Threads in Java Implementing Monitors

EPL222 – Lab8

Producer – Consumer

```java
class Producer implements Runnable {
    private final Queue<Integer> queue;
    private final int maxSize;
    public Producer(Queue<Integer> queue, int maxSize) {
        this.queue = queue; this.maxSize = maxSize;
    }
    public void run() {
        int i = 0;
        while (true) {
            synchronized (queue) {
                while (queue.size() == maxSize) {
                    try {
                        System.out.println(Thread.currentThread().getName() +
                                                    ": Buffer is full");
                        queue.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                i++;
                System.out.println(Thread.currentThread().getName()  +
                                                    ": Producing value: " + i);
                queue.add(i);
                queue.notifyAll();
            }
        }
    }
}
```

Producer

```java
class Consumer implements Runnable {
    private final Queue<Integer> queue;

    public Consumer(Queue<Integer> queue) {
        this.queue = queue;
    }

    public void run() {
        int i;
        while (true) {
            synchronized (queue) {
                while (queue.isEmpty()) {
                    try {
                        queue.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                i = queue.remove();
                System.out.println(Thread.currentThread().
                        getName() + ": Consuming value: " + i);
                queue.notifyAll();
            }
        }
    }
}
```

Consumer

Producer – Consumer (continued)

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        final Queue<Integer> buffer = new LinkedList<>();
        int maxsize = 100, producers = 3, consumers = 5;
        Thread t[] = new Thread[producers+consumers];

        for (int i = 0; i < producers; i++) {
            t[i] = new Thread(new Producer(buffer, maxsize), "Producer " + i);
        }
        for (int i = 0; i < consumers ; i++) {
            t[producers+i] = new Thread(new Consumer(buffer), "Consumer " + i);
        }
        for (int i = 0; i < t.length; i++) {
            t[i].start();
        }
    }
}
```

Problems with previous solution

- This is **NOT a monitor**!
  - Two separate classes that implement the producer and consumer threads
  - The used resource (the *queue*) is not protected from the threads
    - instead the threads have direct access to them!
  - Threads take care of their concurrency!
- Does it work?
  - The provided solution is correct as far as concurrency is concerned
  - This is because all producers and consumers threads synchronize on the same object (there is only one *queue* object) but each thread executes on a different object!
    - Each producer thread continues producing from the last value that thread produced!
- We have no way to wake a producer or a consumer!
  - Both consumers and producers wait on the same object!
  - We have to use `queue.notifyAll()`

Producer – Consumer (continued)

```
public class ProducerConsumerExample2 {
    public static void main(String[] args) {
        final Queue<Integer> buffer = new LinkedList<>();
        int maxsize = 10, producers = 3, consumers = 5;
        Thread t[] = new Thread[producers + consumers];

        Producer p = new Producer(buffer, maxsize);
        Consumer c = new Consumer(buffer);

        for (int i = 0; i < producers; i++) {
            t[i] = new Thread(p, "Producer " + i);
        }
        for (int i = 0; i < consumers; i++) {
            t[producers + i] = new Thread(c, "Consumer " + i);
        }
        for (int i = 0; i < t.length; i++) {
            t[i].start();
        }
    }
}
```

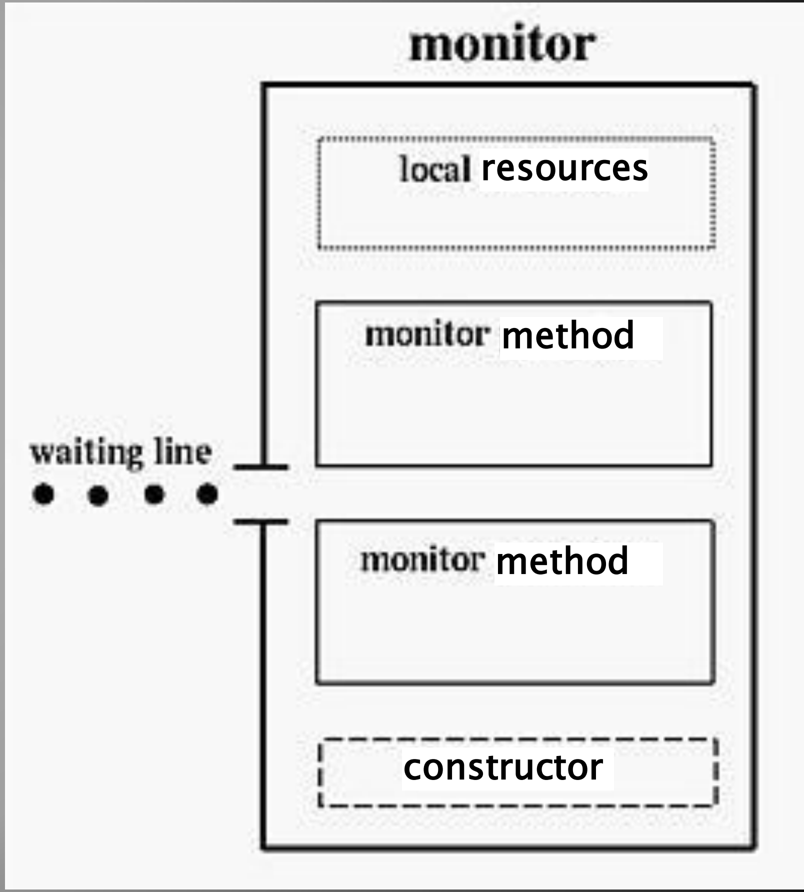
Only 1 producer and 1 consumer object!

Incorrect concurrency!

All producer threads share the same variable to produce values!

Implementing a Monitor

- Remember that a monitor has some local resources, monitor procedures (methods) and an initialization part (constructor)
- In Java a monitor can be a class
  - The public methods of the class form the boundary of the monitor
  - Making them synchronized guarantees that, at any time, there can only be **one thread executing within the monitor** boundary



Implementing a Monitor in Java

- To implement a monitor in Java, we need to address the same three questions as in C:
  - (1) how do we make sure that the local variables/objects will not be accessed by non-monitor procedures
  - (2) how do we make sure that the user can only "see" the interface without knowing the details of the monitor
  - (3) how do we properly setup the monitor boundary so that mutual exclusion can be guaranteed
- Questions (1) and (2) have a simple solution
  - Just make everything beyond the monitor methods private!
  - Or define an interface with just the monitor methods
    - The implantation class can have more (private) methods and/or local variables
- Question (3) has a natural answer: make the monitor methods synchronized!

Producer Consumer Monitor – 1st attempt

Producer

```java
public class Producer implements Runnable {
   private final PCMonitor m;

   public Producer(PCMonitor m) {
      this.m = m;
   }

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread()
                           .getName() + ": Producing value: " +
                           m.produce());
         try {
            if (Math.random() > 0.5)
               Thread.sleep((int) (Math.random()*1000)%1000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Consumer

```java
public class Consumer implements Runnable {
   private final PCMonitor m;

   public Consumer(PCMonitor m) {
      this.m = m;
   }

   public void run() {
      while (true) {
         System.out.println(Thread.currentThread()
                           .getName() + ": Consuming value: " +
                           m.consume());
         try {
            if (Math.random() > 0.5)
               Thread.sleep((int) (Math.random()*1000)%1000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Producer Consumer Monitor – 1st attempt

```java
public interface PCMonitor {

int produce();
int consume();

}
```

```java
import java.util.LinkedList;
import java.util.Queue;

public class PCMonitorImpl
                implements PCMonitor {
    private final Queue<Integer> queue;
    private final int maxSize;
    private int value = 0;

    public PCMonitorImpl(int maxSize) {
        this.queue = new LinkedList<>();
        this.maxSize = maxSize;
    }
```

```java
    public synchronized int produce() {
        while (queue.size() == maxSize) {
            try {
                System.out.println(Thread.currentThread().getName()+": Buffer full");
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        queue.add(++value);
        notifyAll();
        return value;
    }

    public synchronized int consume() {
        int i;
        while (queue.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        i = queue.remove();
        notifyAll();
        return i;
    }
}
```

Monitor

Producer Consumer Monitor – 1st attempt

```java
public class PCMExample {
    public static void main(String[] args) {
        int maxSize = 10, producers = 3, consumers = 5;

        PCMonitor m = new PCMonitorImpl(maxSize);
        Thread t[] = new Thread[producers + consumers];

        for (int i = 0; i < producers; i++) {
            t[i] = new Thread(new Producer(m), "Producer " + i);
        }
        for (int i = 0; i < consumers; i++) {
            t[producers + i] = new Thread(new Consumer(m), "Consumer " + i);
        }
        for (int i = 0; i < t.length; i++) {
            t[i].start();
        }
    }
}
```

Problems with 1st monitor attempt

- Now it is **a monitor**!
  - The classes that implement the producer and consumer threads are separate from the monitor
  - The used resource (the *queue*) NOT accessible from the threads
  - The monitor class takes care of concurrency!
- We have no way to wake a producer or a consumer!
  - Both consumers and producers wait on the same object – **the monitor object**!
  - We must use `notifyAll()`

Condition variables in Java
Interface Condition

- Condition factors out the Object monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations
  - A Lock replaces the use of synchronized methods and statements
  - A Condition replaces the use of the Object monitor methods.
  - It is Java's equivalent to *condition variables*
    - As always access to a condition must be protected, so a lock of some form must be associated with it
    - The key property that waiting for a condition provides is that it *atomically* releases the associated lock and suspends the current thread, just like `Object.wait`
- A Condition instance is intrinsically bound to a lock
  - To obtain a Condition instance for a particular Lock instance use its `newCondition()` method

The Lock interface

- Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements
  - They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects
  - Only one thread at a time can acquire a given *lock instance*
- The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way
  - If multiple locks are acquired they must be released in the opposite order
  - All locks must be released in the **same lexical scope in which they were acquired**
  - There are occasions where you need to work with locks in a more flexible way
    - E.g., the use of "hand-over-hand" or "chain locking": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on
    - Implementations of the Lock interface enable the use of such techniques by allowing a lock to be acquired and released **in different scopes**, and allowing multiple locks to be **acquired and released in any order**

The Lock interface

- With this increased flexibility comes additional responsibility
  - The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements – the following idiom should be used:

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

  - When locking and unlocking occur in different scopes, care must be taken to ensure that all code that is executed while the lock is held is protected by try-finally or try-catch to ensure that the lock is released when necessary.
  - Lock implementations provide additional functionality over the use of synchronized methods and statements by providing a non-blocking attempt to acquire a lock (`tryLock()`, `tryLock(long, TimeUnit)`)
- Note that Lock instances are just normal objects and can themselves be used as the target in a synchronized statement
  - Acquiring the monitor lock of a Lock instance has no specified relationship with invoking any of the lock() methods of that instance
  - It is recommended that to avoid confusion you **never use Lock instances in this way**

Condition and Lock Interfaces

Interface Condition

Type	Method and Description
void	`await()` Causes the current thread to wait until it is signalled or interrupted.
boolean	`await(long time, TimeUnit unit)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	`awaitNanos(long nanosTimeout)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
void	`awaitUninterruptibly()` Causes the current thread to wait until it is signalled.
boolean	`awaitUntil(Date deadline)` Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
void	`signal()` Wakes up one waiting thread.
void	`signalAll()` Wakes up all waiting threads.

Interface Lock

Type	Method and Description
void	`lock()` Acquires the lock.
void	`lockInterruptibly()` Acquires the lock unless the current thread is interrupted.
Condition	`newCondition()` Returns a new Condition instance that is bound to this Lock instance.
boolean	`tryLock()` Acquires the lock only if it is free at the time of invocation.
boolean	`tryLock(long time, TimeUnit unit)` Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.
void	`unlock()` Releases the lock.

All Known Implementing Classes:
`ReentrantLock`, `ReentrantReadWriteLock.ReadLock`, `ReentrantReadWriteLock.WriteLock`

Producer Consumer Monitor – 2nd attempt

```java
public interface PCMonitor {
    int produce();
    int consume();
}
```

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class PCMonitorImp2 implements PCMonitor {
    private final Lock lock = new ReentrantLock();
    private final Condition qFull = lock.newCondition();
    private final Condition qEmpty = lock.newCondition();

    private final Queue<Integer> queue;
    private final int maxSize;
    private int value = 0;

    public PCMonitorImp2(int maxSize) {
        this.queue = new LinkedList<>();
        this.maxSize = maxSize;
    }
```

Monitor

Producer Consumer Monitor – 2nd attempt

```
public int produce() {
    lock.lock();
    try {
        while (queue.size() == maxSize) {
            System.out.println(Thread.currentThread()
                    .getName() + ": Buffer is full");
            qFull.await();
        }
        queue.add(++value);
        qEmpty.signal();
        return value;
    } catch (InterruptedException e) {
        e.printStackTrace();
        return -1;
    } finally {
        lock.unlock();
    }
}
```

```
public int consume() {
    lock.lock();
    int i = 0;
    try {
        while (queue.isEmpty())
            qEmpty.await();
        i = queue.remove();
        qFull.signal();
        return i;
    } catch (InterruptedException e) {
        e.printStackTrace();
        return -1;
    } finally {
        lock.unlock();
    }
}
}
```