Program Analysis

Enforcing Mutual Exclusion

Mutual exclusion is a property of concurrency control to prevent race conditions. A thread is designed in such a way that it is always trying to proceed further into its code. This is done by the semaphore restricting the number of threads to run their part of the program (1 Farmer can cross the bridge, or 5 Customers can enter the restaurant, eat and then leave).

Ensuring the Program is Deadlock Free

The program must be deadlock free – if a process is trying to enter the critical section, one of them must eventually able to do so successfully, provided no process stays in the critical section permanently. In this case, to prevent the race conditions, a thread (Farmer or Customer) have its own conditions to execute any further depending on the conditions of the controller (Bridge, Restaurant). Once one of the threads has finished the logic for their associated run() method, it must notify access the semaphore’s release() for the next thread to begin.

Without monitors, instead of release(), threads must notify() other threads that are waiting that they can begin their execution. This is done easier by running a for loop for the size of the number of threads and having each thread begin notifyAll().

Ensuring the Program is Starvation Free

A form of a waiting queue must be implemented to ensure the program is starvation free. Therefore, once the threads currently running have finished executing their section of code, the controller (being the restaurant as a monitor or semaphore) can easily and immediately locate the next thread to run without any starvation moments.

However, to ensure the thread remains next in queue is done either through the semaphore (keeping track of which thread tried to acquire()and release()) or by using the monitors (wait() and notify() / notifyAll() built-in methods). A waiting thread cannot continue until another thread calls notify() which allows it to continue after that.