COMP 2240 – Assignment 2: Written Report

Harrison Rebesco C3237487

Problem A:

For this problem I used one class; `Farmer`. The `Farmer` class utilizes one semaphore called `bridge`, which protects the method `crossBridge()`from multiple thread access.

There was only one major design issue I encountered for Problem A, where I originally tried making a bridge 'control' class. I decided that this control class was not really necessary, as thread execution was neither time nor order dependant. Because of this, I ended up merging the two classes into one Farmer class to reduce complexity.

The only coding/implementation errors I encountered was when I started, as I was still getting familiarized with thread and semaphore syntax.

Problem B:

For this problem I used two classes; `Customer` and `IceCreamParlour`. `Customer` utilizes two semaphores; `customer` and `table` - which mange thread behaviour. `IceCreamParlour` uses a clock system, in conjunction with the `feedCustomers()` method to control thread execution times.

This problem was the hardest to design a solution for, as customers arrive at different times, thread execution must be time dependant. It was very hard finding a way to ensure only one thread increments the time without affecting the other threads running. My solution to this allows one thread to acquire a lock, while also allowing other threads to release it – this transfer of semaphore locks ensures that the correct thread will always update the time.

I ran into many coding/implementation errors, as this was a time dependant problem using threads. There were many times I encountered a deadlock issue where multiple threads would try and access the same resource, I quickly discovered that it was necessary to use the `Thread.sleep()` function frequently to minimize resource conflicts between threads.

Problem C:

For this problem I used two classes; `Client` and `CoffeeMachine`. `Client` enforces thread behaviour, forcing threads to brew until the `brewTime` is finished. `CoffeeMachine` utilizes synchronized methods that act as monitors to control thread execution and update the time when threads finish brewing.

Originally, I thought this would be the hardest problem to design a solution for, however, because threads must be executed sequentially (C1 before C2, C2 before C3… etc.) all I had to really check for was if the machine temperature was correct, and if there was a dispenser ready for use. The hardest part was ensuring threads updated the time correctly.

The coding/implementation errors I encountered were similar to Problem B – mostly deadlock and timing issues – where two threads would try to access the same resource at once, or a thread would access the resource before or after it was supposed to. To fix this, I once again used the `Thread.sleep()` method which solved all deadlock/timing issues.