Hayden Kendall

Prof. Jean-Yves Hervé

CSC 412: Operating Systems and Networks

21 September 2020

Final Programming Assignment

While multiprocessing and threading have become incredibly useful tools in computer science, a significant dilemma can still occur. This is known as a deadlock. A deadlock is an issue related to synchronization and resource sharing. Most often when concurrent activity disallows each other from continuation. In this situation, processes cannot communicate with the main thread, with each other, and themselves. Deadlock prevention strategies are implemented to make these situations impossible to occur and often can be narrowed down to four conditions. In this project, a simulation is constructed to display the implications of such a situation.

The simulation that was created consists of a similar grid used in the last project. Depending on the parameters provided by the user, robots, boxes, and doors will be created on the screen. Each fulfilling a designated grid space. It is up to the robots to push their assigned boxes to the doors and terminate. Robots may only push from the opposite direction of the box. One must push from the left side of the box to move it East. Boxes and robots may not occupy the same grid space as one another. This was achieved by creating a two-dimensional array of mutex locks that are associated with each grid square. The results of this are robot threads assigned to complete their job without mutual exclusion, and in some cases form a deadlock.

Robots are defined as a struct within the C header files. The objective of this design was so that robots could be viewed as physical entities that had properties to be adjusted. It was also determined that the box assignment should be stored by the robot and not as a separate entity. I decided on doing this because the robots are entirely dependent on the location of the box. I say this because the box needs to exist for the robot to have a purpose, but also for the fact that initial movement is dependent on the box location itself. In addition to that, all robots are assigned to their own unique box. The doors are their own separate entities because multiple robots will need to congregate at any specific door. Doors follow a similar data structure to store attributes.

A grid space map is also implemented at the beginning of the program. This is to allow the program to have an accurate view of the grid. When the doors are created and put on the grid, the grid space changes from a zero to a one. All grid spaces are set to zero until an entity occupies it. In the program I use this grid space for preventing grid overlap at creation, but also for detecting deadlock. An array of robots is iteratively created in the initialization stage of the program where the location is determined based on the grid and assigns the box location. Immediately following that, the program is multithreaded when the operation function of the robot is called.

The operation function of the robot is what centralizes the steps needed for the objective to be completed. A series of sub-functions are called to move, orient, push, and terminate the robot. Another important design decision was the use of modularity. It makes much more sense to designate each stage of the robot as a sub-function and allow other sub-functions to recycle code. An example of this is the initial approach box function. This function simply takes a delta x-coordinate and delta y-coordinate to moves the robot to the new location. Since this function is dependent on parameters and is general purpose, the function gets reused many times to complete other stages of the objective. By modulating the code, we can reduce repetitive calculations and give the software an overall simpler design. At each step of the robot, a mutex lock is used so that the designated robot thread can write what they did in the simulation and produce the text file with the "robot program".

As mentioned earlier, the main learning objective of this project was to further understand the concept of deadlock. As the number of robots increases, we see a stronger likelihood of deadlock occurring. In the book *Operating Systems: Concepts and Applications* by Donald Horner, we can determine that deadlock occurs due to four conditions. These are mutual exclusions, hold-and-wait, no preemption, and circular wait. It is factual that these four conditions do get met in the program and are therefore resulting in the issue. The question to solve the detection problem was difficult but my solution resulted in utilizing the grid space map further. Not only can we set zeros and ones to grid spaces to determine if they are occupied, but also to store important data like what direction the grid space last moved from and what type of entity it is. At each move, another singular mutex lock is activated to analyze the grid space based on a robot's next move. If on the move, code can be implemented to detect if a deadlock will occur based on collision with other boxes or travelers. Solutions were not accounted for as the assignment did not specify to solve them.

In terms of solving the actual deadlock, I think several solutions can be presented. First and foremost, eliminating one of the conditions previously listed will result in nullifying the possibility of the deadlock. In our case we would want to take the deadlock detection and recovery approach as mentioned by Horner. This technique simply lets the program run until we have a deadlock, stops execution, and rolls-back to a previous state. My solution to validating the grid space is a simpler version of Banker's algorithm. This algorithm is determining if a resource allocation is safe or unsafe before moving forward. If the verdict is unsafe then the process would be withheld until it is safe. In our current code we would have the function to at least begin detecting all situations (only two were done) and have a map of the last functional grid. When a deadlock would be detected we need to first stop the execution of all threads and redraw the grid based on the last grid space map that was successful. We should have determined which robot ran into the deadlock, the path they were on, and information about the other members of the deadlock. Conditionally, the solution would handle alternative pathing or busy waiting but would largely depend on holding execution and resetting to the last known good state.

I did notice that there are limitations to the number of robots in the simulation. In the ballpark of eight or so, we begin to see the terminal throw errors. One that was common was stack smashing detected. It seemed to be a buffer overflow error determined by gcc and I was unsure how to resolve this. I think it is just built into the compiler itself and fixing would require a significant run-around. In the scope of the simulation there is a limitation with deadlock detection. When doing this project within the reasonable time scope of when the course took place, I felt a bit rushed to get the deadlock detection implemented fully. I do not feel confident that testing was thorough enough and that there may

situational bugs where deadlocks are ignored when they should not be. Of course, I only did add two scenarios to detect it so determining when they should properly happen was difficult.

My biggest difficulty with this project was certainly the path functionality. When the robot needed to reorient itself to the correct position. I think what confused me the most was dealing with the main robot movement function and properly using that when the box needed to be pushed on the y-axis. In that case I had to call the function twice but run each delta value independently of each other rather than at the same time. The function was not intended to be used that way so I would not consider it the cleanest method to reorient the robot. The mutex locking of the grid square was also a large contributor to confusion. I took a great deal of time trying to determine how it would be possible for a literal grid square to be locked from other threads. I have only used mutex locks singularly so to have one in an array allocated to each space was a bit of weird concept to get used to.

I have to say that through the difficulties, breaks for other studies, and obstacles that have gotten in my way, I'm both proud and satisfied that I took the time to revisit this course and educate myself on the important concepts in operating systems. Through my time in study, I've not only read from a fantastic author on the theory behind why operating system design is important, but also executed them in well-designed projects from an amazing professor that were intended to expose important programming skills that I did not have before. It is incredibly important that the advantages of multiprocessing, threading, interprocess communication, synchronization, and scripting are used in the daily lives of programmers. These are in fact powerful tools that have been used to create strong applications. While I can never retake the course again, I am reflective on the valuable time I had spent to truly see and master the importance of operating systems in my work.