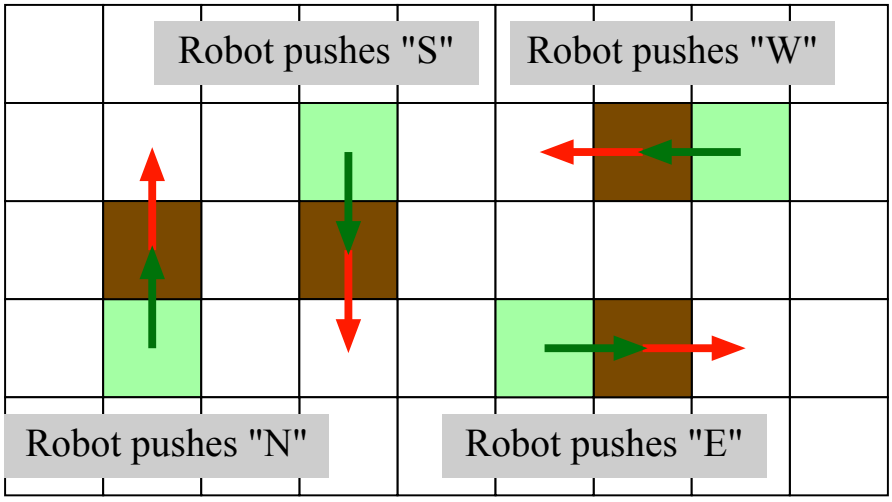

Figure 1: Directions in which a robot can push a box.

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.

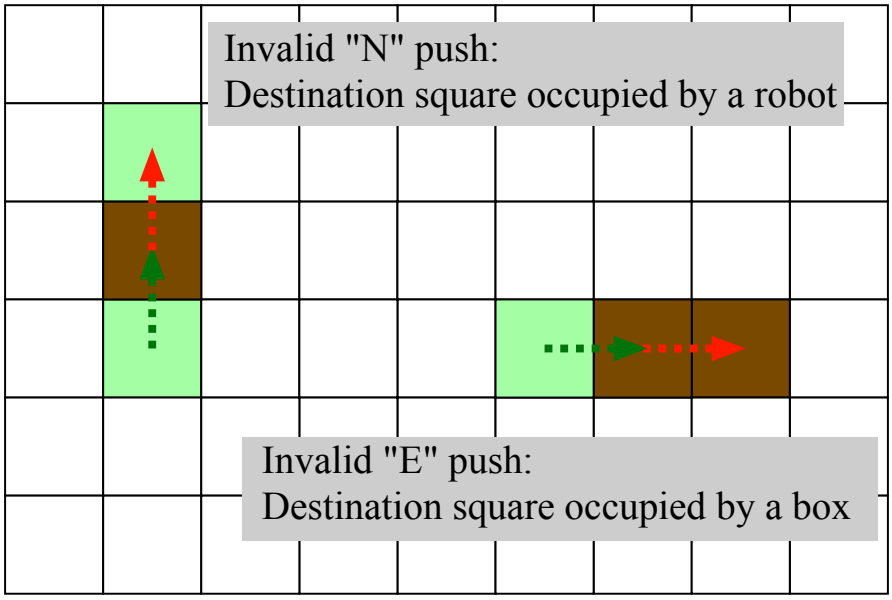

Figure 2: Invalid pushing commands.

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. 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).

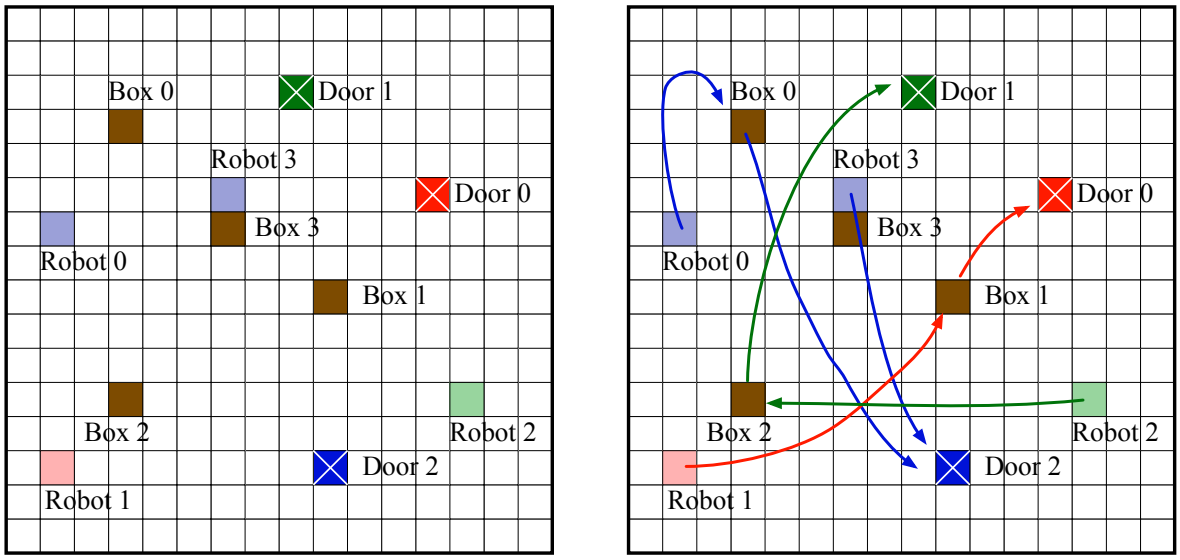

Figure 3: Robots, boxes, and doors assigned.

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.

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.

2.4 Input parameters of the program

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

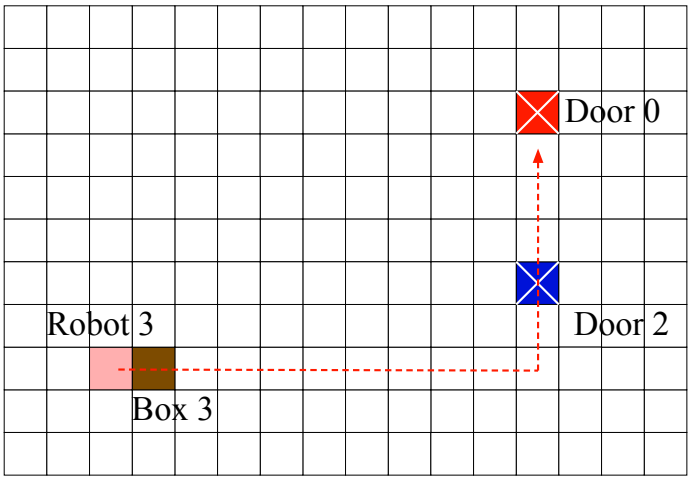

Figure 4: A box or a robot may move over the square of a door other than their assigned destination.

- 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]

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

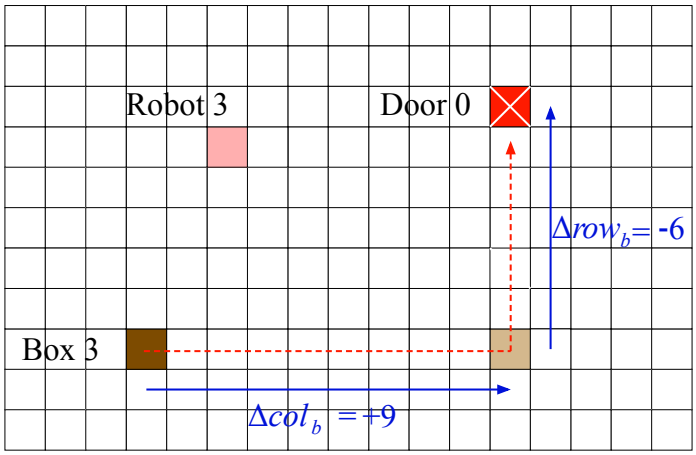

Figure 5: Moving the box to the door.

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.

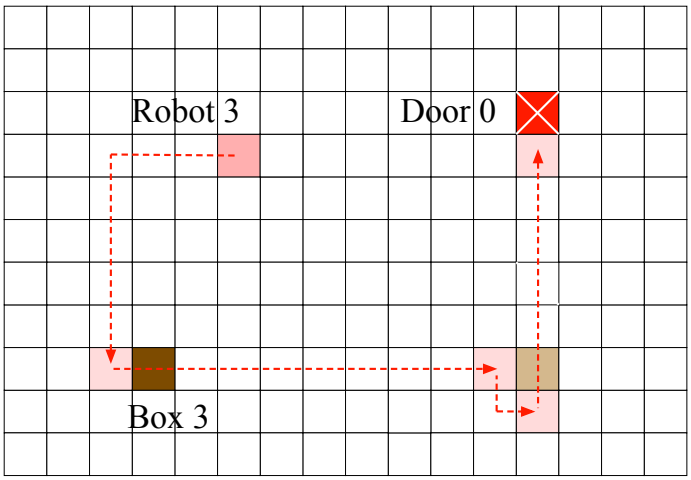

Figure 6: Path followed by the robot.

[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.

The complete "program" 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
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 1 header file. Normally, you shouldn't have to make any changes to the header and source file dealing with the "front end." Even within the `main.c` source file, 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 Install MESA and FreeGlut

MESA is a open-source implementation of OpenGL, the only free option on Linux. FreeGLUT is a more up-to-date version of GLUT (the GL Utility Toolkit), a library that provides binding between OpenGL and the OS[3]. MESA and FreeGLUT come installed by default on Ubuntu, so that you

[3]OpenGL is blissfully OS-unaware. It has no knowledge of input devices, displays, windows, or even of time. All OpenGL says is "give me a buffer/raster to draw into." GLUT is a crossplatform library that gives developers a simple

can run pre-built OpenGL, but in order to build a MESA project, you need to install the developer version of both libraries.

Make sure first that your Ubuntu install is up to date, and then execute:

- `sudo apt-get install freeglut3-dev`
- `sudo apt-get install mesa-common-dev`

3.3 Build and run the handout

After you have installed MESA and FreeGlut, you should be able to build the handout:

```
gcc -Wall main.cpp gl_frontEnd.cpp -lm -lGL -lglut -o robots
```

Of course, feel free to change the name of the executable. What you cannot change, though, is the order that has you list the libraries to load *after* you have listed the source files to compile. This doesn't make much sense, but this is the way you must do it if you want to avoid linker errors.

3.4 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 hard-coded a number of doors, robot, and boxes locations so that you don't stare at an empty grid. Also, the code of the handout shows you how to display things.

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..

Note that currently the size of the text and some aspects of the drawing of the robots, boxes, and doors are hard-coded. If you select a high dimension for your grid, the grid display will look off. I will post in a couple of days a revised version that addresses these problems (as well as the ones that will certainly be reported by some of you).

4 Implementation, Part II: No Mutual Exclusion

4.1 First implementation: `robotsV1`

I don't want you to rush trying to implement directly the full problem, because too many of you will end up with a mess of nonworking code. Therefore, I am going to *impose* some intermediate version, to make sure that you have the basic part of the program working before you add in multiple threads and mutex locks.

---

way to build primitive, cheap-looking interfaces.

In this version, you are going to solve the planning problem for each robot while ignoring other robots and boxes. Furthermore, you will not yet implement the robots as separate threads. Just make sure to implement the path planning for a robot (and output of the computed robot program to the common output file) into a separate function. Since this function is going to turn into the thread start function in the following version, make sure to define a struct type to pass all the information that a robot (and soon, thread) thread will require to do its calculations.

The ”robot program” section of the output file will therefore give the complete program for each robot, listed one after the other (since in this single-threaded version each robot terminates its box pushing task before the next robot undertakes its own task).

4.2 Second implementation, multithreaded: `robotsV2`

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

As was the case in Assignment 5, if you want to join a robot thread in the main thread, for whatever reason, you can only do that 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 `pthread_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.

5 Implementation, Part II: Mutual Exclusion

5.1 Complete implementation: `robotsV3`

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.

5.2 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? (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).

5.3 Extra credit (10 pts): Deadlock resolution

In the report section of your project, discuss what it would take to implement deadlock resolution (by rollback) for this problem.

5.4 Extra credit (15 pts): Implement deadlock detection

I don't expect you to implement a complete, working solution for the deadlock problem. Just select a couple of particular scenarios to take care of (for example, two robots facing each other, trying to move in opposite directions).

5.5 Extra credit (10 pts): Scripting

Write a script that takes 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.

5.6 Multiprocessing with common front-end (20 points extra credit)

[mandatory for groups of 3]

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

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.

6.2.1 Limitations of the program

It is important to be able to identify things that your program cannot do, or cannot do well, and it's crucial that you identify situations that can make it crash. Even if you lack the time to solve these issues, it is important to identify problems. Think of it as a sort of "todo" list for future revisions.

6.2.2 Difficulties you ran into

By this I don't mean "my car was for repair this week." Rather, I mean coding problems that forced you to think of some clever solution.

7 Grading

- Version 1 completed and performs as required: 10%
- Version 2 completed and performs as required: 20%
- Version 3 completed and performs as required: 20%
- good code design: 10%
- comments: 15%
- readability of the code: 10%
- report: 15%