CS 318/418/618, Fall 2021

Homework #1

Ryan Huang huang@cs.jhu.edu

1. Microsoft .NET provides a synchronization primitive called a `CountdownEvent`. Programs use `CountdownEvent` to synchronize on the completion of many threads (similar to `CountDownLatch` in Java). A `CountdownEvent` is initialized with a count, and a `CountdownEvent` can be in two states, nonsignalled and signalled. Threads use a `CountdownEvent` in the nonsignalled state to Wait (block) until the internal count reaches zero. When the internal count of a `CountdownEvent` reaches zero, the `CountdownEvent` transitions to the signalled state and wakes up (unblocks) all waiting threads. Once a `CountdownEvent` has transitioned from nonsignalled to signalled, the `CountdownEvent` remains in the signalled state. In the nonsignalled state, at any time a thread may call the Decrement operation to decrease the count and Increment to increase the count. In the signalled state, Wait, Decrement, and Increment have no effect and return immediately.

   (a) Use pseudo-code to implement a thread-safe CountdownEvent using locks and condition variables by implementing the following methods:

   ```
   class CountdownEvent {
           ...private variables...
           CountdownEvent (int count) { ... }
           void Increment () { ... }
           void Decrement () { ... }
           void Wait () { ... }
   }
   ```

   Notes:

   - The CountdownEvent constructor takes an integer count as input and initializes the CountdownEvent counter with count. Positive values of count cause the CountdownEvent to be constructed in the nonsignalled state. Other values of count will construct it in the signalled state.

- Increment increments the internal counter.
- Decrement decrements the internal counter. If the counter reaches zero, the CountdownEvent transitions to the signalled state and unblocks any waiting threads.
- Wait blocks the calling thread if the CountdownEvent is in the nonsignalled state, and otherwise returns.
- Each of these methods is relatively short.

(b) Semaphores also increment and decrement. How do the semantics of a CountdownEvent differ from a Semaphore?

2. A common pattern in parallel scientific programs is to have a set of threads do a computation in a sequence of phases. In each phase i, all threads must finish phase i before any thread starts computing phase $i + 1$. One way to accomplish this is with barrier synchronization. At the end of each phase, each thread executes `Barrier::Done(n)`, where n is the number of threads in the computation. A call to `Barrier::Done` blocks until all of the n threads have called `Barrier::Done`. Then, all threads proceed. You may assume that the process allocates a new Barrier for each iteration, and that all threads of the program will call Done with the same value.

   (a) Implement a Barrier using a `CountdownEvent` in the previous exercise.

   (b) Write a monitor that implements Barrier using Mesa semantics.

```
monitor Barrier {
  ...
}
```

   (c) Implement Barrier using an explicit lock and condition variable.

```
class Barrier {
  ...private variables...
  void Done (int n) {
    ...
  }
  ...
}
```

3. Consider a problem in which there is a producer p and two consumers c1 and c2. The producer produces pairs of values $< a, b >$. The producer

does not have to wait in Put for a consumer, and the monitor will have to accumulate the values in auxiliary data structures to ensure nothing gets lost (you can assume the use of lists or arrays). Assume that Put can accumulate at most k pairs of values. Consumer c1 consumes the a values of these pairs and c2 consumes the b values of these pairs. A consumer consumes only one value per call.

> Hint: This problem is very similar to the producer/consumer problem-it just so happens that objects are produced in pairs, and each part of a pair is consumed individually.

Write a Mesa-style monitor for this problem. It should have three entry methods: `void Put(int a, b)` that p would use to produce values, `int GetA(void)` that c1 would use to consume a values, and `int GetB(void)` that c2 would use to consume b values. For synchronization, you should only use condition variables.

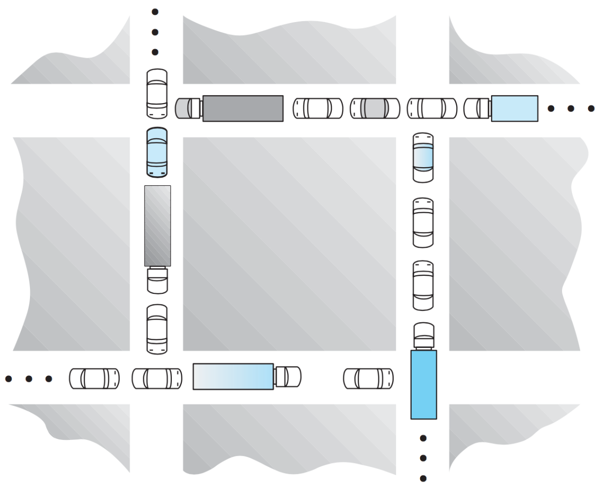
An example sequence of calls could be:

```
Put(10,20)
GetA() -> returns 10
Put(300,400)
GetA() -> returns 300
GetB() -> returns 20
GetA() blocks the caller
```

4. Demonstrate that monitors and semaphores are equivalent so they can be used to implement the same types of synchronization problems.

5. [Silberschatz] Windows Vista provides a new lightweight synchronization tool called a *slim reader–writer* (SRW) lock. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers and do not order waiting threads in a FIFO queue. Explain the benefits of providing such a synchronization tool.

6. [Silberschatz] Consider the traffic deadlock depicted in the following figure.



a) Show that the four necessary conditions for deadlock indeed hold in this example.

b) State a simple rule that will avoid deadlocks in this system

7. [Silberschatz] A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if a northbound and a southbound farmer get on the bridge at the same time. (Vermont farmers are stubborn and are unable to back up.) Using semaphores and/or mutex locks, design an algorithm in pseudocode that prevents deadlock.

   (a) Using exactly one semaphore, design an algorithm that prevents deadlock. Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

   (b) Modify your solution so that it is starvation-free.

8. [Silberschatz] Consider the variation of the Dining Philosophers problem (See Section 31.6 of the OSTEP textbook for a description of the problem), where all unused forks are placed in the center of the table and any philosopher can eat with any two forks. Assume that requests for forks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of forks to philosophers.

9. Annabelle, Bertrand, Chloe and Dag are working on their term papers in CS 318, which is a 10,000 word essay on My All-Time Favorite Race Conditions. To help them work on their papers, they have one dictionary, two copies of Roget's Thesaurus, and two coffee cups.

   Annabelle needs to use the dictionary and a thesaurus to write her paper;

   - Bertrand needs a thesaurus and a coffee cup to write his paper;
   - Chloe needs a dictionary and a thesaurus to write her paper;
   - Dag needs two coffee cups to write his paper (he likes to have a cup of regular and a cup of decaf at the same time to keep himself in balance).

   Consider the following state:

   - Annabelle has a thesaurus and need the dictionary.
   - Bertrand has a thesaurus and a coffee cup.
   - Chloe has the dictionary and needs a thesaurus.
   - Dag has a coffee cup and needs another coffee cup.
     - Is the system deadlocked in this state? Explain using a resource allocation graph.
     - Is this state reachable if the four people allocated and released their resources using the Banker's algorithm? Explain.