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;
        }
    }
}
```

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;
    }
}
```

Part II

Notes From Text

INDEX