**Logo, company name

Description automatically generated**

Operating Systems Concepts

Assignment 2 – Design Programming Project

CSIS 3810

Ernesto Louis

11/21/2022

**The Great Electric-Vehicle Race**

**Methodology**

Conceptual Framework / Overview

This project aims to efficiently demonstrate the use of synchronization through various synchronization techniques using semaphores and threads. This will be accomplished by simulating a race amongst 6 individual cars where each car can access a replenish area one at a time. Through this, the design aims to find a solution to prevent deadlocks or infinite postpones amongst the cars and the replenish area. Threads are used in order to allow each program to run simultaneously, and in this case, the cars to run around the track at the same time.

Goals

The goals (in mind) are to allow the cars and supply coordinator to communicate, provide, and stall vehicles permitted/not permitted into critical areas. Another is to create robust algorithms that allows the cars to run indefinitely until the user decides to terminate the program. I also wanted to provide some form of maintenance and scalability within the code thus, I created the Item and Vehicle class so that the specific cars and items could then be derived form them and I can easily pass in the necessary arguments to its parent classes.

Strategies

To efficiently allow ease of scalability between the types of cars and items, this project will extend concepts such as inheritance and polymorphism. Created are vehicles and Item class which set templates for different of its types to be created. I created it in this manner such hat scaling the classes and the derived classes would be much easier and it was, as there were pieces moved around as the program evolved. Through composition, each vehicle would compose of the 3 required items.

When designing the algorithm, I needed to keep in mid the logical process of when the shared resource should be locked and released, such that deadlocks and indefinite postponement do not occur. Later, the concept and logic for this scenario will be detailed. The algorithms also need to determine how it will process items that were not outputted by the supply coordinator but is critical so that the vehicle can continue to run it laps.

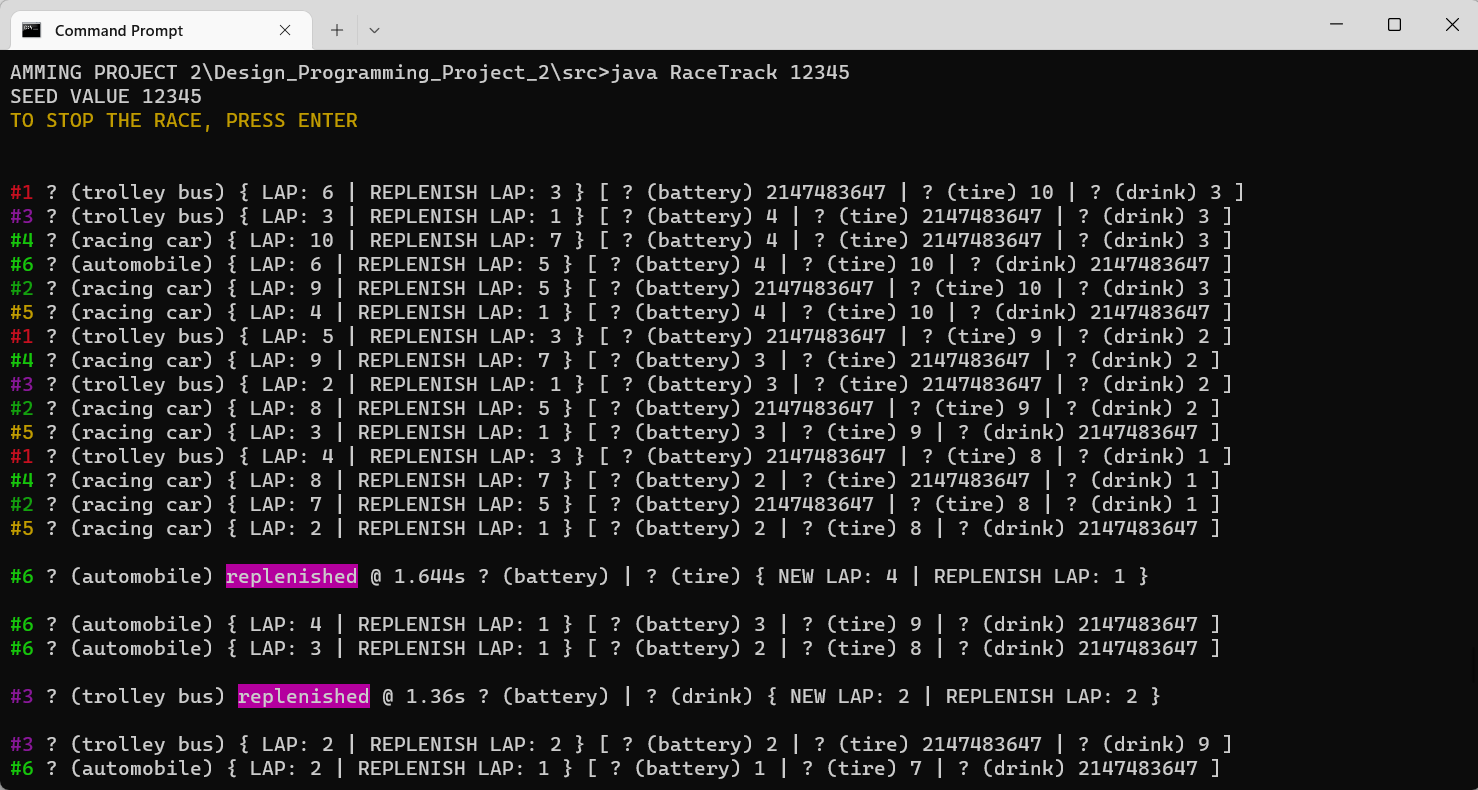
**Design**

Data Design

At the start, the user input a seed value.As the cars are created, the seed value is passed into the constructor as it will be a shared resource as well such that unique sequences re distributed at each run of the program. Each car then consists of an item in infinite quantity.

For each out, when a car has run a lap, it displays the current lap that it is on and the lap until it needs to replenish. Additionally, it also output the quantity of items left after each lap. As the car replenishes it takes a certain amount of seconds before it leave the replenishing section and releases the semaphore.

Below is a sample output



Data Structures and Algorithms Design

*Classes*

**Vehicle*.*** This class provides a template for all classes that will be derived from it, the parent class, essentially. The class consists of the properties such as the laps the specific car will run. It also takes an attribute value that determines what lap it will need to replenish.

This consists of the overridden method extended via the Thread class and this method is solely responsible in continuously allowing the cars to race around the track indefinitely and determine when it is time for the vehicle to replenish on item.

**Automobile.** This class inherits the properties of the vehicle class uses its parent constructor in order to initialized its own properties and provide accessible methods.

**SportsUtilityVehicle.** This class inherits the properties of the vehicle class uses its parent constructor in order to initialized its own properties and provide accessible methods.

**RacingCar*.*** This class inherits the properties of the vehicle class uses its parent constructor in order to initialized its own properties and provide accessible methods.

**Item*.*** This class provides a template for all of the 3 specific items need to ensure that a vehicle can continue its laps.

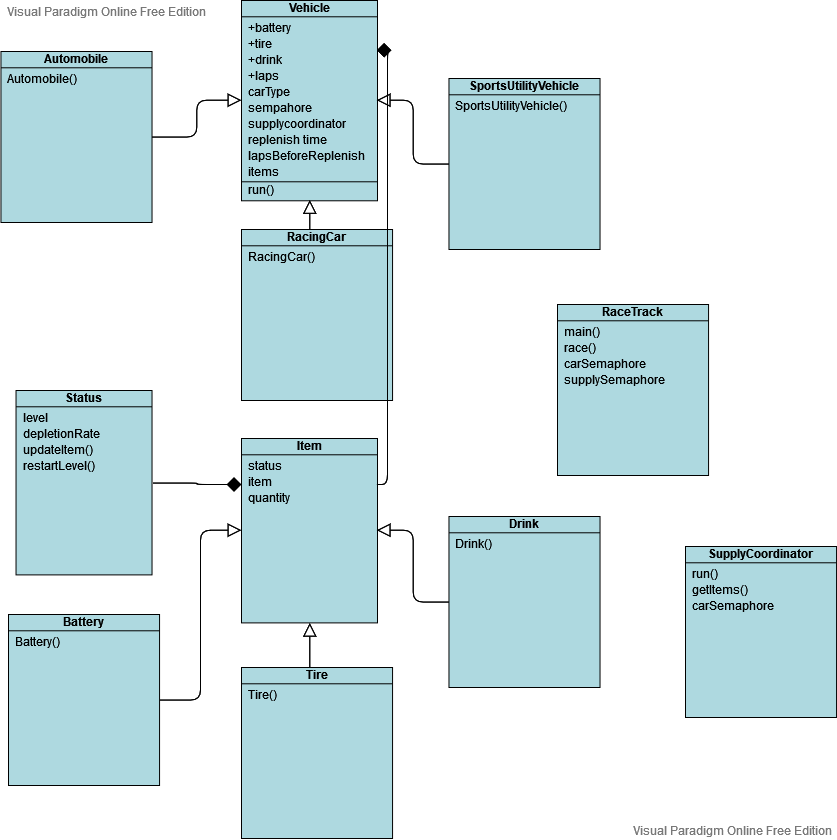
**Battery*.*** This class inherits the properties of the Item class using its parent constructor in order to initialize its own properties and provide accessible methods and is compose within the vehicle class as an item required in the vehicle.

**Tire*.*** This class inherits the properties of the Item class using its parent constructor in order to initialize its own properties and provide accessible methods and is compose within the vehicle class as an item required in the vehicle.

**Drink*.*** This class inherits the properties of the Item class using its parent constructor in order to initialize its own properties and provide accessible methods and is compose within the vehicle class as an item required in the vehicle.

**Status***.* Though not used for this assignment, this class would have served the purpose of checking the individual status of each item. The idea was to use a (random) depletion rate in which the item will deplete after each subsequent lap. Each derived Item class would compose of a status object

**CarColor***.* This class allows a random color to be associated with car within the console/command line so that tracking the car is much more organized.



*Algorithms*

**Replenish Algorithm*.*** This algorithm, within the Vehicle class, in the run() method, hinges on the current status of the car’s lap, batteries, tires and drinks. This algorithm checks to see two things: one, has the lap come to an end in which the vehicle must make an immediate stop, and two, has the vehicle run out of any specific item (besides the on in infinite quantity). Once this is deemed true, the very first thing that occurs is that the shared semaphore that determines if a driver is currently at the pit stop is acquired. Then, the supply coordinator outputs 2 uniquely generated items for the driver. If it is the case that the driver did not get the proper contents, the supply coordinator will dispense another 2 uniquely generated items until the driver can fulfill it requirements and replenish properly. Once the vehicle is replenished the vehicle is given a new set of lap which keeps the while loop going continuously and never reaches false. Subsequently, the shared semaphore is released allow (potentially) the next vehicle to replenish and run more lap.

Replenish Algorithm Pseudocode:

*Run(){  
  
 While( laps not OR any item not close to empty ){  
   
 subtract 1 from laps and items  
   
 if( laps finished OR any item is empty ){  
   
 lock the supply coordinator  
   
 get items from coordinator  
   
 while( supply coordinator does not have items needed ){  
   
 get items from coordinator  
 }  
   
 for( all items in the given by the supply coordinator ){  
   
 check which items needs to be replenish  
 }  
   
 get new laps and laps to replenish for the vehicle  
   
 release lock form supply coordinator  
   
 }  
   
 }  
  
}*

**Item Distribution Algorithm*.*** Within the SupplyCoordindator class and in the getItems() function, this algorithm uses the seed value to allow specific distribution of sequential values, this algorithm creates an array of 3 possible items, battery, tire and drink. Two variables are the used to track 2 unique indices so that those items at the indices can be returned to and used by the vehicle. The algorithm makes sure no duplicate indices or items will be reproduced such that the driver will have 2 unique items at every stop.

Item Distribution Algorithm Psuedocode:

*getItems(){  
  
 array -> { battery, tire, drink }  
   
 index 1 -> value between 0 - 2  
   
 index 2 -> value between 0 - 2  
   
 while( index 2 is the same value as index 1 ){  
   
 assign index 2 a new value   
 }  
   
 return array items at index 1 and index 2  
  
  
}*

*Problem Analysis*

As the threads run simultaneously, depending on the task gets to the critical area first, that is, a car has finished its laps first, the shared semaphore would call a method called acquire which then “broadcasts” (in a sense) to the other cars coming in that the critical section is being occupied. While this occurs the task stalls and keeps checking if the lock on the section has been released.

Through the use of semaphores and running threads simultaneously, I was able to achieve and design a program that would efficiently allow tasks to access resources. Achieving these have also allowed me to demonstrate my understanding of synchronization and optimizing ways to prevent deadlocks and indefinite postponement.

Architectural Design

*Initialization*

At startup, the seed value is displayed to ensure the user it will successfully be utilized to ensure a specific distribution of sequences.

*Semaphores / Threads*

Leveraging the semaphore class and thread class, each vehicle class takes in the shared supply semaphore. From the car classes, threads are created run for their respective tasks. The supply semaphore controls any access to the resource that is the supply coordinator. The supply coordinator also makes sure that only one car is accessing it at a time. The shared semaphores only carries one permit such that only one task can access a critical section at a time as the threads run independently of each other. Another semaphore include is a counting semaphore which would track the capacity within the track. This semaphore would lock the racetrack if the cars on the track exceeds the limit and an incoming car wants to join the race.

*Termination*

To successfully end the program, the user is allowed to click any button which will then terminate the program, as it initially runs continuously.

**Program Simulation/Output**

To run the program, write out the commands to the terminals in order, “javac RaceTrack.java”. Once that command is executed, run “java RaceTrack seedValue” where seed value is of type long.