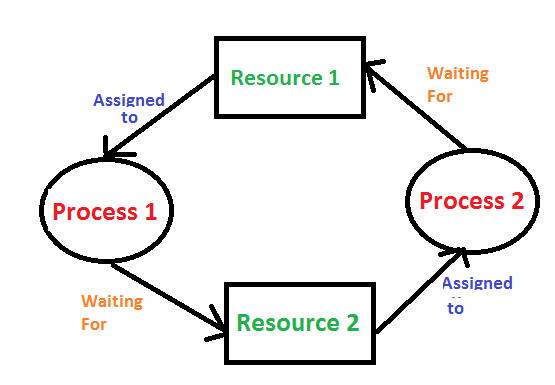
Over the developmental process of this application, we encountered many different challenges. To organize all the different moving parts we incorporated them into a separate file with a custom namespace. Inside of this namespace, we have stored essentially every value the robot has and needs such as robot location. We stored it using a vector of pointers of pairs of unsigned integers. This was so we can reference the data properly whereas a simple vector of pairs being passed by values would be lost due to the scope. We encountered a few problems when it came to mutex locks in specific but when it came to incorporating them to print the robot moves it was quite simplistic locking the file mutex when we needed to write to it and unlocking after we were done with our use. Where something like using mutex locks to protect the location of the robots and boxes was a little more tricky to implement.

In regards to performance, the actual moving and detection algorithm is very time efficient at O(n), however, the space complexity is rather large due to the storage of the structs and vectors. We stored all robot commands to a vector called RClist with a pointer to a pair with the type of move and direction. While this is expensive for space it saves us time having to differentiate all the data when we do output it to the files. There were a lot of different problems that arose from this project; the most difficult being getting the orientation of the boxes to align properly for when they need to push. At first, it seemed like a simple enough concept; however, there ended up being many issues derived from this problem. One of the many ways, with certain orientations the robot would ignore the box and walk through it which took a lot of debugging to figure out a solution. As described in our code 99% of the time the robot will move up/down rows first; however, we added a simple check to see if the robot’s box would be in the path as it goes down. It ended up being as simple as that to check the future path and if so it would move along the columns first instead to avoid this.

Limitations for our project would include the fact there is no deadlock detection for the third version implemented. While it will not crash it will; however, get stuck and never end when there is deadlock. Besides that, there are no known limitations with versions 1,2,3. Given more time however a better tracking system for the resources that are created with new and delete would benefit the application. As well as not only deadlock detection but also deadlock resolution would be the natural progression for the application.

**Deadlock Detection 5.3**

To define deadlock is the state when multiple processes can not function anymore due to the resources they need to be 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. There are a couple of different solutions to this problem such as rollback. Where 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 use 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.

Bibliography

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