1. For my simulation I chose to use a Binary Semaphore by implementing the build in java.util.concurrent.Semaphore class. To do this, I simply created a new Semaphore Object:

Semaphore binary = **new** Semaphore(1, true);

This creates a semaphore which can only hand out one permit at a time, with the “fairness” flag set to true. The fairness setting is turned on here because I wanted to ensure that the same lane would get no more than one chance to let cars through at a time, and the “traffic monitor” would alternate between the two lanes, instead of just letting one get down to zero, and then switching to the other lane.

1. There are two critical sections in my code, and they used the shared resource of a “Road”, which is declared in line 97:

Road road = **new** Road();

This road has one field, road, which is a Linked List, and two methods, Arrive and Depart. The road Linked List simply stores cars while they are “Passing Through” the one lane section. Arrive takes a Linked List argument, and removes the first element in that list, and adds it to the road Linked List. Depart does the opposite, removing a car form the road Linked List, and returning it:

/\*\*

\* A LinkedList to represent cars currently in the critical section

\*/

LinkedList<Car> road = **new** *LinkedList*<Car>();

/\*\*

\* This method is to represent a car moving into the critical section, the closed down "lane"

\* @param *queue* The queue to take the car from (North or South)

\*/

public *void* arrive(LinkedList<Car> *queue*){

road.add(*queue*.removeFirst());

}

/\*\*

\* A method to represent a car leaving the critical section

\* @return The car which left the lane

\*/

public Car depart(){

Car car = road.removeFirst();

car.totalWaitTime = System.currentTimeMillis() - car.startingWaitTime;

return car;

}

There are two critical areas in my program. The first being in a thread that I made to handle cars headed north, and the second being in a thread that I made to handle cars headed south. These are in lines 159-178, and lines 193-211 respectively.

try {

binary.acquire();

*long* startTime = System.currentTimeMillis();

while(System.currentTimeMillis() < (startTime + BUFLENGTH)){

if (northBoundQueue.size() > 0){

road.arrive(northBoundQueue);

sleep(TRAVELTIME);

passedCars.add(road.depart());

}

}

} catch (Exception *exc*) {

exc.printStackTrace();

}

binary.release();

try {

binary.acquire();

*long* startTime = System.currentTimeMillis();

while(System.currentTimeMillis() < (startTime + BUFLENGTH)){

if(southBoundQueue.size() > 0){

road.arrive(southBoundQueue);

sleep(TRAVELTIME);

passedCars.add(road.depart());

}

}

} catch (Exception *exc*) {

exc.printStackTrace();

}

binary.release();

1. These critical sections are guarded by this line:

binary.acquire();

This works because the method “acquire()” blocks the current thread until it can acquire a permit from the semaphore object, so if another thread already has the permit, this thread cannot execute the code following the acquire() method. Further, this ensures that only vehicles heading in the same direction can move for the buffer time. Once the permit is acquired, the current system time is stored in a variable, startTime, and a while loop is executed which only ends if the system time is greater than startTime + the buffer time, which is declared in an earlier variable, BUFTIME. Both threads are identical, save that one works with the cars going North, and the other with the cars going South.

1. My data are generated randomly, using a Car object, and constructing it with a boolean telling us if it is going North or South, and an Arrival Time, telling us when it is pulling into the intersection.

for (*int* i = 0; i < NUMCARS; i++){

cars.add(**new** Car((rand.nextFloat() < .5), (Math.round((rand.nextInt(SIMTIMEFRAME) \* 60000) \* SIMSCALE))));

}

This is run according to how many cars are being simulated. As can be seen above, rand.nextFloat() < .5 gives us either true or false depending on the random floating point value generated, and this will tell us where the car is headed, additionally, the cars arrival time is generated by selecting a random integer between our constant 0, and SIMTIMEFRAME, excluding SIMTIMEFRAME, and the multiplying that by 60000 to get the value in milliseconds, and finally multiplying by our SIMSCALE variable, which allows us to run the simulation faster, or slower, by multiplying all time values by the same amount. I have been running the simulations at .01, as a SIMSCALE of 1 correlates to 24 hours when SIMTIMEFRAME is 1450. Once the cars are generated, they are sorted and then added to their respective LinkedLists as their respective arrival times come.

1. For the purposes of my simulation, I assumed that 15,000 cars were using this road throughout a 24 hour period. I randomly generated the 15,000 cars with a predetermined time of arrival at the critical section of the road, along with a direction represented by a boolean, “true” for cars heading north, and “false” for cars heading south. I also assumed that the cars would spend five seconds travelling through this critical area, as it is 180 feet long, and each car is 15 feet long, and they are all assumed to be travelling at 25 miles per hour, which comes out to 36.66 feet per second. This simple math gives us: (180ft + 15ft + 3ft) / 36.66 fps = 5.40 seconds

But, for simplicity’s sake, I have rounded this down to five seconds, as some cars may be going slightly faster, and some slightly slower, so I feel that five seconds is a nice round number for this simulation’s purposes. Finally, I sped the simulation up by a factor of .01, so instead of taking 24 hours it would take roughly 15 minutes.

To run these simulations, I wrote a driver class which simply creates five threads, one for each buffer time, and runs them all side by side, and then returns the average wait time of each simulation, in milliseconds.

Finally, I believe that the two-minute buffer policy is the most optimal, assuming that the most optimal method is the one with the least wait time.

Using the above stated methods and assumptions, the results for 15,000 cars, randomly generated come out to:

|  |  |
| --- | --- |
| Buffer Time (In MS) | Average Wait (In MS) |
| 120000 | 589815.46 |
| 360000 | 536486.76 |
| 480000 | 393065.21 |
| 600000 | 306732.62 |
| 720000 | 370740.11 |

In minutes, this table calculates to:

|  |  |
| --- | --- |
| Buffer Time (In Minutes) | Average Wait (In Minutes) |
| 2 | 9.83 (9 minutes and 50 seconds) |
| 6 | 8.94 (8 minutes and 54 seconds) |
| 8 | 6.55 (6 minutes and 33 seconds) |
| 10 | 5.11 (5 minutes and 7 seconds) |
| 12 | 6.18 (6 minutes and 11 seconds) |

These data bring me to the conclusion that the most efficient buffer time is in fact ten minutes; That is to say, on a two lane road with road work being done on one of the lanes, the most efficient way to let traffic through is to set up a traffic light with a monitoring system that allows cars to go through either side for ten minutes at a time.

Side Note: To run the program, I highly recommend using the pre-built Main function that I used for testing, but I tried to document the program to the best of my ability, so it (hopefully) should be clear if one wants to play around with it.