Solutions Manual

AUTHOR NAME

November 5, 2016

Contents

PART I

LECTURE SLIDES

CHAPTER 9

MONITORS AND CONDITION VARIABLES

WHAT'S WRONG WITH *semaphors*?

- are shared global variables
- no linguistic connection between semaphores and data they control
- can be accessed from anywhere
- dual purposed (mutex and sched constraints)
- no guarantee of proper usage

Solution: use a higher level construct

MONITORS

A monitor is similar to a class that ties data/operations and synchornization together.

They differ from classes by guaranteeing mutual exlusion and requiring all data to be private.

Definition.

1. (From Wikipedia) A *monitor* is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true.
2. (From slides) A *monitor* is a defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data.
    - Monitors use a *lock* to ensure that only a single thread is active in the monitor at a given time
    - The *lock* also provides mutual exclusion for shared data
    - *Condition variables* enable threads to go to sleep inside the critical sections, by releasing their lock at the same time it puts the thread to sleep

Monitor Operations:

- Encapsulates shared date to protect
- Acquires the mutex at start
- Operates on the shared data
- Temporarily release mutex if it can't complete
- Reqcquires the mutex when it can continue
- Releases the mutex at the end

Implementing Monitors in Java

It is simple to turn a Java class into a monitor:

- Make all data private
- Make all methods synchronized (or at least the non-private ones)

```
class Queue{
    private data;     // queue data

    public void synchronized Add(Object item) {
        put item on queue;
    }

    public void synchronized Remove(){
        if (queue not empty){
            remove item;
            return item;
        }
    }
}
```

More on Monitors – From Wikipedia

An alternate definition of a monitor:

Definition. A *monitor* is a thread-safe class, object, or module that uses wrappted mutual exclusion in order to safely allow acess to a method or viariable by more than one thread.

The defining characteristic of a monitor is that its methods are executed with mutual exclusion. Thus we have an
Invariant: at each point in time, at most one thread may be executing any of a monitors methods.
Monitors were invented by Per Brinch Hansen and C. A. R. Hoare, and were first implemented in Brinch Hansen's Concurrent Pascal language.

Mutual Exclusion

While a thread is executing a method of a thread-safe object, it is said to *occupy* the object by holding its mutex. Thread-safe objects are implemented to enforce that at *each point in time*, at most one thread may occupy the object. When a thread calls a thread-safe object's method it mus twait until no other thread is occupying the thread-safe object.

```
class Account {
  private lock myLock

  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      if balance < amount {
        return false
      } else {
        balance := balance - amount
        return true
      }
    } finally {
      myLock.release()
    }
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    myLock.acquire()
    try {
      balance := balance + amount
    } finally {
      myLock.release()
    }
  }
}
```

Figure 9.3.1: An implementation of a class with mutual exclusion

CONDITION VARIABLES

Question: How can we change `remove()` to wait until something is on the queue?

- Logically, we want to go to sleep inside the critical section.
- But if we hold on to the lock and sleep, then other threads cannot access shared queue, add an item to it, and wake up the sleeping thread.
- **THREAD COULD SLEEP FOREVER**

Solution: use condition variables

- Condition variables enable a thread to sleep inside a critical section
- Any lock held by the thread is atomically released when the thread is put to sleep.

Operations on Condition Variables

Definition. A *condition variable* is a queue of threads waiting for something inside a critical section.

Condition variables support three operations:

1. `Wait()`: atomic (release lock, go to sleep). When the process wakes up it re-acquires the lock
2. `Signal()`: wake up waiting thread, if one exists.
3. `Broadcast()`: wake up all waiting threads.

Invariant: a thread must hold the lock while doing condition variable operations.
In java, we use `wait()` to give up the lock, `notify()` to signal that the condition a thread is waiting on is satisfied, `notifyAll()` to wake up all waiting threads. Effectively there is one condition variable per object.

```
class Queue{
    private Object[] queue;    // queue data

    public void synchronized Add(Object item) {
        queue.append(item);
        notify();
    }

    public Object synchronized Remove(){
        while (queue.size == 0)
            wait();           // Give up lock and go to sleep

        remove and return item;
    }
}
```

More on Condition Variabiables – From Wikipedia

Often, mutual exclusion is not enough. Threads attempting an operation may need to wait until some condition P holds true. Busy waiting (ie, `while (not P) continue`; since mutual exclusion will stop any other thread from updating P. There are other solutions but they have shortfalls (loop and release, for example).
A classic example is the Producer/Consumer problem.

```
global RingBuffer queue; // A thread-unsafe ring-buffer of tasks.

// Method representing each producer thread's behavior:
public method producer(){
    while(true){
        task myTask=...; // Producer makes some new task to be added.
        while(queue.isFull()){} // Busy-wait until the queue is non-full.
        queue.enqueue(myTask); // Add the task to the queue.
    }
}

// Method representing each consumer thread's behavior:
public method consumer(){
    while(true){
        while (queue.isEmpty()){} // Busy-wait until the queue is non-empty.
        myTask=queue.dequeue(); // Take a task off of the queue.
        doStuff(myTask); // Go off and do something with the task.
    }
}
```

Figure 9.4.1: The classic Producer/Consumer problem — this code has a serious problem in that accesses to the queue can be interrupted and interleaved with other threads accesses to the queue. In particular, the `queue.enqueue()` and `queue.dequeue()` methods will likely have instructions to update the queue's member variables such as size, start and end positions, etc.

Mesa vs Hoare Style Monitors

Question: What should happen when `signal()` is called?

- **If there are no waiting threads:** No waiting threads $\Longrightarrow$ the signaler continues and the signal is lost (different from behavior with semaphores)
- **If there is a waiting thread:** one of the threads starts executing, others must wait.

Mesa-style: *(Nachos, Java, most real OSs)*

- The thread that signals keeps the lock (and thus the processor)

- The waiting thread waits for the lock

There are two main styles of Monitor implementations. **Hoare-style:** *(Most textbooks)*

- The thread that signals gives up the lock and the waiting thread gets the lock
- When the thread that was waiting and is no executing exits or waits again, it releases the lock back to the signaling thread.

PART II

NOTES FROM TEXT

Index