Java Monitor Objects: Coordination (Part 1)

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 Part of the Module

- Understand how Java built-in monitor objects provide waiting & notification mechanisms that coordinate threads running in a concurrent program

Critical Section

1. Enter monitor object
2. Acquire lock
3. wait()
4. notifyAll()
5. Release lock
6. Leave monitor object

Java Built-in Waiting & Notification Mechanisms
Java Built-in Waiting & Notification Mechanisms

Java synchronized methods & statements only provide a partial solution to concurrent programs.
Java Built-in Waiting & Notification Mechanisms

- Java build-in monitor objects allow threads to coordinate their interactions.
Java Built-in Waiting & Notification Mechanisms

- Java build-in monitor objects allow threads to coordinate their interactions via the `wait()`, `notify()`, & `notifyAll()` methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait()</code></td>
<td>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.</td>
</tr>
<tr>
<td><code>notify()</code></td>
<td>Wakes up a single thread that is waiting on this object's monitor.</td>
</tr>
<tr>
<td><code>notifyAll()</code></td>
<td>Wakes up all threads that are waiting on this object's monitor.</td>
</tr>
</tbody>
</table>

See docs.oracle.com/javase/8/docs/api/java/lang/Object.html
Java Built-in Waiting & Notification Mechanisms

- Java build-in monitor objects allow threads to coordinate their interactions
- via the `wait()`, `notify()`, & `notifyAll()` methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait()</code></td>
<td>Causes the current thread to wait until another thread invokes the <code>notify()</code> method or the <code>notifyAll()</code> method for this object</td>
</tr>
<tr>
<td><code>notify()</code></td>
<td>Wakes up a single thread that is waiting on this object's monitor</td>
</tr>
<tr>
<td><code>notifyAll()</code></td>
<td>Wakes up all threads that are waiting on this object's monitor</td>
</tr>
</tbody>
</table>
Java Built-in Waiting & Notification Mechanisms

• Java build-in monitor objects allow threads to coordinate their interactions
  • via the wait(), notify(), & notifyAll() methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void wait()</td>
<td>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object</td>
</tr>
<tr>
<td>void notify()</td>
<td>Wakes up a single thread that is waiting on this object's monitor</td>
</tr>
<tr>
<td>void notifyAll()</td>
<td>Wakes up all threads that are waiting on this object's monitor</td>
</tr>
</tbody>
</table>
Java Built-in Waiting & Notification Mechanisms

- Java build-in monitor objects allow threads to coordinate their interactions
  - via the wait(), notify(), & notifyAll() methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void wait()</td>
<td>Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object</td>
</tr>
<tr>
<td>void notify()</td>
<td>Wakes up a single thread that is waiting on this object's monitor</td>
</tr>
<tr>
<td>void notifyAll()</td>
<td>Wakes up all threads that are waiting on this object's monitor</td>
</tr>
</tbody>
</table>
Java Built-in Waiting & Notification Mechanisms

- Java build-in monitor objects allow threads to coordinate their interactions
  - via the `wait()`, `notify()`, & `notifyAll()` methods

```java
void wait() – Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object

void notify() – Wakes up a single thread that is waiting on this object's monitor

void notifyAll() – Wakes up all threads that are waiting on this object's monitor
```

See [en.wikipedia.org/wiki/Thundering_herd_problem](en.wikipedia.org/wiki/Thundering_herd_problem)
• Java built-in monitor objects have one entrance queue & one wait queue

See en.wikipedia.org/wiki/Monitor_(synchronization) #Implicit_condition_variable_monitors
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue

- Serializes thread access to monitor object’s critical section
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue.

```
Entrance Queue

enter

\[ \Phi \]

notified

Wait Queue

\[ b \]

wait

leave

Critical Section

All threads that call wait() are parked on the wait queue.
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue

- All notify() & notifyAll() calls also apply to the wait queue
Java Built-in Waiting & Notification Mechanisms

• Java built-in monitor objects have one entrance queue & one wait queue, e.g.
  • put() calls wait() when the queue is full

Atomically releases the intrinsic lock & sleeps on the wait queue

```
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...
    public void put(E msg) {
        synchronized(this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    public E take() ... {
        synchronized(this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
```

See en.wikipedia.org/wiki/Guarded_suspension
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue, e.g.
  - put() calls wait() when the queue is full
  - It also calls notifyAll() after adding an item

Wakes up all the threads blocked on the wait queue since waiters are non-uniform

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public void put(E msg){
        synchronized(this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    ...

    public E take() ... {
        synchronized(this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
```
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue, e.g.
  - put() calls wait() when the queue is full
  - It also calls notifyAll() after adding an item

```java
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...
    public void put(E msg) {
        synchronized(this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    public E take() ... {
        synchronized(this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
}
```

notifyAll() is required here due to the limitations of Java built-in monitor objects, which only have one wait queue

See stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue, e.g.
  - put() calls wait() when the queue is full
  - take() calls wait() when the queue is empty

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public void put(E msg) {
        synchronized(this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    ...

    public E take() {
        synchronized(this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
}
```

See [en.wikipedia.org/wiki/Guarded Suspension](en.wikipedia.org/wiki/Guarded_Suspension)

Atomically releases the intrinsic lock & sleeps on the wait queue
Java built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue, e.g.
  - put() calls wait() when the queue is full
  - take() calls wait() when the queue is empty
  - It also calls notifyAll() after removing an item

Wakes up all the threads blocked on the wait queue since waiters are non-uniform

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public void put(E msg) {
        synchronized(this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    ...
    public E take() ... {
        synchronized(this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
}
```

Again, notifyAll() is required here due to the limitations of Java built-in monitor objects, which only have one wait queue
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor objects have one entrance queue & one wait queue

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public void put(E msg) {
        synchronized (this) {
            while (mList.isFull()) wait();
            mList.add(msg);
            notifyAll();
        }
    }
    ...
    public E take() {
        synchronized (this) {
            while (mList.isEmpty()) wait();
            notifyAll();
            return mList.poll();
        }
    }
    ...
}
```

The put() & take() methods are examined further later in this lesson
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor object synchronizers can be implemented with POSIX-like synchronizers.
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor object synchronizers can be implemented w/ POSIX-like synchronizers, e.g.
  - Entrance queue is akin to a POSIX recursive mutex.

See computing.llnl.gov/tutorials/pthreads/#Mutexes
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor object synchronizers can be implemented with POSIX-like synchronizers, e.g.
  - Entrance queue is akin to a POSIX recursive mutex
  - Wait queue is akin to a POSIX condition variable

See computing.llnl.gov/tutorials/pthreads/#ConditionVariables
Java built-in monitor object synchronizers can be implemented with POSIX-like synchronizers, e.g.

- Entrance queue is akin to a POSIX recursive mutex
- Wait queue is akin to a POSIX condition variable
- Similar to Java ConditionObjects

See earlier lesson on “Java ConditionObjects”
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor object synchronizers can be implemented w/POSIX-like synchronizers, e.g.
  - Entrance queue is akin to a POSIX recursive mutex
  - Wait queue is akin to a POSIX condition variable
- The implementation in the Oracle JDK uses lower-level locking primitives

See github.com/JetBrains/jdk8u_hotspot/blob/master/src/share/vm/runtime/objectMonitor.cpp
End of Java Monitor Object: Coordination (Part 1)
Java Monitor Objects:
Coordination (Part 2)

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 Part of the Module

- Learn how to fix a buggy concurrent Java program using Java’s wait & notify mechanisms, which provide coordination.

- Visualize how Java built-in monitor objects can be used to ensure mutual exclusion & coordination between threads running in a concurrent program.

Diagram:

1. Enter monitor object
2. Acquire lock
3. wait()
4. notifyAll()
5. Release lock
6. Leave monitor object

Critical Section
Visual Analysis of the SimpleBlockingBounded Queue Example
Visual Analysis of SimpleBlockingBoundedQueue

1. Enter monitor object
2. Acquire lock
3. wait()
4. notifyAll()
5. Release lock
6. Leave monitor object

Critical Section

See github.com/douglasraigschmidt/POSA/tree/master/ex/M3/Queues/SimpleBlockingBoundedQueue
Visual Analysis of SimpleBlockingBoundedQueue

1. Enter monitor object
2. Acquire lock
3. wait()
4. notifyAll()
5. Release lock
6. Leave monitor object

Queue of threads blocked on the monitor lock’s “entrance queue”

Queue of threads waiting on the monitor condition’s “wait queue”

See en.wikipedia.org/wiki/Monitor_(synchronization) #Implicit_condition_variable_monitors
new Thread(() -> {
    while(true)
    System.out.println(take());
}).start();
Acquire lock

T₁

Critical Section

new Thread(() -> {
    while(true)
    System.out.println(take());
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

```
new Thread(() -> {
    while (true)
        System.out.println(take());
}).start();
```
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

while (mList.isEmpty())
    wait();

new Thread(() -> {
    while (true)
        System.out.println(take());
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

T1

while (mList.isEmpty())
    wait();

new Thread(() -> {
    while (true)
        System.out.println(take());
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

Release lock

Block on monitor condition

new Thread(() -> {
    while(true)
    
    System.out.println(take());
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Enter monitor object

Critical Section

Block on monitor condition

new Thread(() -> {
    for(int i = 0;
        i < 10; i++)
        put(Integer.toString(i));
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

```
new Thread(() -> {
    for(int i = 0; i < 10; i++)
        put(Integer.toString(i));
}).start();
```
new Thread(() -> {
    for(int i = 0; i < 10; i++)
        put(Integer.toString(i));
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

new Thread(() -> {
    for(int i = 0; i < 10; i++)
        put(Integer.toString(i));
}).start();
```
        new Thread(() -> {
            for(int i = 0; i < 10; i++)
                put(Integer.toString(i));
        }).start();
```
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

Release lock

T2

Block on
monitor condition

T1

new Thread(() -> {
    for(int i = 0;
        i < 10; i++)
    put(Integer.
        toString(i));
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

Exit monitor object

new Thread(() -> {
    for(int i = 0; i < 10; i++)
        put(Integer.toString(i));
}).start();

Block on monitor condition

T1

T2
Visual Analysis of SimpleBlockingBoundedQueue

```java
new Thread(() -> {
    while(true)
        System.out.println(take());
}).start();
```
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

Acquire lock

T₁

new Thread(() -> {
    while(true)
        System.out.println(take());
}).start();
SimpleBlockingBoundedQueue

new Thread(() -> {
    while (true)
        System.out.println(take());
}).start();

while (mList.isEmpty())
    wait();
It's ok to call notifyAll() before removing/returning the front item in the queue since the monitor lock is held & only one method can be in the monitor object.
Critical Section

```
notifyAll();
return mList.poll();
```

```
new Thread(() -> {
    while(true)
        System.out.println(take());
}).start();
```
Visual Analysis of SimpleBlockingBoundedQueue

```
SimpleBlockingBoundedQueue

new Thread(() -> {
    while(true)
    System.out.println(take());
}).start();
```
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

Leave monitor object

new Thread(() -> {
    while(true)
    System.out.println(take());
}).start();
End of Java Monitor Object: Coordination (Part 2)
Java Monitor Objects: Coordination (Part 3)

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 Part of the Module

• Learn how to fix a buggy concurrent Java program using Java’s wait & notify mechanisms, which provide coordination

• Visualize how Java built-in monitor objects can be used to ensure mutual exclusion & coordination between threads running in a concurrent program

• Know how to program the SimpleBlockingBoundedQueue in Java
Code Analysis of the SimpleBlockingBounded Queue Example
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    SimpleBlockingBoundedQueue(int capacity)
    {
        mList = new ArrayList<E>();
        mCapacity = capacity;
    }

    ...
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    SimpleBlockingBoundedQueue(int capacity)
    {
        mList = new ArrayList<E>();
        mCapacity = capacity;
    }
    ...

The constructor needn’t be protected against race conditions
A thread can “wait” for a condition in a synchronized method.

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...

    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }
}
```

See en.wikipedia.org/wiki/Guarded_suspension
A thread can “wait” for a condition in a synchronized method.

Code Analysis of SimpleBlockingBoundedQueue

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...

    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }
```

e.g., thread T1 calls take(), which acquires the intrinsic lock & waits while the queue is empty.
A thread can “wait” for a condition in a synchronized method.

e.g., thread $T_1$ calls `take()`, which acquires the intrinsic lock & waits while the queue is empty.

class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...

    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

wait() should be called in a loop that checks whether the condition is true or not.

```java
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...

    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }
```
wait() should be called in a loop that checks whether the condition is true or not.

A thread can’t assume a notification it receives is for its condition expression.

class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    
    public synchronized String take(){
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }

This limitation is a consequence of a Java built-in monitor object only having a single wait queue.
• wait() should be called in a loop that checks whether the condition is true or not
• A thread can’t assume a notification it receives is for its condition expression
• It also can’t assume the condition expression is true!

This is due to the non-determinism of concurrency

class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public synchronized String take()
    {
        while (mList.isEmpty())
            wait();
        final E e = mList.poll();
        notifyAll();
        return e;
    }

wait() should be called in a loop that checks whether the condition is true or not

- A thread can’t assume a notification it receives is for its condition expression
- It also can’t assume the condition expression is true!
- Must also guard against “spurious wakeups”
- A thread might be awoken in wait() although no thread signaled the monitor object

```
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...

    public synchronized String take(){
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }
```

See en.wikipedia.org/wiki/Spurious_wakeup
A thread blocked on wait() won’t continue until it’s notified that the condition expression may be true.

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    
    public synchronized String take(){
        while (mList.isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }
```
A thread blocked on \texttt{wait()} won’t continue until it’s notified that the condition expression may be true.

e.g., thread \texttt{T}_2 calls \texttt{put()}, which acquires the intrinsic lock & adds an item to the queue so it’s no longer empty.

```java
class SimpleBlockingBoundedQueue\<E\> implements BlockingQueue\<E\> {
    ...

    public synchronized void put(E msg) {
        ...
        while (mList.isFull())
            wait();
        mList.add(msg);
        notifyAll();
    }

    private boolean isFull() {
        return mList.size() >= mCapacity;
    }
    ...
```
A thread blocked on wait() won’t continue until it’s notified that the condition expression may be true.

Assuming that thread T1 is blocked in take() the queue won’t be full!

```java
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {
    ...

    public synchronized void put(E msg) {
        ...
        while (mList.isFull())
            wait();
        mList.add(msg);
        notifyAll();
    }

    private boolean isFull() {
        return mList.size() >= mCapacity;
    }
    ...
```
• A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true

```java
class SimpleBlockingBoundedQueue<E>
    implements BlockingQueue<E> {

    ... public synchronized void put(E msg) {
        ... while (mList.isFull())
            wait();

        mList.add(msg);
        notifyAll();
    }

    private boolean isFull() {
        return mList.size() >= mCapacity;
    }
    ...
```

`notifyAll()` is used here due to Java’s monitor object limitations, i.e., there’s only a single wait queue.
• Several steps occur when a waiting thread is notified

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    
    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
```
Several steps occur when a waiting thread is notified
• wakes up & obtains lock

class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public synchronized String take(){
        while (mList.isEmpty())
            wait();
        notifyAll();
        return mList.poll();
    }

    ...
Code Analysis of SimpleBlockingBoundedQueue

Several steps occur when a waiting thread is notified
- wakes up & obtains lock
- re-evaluates the condition expression

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    
    public synchronized String take() {
        while (mList.isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
```
Code Analysis of SimpleBlockingBoundedQueue

- Several steps occur when a waiting thread is notified
  - wakes up & obtains lock
  - re-evaluates the condition expression
  - continues after wait()

class SimpleBlockingBoundedQueue<E>
   implements BlockingQueue<E> {
   ...
   public synchronized String take(){
        while (mList.isEmpty())
                wait();

        notifyAll();
        return mList.poll();
   }

Code Analysis of SimpleBlockingBoundedQueue

- Several steps occur when a waiting thread is notified
  - wakes up & obtains lock
  - re-evaluates the condition expression
  - continues after wait()

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public synchronized String take() {
        while (mList.isEmpty())
            wait();
        notifyAll();
        return mList.poll();
    }
}
```

It’s ok to call `notifyAll()` before removing/returning the front item in the queue since the monitor lock is held & only one method can be in the monitor object
Several steps occur when a waiting thread is notified:

- Wakes up & obtains lock
- Re-evaluates the condition expression
- Continues after `wait()`
- Releases lock when it returns

```java
class SimpleBlockingBoundedQueue<E> implements BlockingQueue<E> {
    ...
    public synchronized String take() {
        while (mList.isEmpty())
            wait();
        notifyAll();
        return mList.poll();
    }
```
End of Java Monitor Object: Coordination (Part 3)