Java Monitor Object
Usage Considerations

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Lesson

- Appreciate Java monitor object usage considerations
Learning Objectives in this Lesson

• Appreciate Java monitor object usage considerations
  • In particular, know common traps & pitfalls of Java’s built-in monitor objects
Usage Considerations of Java Monitor Objects
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects.
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
- Monitor objects are limited

![Diagram of producer and consumer with synchronized methods](image)

- **Producer**
  - `put()`
  - `take()`

- **Consumer**
  - `put()`
  - `take()`

- **SimpleBlockingBoundedQueue**
  - `synchronized put()`
  - `synchronized take()`

- **Wait Queue**
  - `wait()`
  - `notify()`
  - `notifyAll()`

- **Entrance Queue**
  - `<<contains>>`
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited, e.g.
    - No non-blocking, timed, or interruptible synchronizers

See lessons on "Java ReentrantLocks" for examples of these capabilities
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited, e.g.
    • No non-blocking, timed, or interruptible synchronizers
    • Only one wait queue & one entrance queue

See www.dre.vanderbilt.edu/~schmidt/C++2Java.html#concurrency
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited, e.g.
    • No non-blocking, timed, or interruptible synchronizers
    • Only one wait queue & one entrance queue
    • May yield “nested monitor lockout”

```java
public class BuggyLock {
    Object mMonObj = new Object();
    boolean mLocked;

    synchronized void lock() {
        while (mLocked)
            synchronized (mMonObj)
                mMonObj.wait();
        mLocked = true;
    }

    synchronized void unlock() {
        mLocked = false;
        synchronized (mMonObj)
            mMonObj.notify();
    }
}
```

lock() is a synchronized method

See tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited, e.g.
    • No non-blocking, timed, or interruptible synchronizers
    • Only one wait queue & one entrance queue
    • May yield “nested monitor lockout”

```java
public class BuggyLock {
  Object mMonObj = new Object();
  boolean mLocked;

  synchronized void lock() {
    while (mLocked)
      synchronized (mMonObj)
        mMonObj.wait();
    mLocked = true;
  }

  synchronized void unlock() {
    mLocked = false;
    synchronized (mMonObj)
      mMonObj.notify();
  }

  // BuggyLock monitor lock is still held here, so unlock() never runs!
}
```

BuggyLock monitor lock is still held here, so unlock() never runs!
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited, e.g.
    - No non-blocking, timed, or interruptible synchronizers
    - Only one wait queue & one entrance queue
    - May yield “nested monitor lockout”
    - Doesn’t support “two lock queue” optimizations

```java
class LinkedBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, ...

/** Lock held by take, poll, etc */
private final ReentrantLock takeLock =
    new ReentrantLock();

/** Lock held by put, offer, etc */
private final ReentrantLock putLock =
    new ReentrantLock();
```

See src/share/classes/java/util/concurrent/LinkedBlockingQueue.java
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited, e.g.
    • No non-blocking, timed, or interruptible synchronizers
    • Only one wait queue & one entrance queue
  • Synchronized statements only support scoped locking

synchronized(this) {
  ...
  // this lock is always
  // released at the
  // end of this block
}

Scoped locking is inefficient for certain concurrent algorithms,
e.g., it may require redundant checks for internal state(s)
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited, e.g.
    - No non-blocking, timed, or interruptible synchronizers
    - Only one wait queue & one entrance queue
    - Synchronized statements only support scoped locking
    - No support for sensible timed waits...

See [stackoverflow.com/questions/3397722/how-to-differentiate-when-waitlong-timeout-exit-for-notify-or-timeout](https://stackoverflow.com/questions/3397722/how-to-differentiate-when-waitlong-timeout-exit-for-notify-or-timeout)
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between `notify()` & `notifyAll()` is tricky

See stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited
  • Choosing between `notify()` & `notifyAll()` is tricky

<table>
<thead>
<tr>
<th>Uniform waiters</th>
<th>Only one condition expression that <code>wait()</code> is waiting for is associated with the monitor object &amp; each thread executes the same logic when returning from <code>wait()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>One-in &amp; one-out</td>
<td>A <code>notify()</code> on the monitor object enables at most one thread to proceed</td>
</tr>
</tbody>
</table>

*Conditions under which `notify()` can be used*
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between `notify()` & `notifyAll()` is tricky

<table>
<thead>
<tr>
<th>Uniform waiters</th>
<th>Only one condition expression that <code>wait()</code> is waiting for is associated with the monitor object &amp; each thread executes the same logic when returning from <code>wait()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>One-in &amp; one-out</td>
<td>A <code>notify()</code> on the monitor object enables at most one thread to proceed</td>
</tr>
</tbody>
</table>

*Conditions under which `notify()` can be used*
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects

• Monitor objects are limited

• Choosing between notify() & notifyAll() is tricky
  • Use notify() when possible since it’s more efficient & avoids the “Thundering Herd” problem..

See en.wikipedia.org/wiki/Thundering_herd_problem
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between `notify()` & `notifyAll()` is tricky
    - Use `notify()` when possible since it’s more efficient & avoids the “Thundering Herd” problem..
    - However, `notifyAll()` is often needed since there’s just one wait queue..

A monitor object may need to wait for different condition expression
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between notify() & notifyAll() is tricky
  - Fairness issues arise due to the order in which waiting threads are notified
Usage Considerations of Java Monitor Objects

• Programmers must be aware of issues with Java monitor objects
  • Monitor objects are limited
  • Choosing between notify() & notifyAll() is tricky
  • Fairness issues arise due to the order in which waiting threads are notified
    • Monitor object’s implement “haphazard notification” to optimize performance
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between notify() & notifyAll() is tricky
  - Fairness issues arise due to the order in which waiting threads are notified
    - Monitor object’s implement “haphazard notification” to optimize performance
  - The *Specific Notification* pattern can be applied to control ordering

See [www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf)
Usage Considerations of Java Monitor Objects

- Programmers must be aware of issues with Java monitor objects
  - Monitor objects are limited
  - Choosing between `notify()` & `notifyAll()` is tricky
  - Fairness issues arise due to the order in which waiting threads are notified
    - Monitor object’s implement “haphazard notification” to optimize performance
  - The *Specific Notification* pattern can be applied to control ordering
    - i.e., programmatically choose a particular thread to run from a family of waiting threads
Usage Considerations of Java Monitor Objects

- In practice, you often need more than Java’s monitor object mechanisms

java.util.concurrent & java.util.concurrent.locks

package

java.util.concurrent.locks

Added in API level 1

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax. The Lock interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is ReentrantLock.

package

java.util.concurrent

Added in API level 1

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the java.util.concurrent.locks and java.util.concurrent.atomic packages.

Usage Considerations of Java Monitor Objects

- In practice, you often need more than Java’s monitor object mechanisms
  - java.util.concurrent & java.util.concurrent.locks
  - e.g., ReentrantLock & ConditionObject

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>, java.io.Serializable {

    final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;

    ...
In practice, you often need more than Java’s monitor object mechanisms

- java.util.concurrent & java.util.concurrent.locks
- Android concurrency frameworks

See developer.android.com/guide/components/processes-and-threads.html#Threads
Usage Considerations of Java Monitor Objects

- In practice, you often need more than Java’s monitor object mechanisms
  - java.util.concurrent & java.util.concurrent.locks
- Android concurrency frameworks
  - Message passing may avoid need for monitor objects & synchronization altogether

End of Java Monitor Object
Usage Considerations