Homework 2 for CS153 (Winter 2019)

Due: 2/8/2019

Instructions:

I. Write synchronization code to simulate each of the following scenarios:

   a. (3 points) A barrier: a group of us go to a restaurant; we wait until the last person arrives before we go in.

   This pattern is similar to the last reader thread exiting in the readers-writers pattern.

   ```
   int group_count = N; //initialized to the number of people in our party
   Semaphore barrier(0); //initialized to 0
   arrive() {
      mutex.wait();
      group_count--;
      if(group_count == 0) {
         for(int i=0; i < N; i++)
            barrier.signal();
      }
      mutex.signal();
      barrier.wait();
   }
   //party time!
   }
   ```

   This particular pattern can also be implemented nicely with a monitor since the for loop represents a broadcast, which we can get with a condition variable inside the monitor.

   b. (3 points) A bakery where threads of three types representing three ingredients cake mix, filling and icing arrive. Whenever we have one of each, we make a cake.

   The key to this problem is to check if there are at least 1 of each ingredient. The code for icing is shown, but filling and cake are also similar. This code is written with the assumption of 1 of each thread, but it can be generalized by resetting the ingredients when cake is made. Its also acceptable to have only one thread check (the check would be different; I am including this implementation at the end as an example), or to have a separate thread that checks the three ingredients, simplifying the ingredients role to be just to signal arrival and wait to be told they are done.

   ```
   Semaphore mutex(1), icing(1), cake(1), filling(1); //all mutexes initialized to 1
   int icing=0, filling=0, cake=0;
   Icing() {
      icing.wait(); //one icing at a time
      mutex.wait();
   }
   ```
if (cake == 1 && filling == 1)
{
    makeCake(); //success
    icing.signal(); filling.signal(); cake.signal(); //let more threads in to make more cake
} else {
    icing ++;
}
mutex.signal();

Alternative implementation with only icing checking:

Semaphore icing(1), cake(0), filling(0), fillingArrived(0), cakeArrived(0);

int icing = 0, filling = 0, cake = 0;

Icing() {
    icing.wait(); //one icing at a time
    cakeArrived().wait();
    fillingArrived.wait();
    makeCake(); //success
    filling.signal(); cake.signal(); //let more threads in to make more cake
}

Final implementation would have all three threads similar to cake(), and another thread (e.g., cook) that waits on all three ingredients and then lets them go. You should get credit for any of these, and I suspect there are other valid implementations.

c. (3 points) The recipe in part b has changed – now we need two portions of cake mix to arrive (in addition to filling and icing) before we can make cake. Update your implementation.

This part is similar to the previous part, except we have to keep a count of the cake mixes and have the second cake mix only carry out the signaling for the first and second implementations above. The implementation with a single coordinating thread (cook) would simply change wait twice on the cake mix and signal it twice, otherwise be the same.
II. You are writing code for the voting machines for an upcoming election. You use shared counters, one for each candidate to keep track of the votes as they come from the different voting machines. You can think of each machine as a thread: every time it receives a vote, it increments a counter for that candidate.

(a) (2 point) Explain what could go wrong with this implementation if we do not use synchronization

We lose votes as multiple machines read the count at the same time, add their new vote, similar to the bank withdrawal example we went over in class.

(b) (2 points) Suggest two ways to use locks to solve this problem without changing the code other than adding the lock operations; which one is more conservative.

Can have one big lock for the vote count, or a separate lock based on the candidate for whom the vote was cast. The first implementation is more conservative since it serializes all updates. The second could allow updates to two different candidates at the same time.

(c) (1 point) Consider the following improvement to the implementation suggested by a cs153 veteran: for each thread, maintain a local count of the votes, and then update the global count periodically. Do we still need synchronization?

This is called privatization (creating a local copy of a variable) and is a commonly used technique to optimize concurrent programs. We still need to synchronize on the update of the global.

(1 point) Compare the implementation in c to the better of the two implementations in b.

Since the update to the global variable is happening less often now, less overhead and contention for the locks is experienced. Local updates do not have a race condition and can be directly updated without synchronization.
III. (5 points) Traffic in Manhattan goes around a block as shown in the figure below.

![Traffic Gridlock](https://upload.wikimedia.org/wikipedia/commons/thumb/d/d9/Gridlock.svg/440px-Gridlock.svg.png)

Having studied concurrency, you recognize that even though we call this gridlock, this situation may be a case of deadlock. Use our criteria for deadlock to show whether this is indeed deadlock or not.

Here we want to identify who are players (processes/threads) and resources are. The players are each traffic in a particular direction. The resources are the intersections. 

- **Mutual exclusion**: yes, each intersection can be held by only one of the two directions competing for it.
- **Hold and wait**: yes, each direction holds an intersection and is waiting for the other
- **No preemption**: yes, there is no way to back out of an intersection or to force our way through it without causing accidents.
- **Circular wait**: yes, East is waiting for North, North is waiting for West, West is waiting for South, and South is waiting for East – we have a circular wait.

If this is deadlock, discuss and compare two solutions to prevent it from happening.

We can try to prevent the different ingredients of deadlock. Mutual exclusion is a property of the intersection and we can’t prevent it. Hold and wait can be enforced by making each direction get both intersections together so that it is either holding or waiting. We can prevent circular dependency by making at least one direction get its second intersection first. These solutions do not make sense for cars since you cannot reserve an intersection without moving into it physically. So, what we do is use traffic lights and fine cars that block the intersection so we do not end up with any red cars blocking traffic.
IV. (a) (5 points) Show how you would simulate traffic on an intersection with two directions for traffic flow, say N and S. Each car (a thread) arrives in one of the two directions. Your intersection should let 5 cars pass from each direction before switching to the other.

You can approach this as readers writers, with the advantage that we know the total number of readers in each direction.

Semaphore  Mutex_N(1), Mutex_S(1), N(5), S(5), N_or_S(1); //

North () { //dual for S
    N.wait(); // allow up to 5
    Mutex_N.wait();
    N_count ++;

    If(N_count == 1) //first N
        N_or_S.wait(); //wait until north turn
    Mutex_N.signal();

    CrossBridge();

    Mutex_N.wait();
    N_count --;
    If(N_count == 0) //last N
    {  
        N_or_S.signal();
    S.signal(); //repeat this 5 times
    }
    Mutex_N.signal();
}

Note that this will switch after less than 5 cars if there are no more cars available. It also carries the credit of any unused turns for future turns. Both of these can be cleaned up with some work. For example, to make it switch after exactly 5, change the condition for the release section (i.e., the code after crossBridge()) to check N_total. We add an increment of N_total right before or after the increment of N_count, and change the check of N_or_S.wait() to be based on N_total. The updated code looks like:

Semaphore  Mutex_N(1), Mutex_S(1), N(5), S(5), N_or_S(1); //

North () { //dual for S
    N.wait(); // allow up to 5
    Mutex_N.wait();
    N_count ++;
    N_total ++;

    If(N_total == 1) //first car in this direction
        N_or_S.wait(); //wait until north turn
    Mutex_N.signal();

    CrossBridge();

    Mutex_N.wait();
If(N_count == 0 && N_total==5) //last N
{
    N_or_S.signal();
    S.signal();  //repeat this 5 time
    N_total=0;
}

Mutex_N.signal();}

In the new implementation, we only check if we are the last thread, and 5 have already passed before switching. If there are only 3 threads, we keep the direction the same until 2 more come in.

(b) (2 points bonus). Explain how you would update your implementation so that emergency cars do not have to wait (they bypass the cars in the same direction). They should still wait for the traffic light to switch to their direction before crossing.

We can treat this the same way we use the fast pass at amusement parks. Once you stand in line, even when you get to the front, there is another check to see if there is fast pass customer.

So, we can add a semaphore right before cross bridge. The emergency vehicle would wait at this semaphore directly, while the other cars have to go through the beginning section first.