Homework 2 for CS153 (Winter 2020)

Instructions:

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

   a. (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 most straightforward implementation is to assume that the ingredients are threads and the baker is a separate thread. The ingredients can communicate to the baker either through shared variables or more directly using a semaphore. Here is example pseudo code for the implementation using semaphores.

   Semaphore icing(0), cake(0), filling(0);  //semaphore for each ingredient initially empty

   //icing arrived. Threads for cake and filling identical with the corresponding semaphore

   icingThread() {
       icing.signal();  //signal that we have icing
   }

   baker() {
       icing.wait();
       filling.wait();
       cake.wait();  //order does not matter
       makeCake();
   }

   Here is an alternative implementation with shared variables, and integrating the baker function into the ingredients thread (basically, we check if we have enough of the other two when we arrive and bake if we do). The key to this implementation 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
}
}
cake() { //cake mix (and filling) code executed by cake mix (and filling) threads

cakeArrived.signal(); //let icing know we are here

cake.wait(); //wait until we are let go
}

You should get credit for any of these, and I suspect there are other valid implementations.

b. (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 wait twice for the cake mix semaphore. In the shared variable implementation, we 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) (2 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.

(2 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.

[Image: 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. An OS uses a multiple level feedback scheduler with 2 round-robin levels. The quantum for the two levels are 1 and 8 respectively. Assume that we have 5 jobs that arrive at intervals of 5 msec starting at time 0. Assume that once a quantum starts, it runs until it ends (or the process finishes); it is not interrupted by a newly arriving process. The processes have a burst lengths 17, 5, 1, 11, and 8.

a. (4 points) Show the scheduling timeline (which process runs at what times) for the processes until the end.

In this problem, there were some aspects that were underspecified. It did not say how many rounds you will spend in the top level, and whether there is a third level beyond those two. We will accept and work with your solution provided assumptions are stated clearly. In this key, I will assume 1 round in the top queue, and then we remain in the second queue until we are done (i.e., there is no third queue).

Time 0: process 1 arrives, gets scheduled in queue 1 for 1 msec. Then gets demoted to lower queue. It has 16 remaining seconds.

Time 1: process 1 gets scheduled for 8 milliseconds in the lower queue, ending at time 9 (8 remaining for process 1). Meanwhile process 2 arrives at time 5.

Time 9: process 2 gets scheduled for 1 second in top queue. Has 4 seconds left.

Time 10: Process 3 arrives to top queue. Bottom queue has p1 ahead of p2. We schedule process 3 for 1 second (it finishes execution)

Time 11: P1 gets scheduled (top of bottom queue, top queue is empty) for 8 seconds (will finish after 8).

Time 19: P1 finishes, P4 arrived at time 15 and is in top queue. Bottom queue has P2>P1. Schedule P4.

Time 20: P4 finishes and is demoted (has 10 left), P5 arrives and is in top queue. Bottom queue has P2>P1>P4. P5 arrives and is scheduled.

Time 21: P5 finishes and is demoted (has 7 left). Bottom queue has P2>P1>P4>P5. P2 is scheduled (has 4 msec left).

Time 25: P2 finishes and is done, schedule P1 (has 8 msec left).

Time 33: P1 finishes and is done, schedule P4
Time 41: P4 finishes (has 2 left), schedule P5

Time 48: P5 finishes after 7msec and is done. Schedule P4

Time 50: P4 is done.

P1 [0-9], P2 [9-10], P3 [10-11], P1 [11-19], P4 [19-20], P5 [20-21], P2 [21-25], P1 [25-33], P4 [33-41], P5 [41-48], P4 [48-50]

(b) (2 points) Compute the average response time for each process (average time a process waits every time it is stopped until it sees the CPU again).

As defined in the problem:

P1 wait periods: 0 (time 0 arrive to 0 schedule), 2 (waiting from 9 to 11), 14 (waiting from 19 to 33). Average is 16/3 or 5.3333

P2 wait periods: 4 (arrive 5 scheduled 9), 11 (waiting 10 to 21). Average is 7.5

P3: 0 (arrive 10, scheduled 10). Average 0

P4: 4 (arrive 15, run 19), 13 (20 to 33). Average 8.5

P5: 0 (arrive 20, run 20), 21 (21 to 41). Average 10.5