Homework 2 for CS153 (Fall 2017)

Due: Optional/ungraded homework

Instructions:

1. Write synchronization code to simulate each of the following scenarios:
   a. Two players playing table tennis

   This is exactly the Frisbee problem with two players.
   Semaphore haveBall[2]; //initialize one to 1, the other to 0
   Semaphore waitingForBall[2]; //initialize in the opposite way to first semaphore
   Player() {
      haveBall[0].wait(); //I have the ball
      waitingForBall[1].wait(); //other player doesn’t
   }
   //hit ball!
   haveBall[1].signal(); //now you have it
   waitingForBall[0].signal(); //and I can receive it
   }
   Since we only have two players/1 ball, we can solve this problem with a single semaphore (e.g., get rid of waitingForBall semaphores and operations on them), but the solution does not generalize to multiple balls/players.
   You can also solve the problem correctly using locks, monitors etc.. since the problem did not specify.

   b. Taxi waits for a passenger then drives them to their destination

   This is the “happens before” pattern where you keep one activity waiting for the other to order them.

   Semaphore passenger(0); //initialized to 0
   
   Taxi() {
      passenger.wait();
      Drive();
   }
   Passenger() {
      Arrive();
      passenger.signal();
      //enjoy the ride
   }
c. 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.*

```c
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.*

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) Explain what could go wrong with this implementation if we do not use synchronization

Updates to the counters represent a race condition. So, we can lose updates and end up with under estimate of the vote tally.

(b) 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.

*One conservative way is to add a lock to the set of counters so that only one update can happen at a time. This will lead to unnecessary serialization between updates to different candidates.*

*The better solution is to add a lock for each individual counter so that serialization only occurs between updates to the same counter.*
(c) 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?

Yes, we need synchronization when the local counters are updated to the global count, but not to update the local counters.

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

In terms of performance, this should be much faster since locks are used less frequently. This leads to better performance (less serialization and less lock contention).

On the other hand, there are periods of time where the global counters do not represent the latest count exactly since we are batching updates to them. This is probably ok for a voting application, but for an application like a stock market ticker we need updates to be in near real-time. Differences of msec can lead to huge opportunities for day trading software.

Here is an interesting read: http://queue.acm.org/detail.cfm?id=2536492
III. Traffic in Manhattan goes around a block as shown in the figure below.

![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.*