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

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

Critical Section

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/SimpleBlockingBoundedQueue
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.

![Diagram of thread interactions](image_url)
**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>void 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>void notify()</code></td>
<td>Wakes up a single thread that is waiting on this object's monitor.</td>
</tr>
<tr>
<td><code>void 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](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>
</table>
| 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          |
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

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
Java Built-in Waiting & Notification Mechanisms

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

See [en.wikipedia.org/wiki/Monitor_(synchronization) #Implicit_condition_variable_monitors](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

Entrance Queue

Critical Section

Wait Queue

enter

notified

wait

leave

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

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

```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();
        }
    }
    ...
}
```

Atomically releases the intrinsic lock & sleeps on the wait queue

See en.wikipedia.org/wiki/GuardedSuspension
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();
    }
  }
  ...
}
```

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

`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, 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
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

```
public 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();
        }
    }
    ...
```

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

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 are often implemented via POSIX mechanisms.
Java Built-in Waiting & Notification Mechanisms

- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
- Entrance queue can be a POSIX recursive mutex

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

- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
  - Entrance queue can be a POSIX recursive mutex
  - Wait queue can be a POSIX condition variable

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

- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
  - Entrance queue can be a POSIX recursive mutex
  - Wait queue can be a POSIX condition variable
  - Similar to Java ConditionObjects

See earlier lesson on “Java ConditionObjects”
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

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

See github.com/douglascraigschmidt/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)](en.wikipedia.org/wiki/Monitor_(synchronization))
#Implicit_condition_variable_monitors
new Thread(() -> {
    while (true)
        System.out.println(take());
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Acquire lock

Critical Section

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

SimpleBlockingBoundedQueue

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

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

Visual Analysis of SimpleBlockingBoundedQueue
Visual Analysis of SimpleBlockingBoundedQueue

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

T1 = new Thread(() -> {
    while(true)
        System.out.println(take());
    }).start();
```

Critical Section

```java
while(mList.isEmpty())
    wait();
```
SimpleBlockingBoundedQueue

Critical Section

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

T1

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

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

SimpleBlockingBoundedQueue

Critical Section

Acquire lock

T₂

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

Block on
monitor condition

T₁
```java
new Thread(() -> {
    for(int i = 0; i < 10; i++)
        put(Integer.toString(i));
}).start();
while (mList.isFull())
    wait();
```
new Thread(() -> {
    for (int i = 0; i < 10; i++)
        put(Integer.toString(i));
    mList.add(msg);
    notifyAll();
}).start();
Visual Analysis of SimpleBlockingBoundedQueue

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

mList.add(msg);
notifyAll();
Visual Analysis of SimpleBlockingBoundedQueue

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

SimpleBlockingBoundedQueue

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

SimpleBlockingBoundedQueue

Unblock on monitor condition

T₁

critical Section

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

SimpleBlockingBoundedQueue

Acquire lock

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

SimpleBlockingBoundedQueue

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

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.
Visual Analysis of SimpleBlockingBoundedQueue

SimpleBlockingBoundedQueue

Critical Section

notifyAll();
return mList.poll();

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

SimpleBlockingBoundedQueue

Critical Section

Release lock

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

SimpleBlockingBoundedQueue

Critical Section

Leave monitor object

T₁

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
• 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
Code Analysis of SimpleBlockingBoundedQueue

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

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

This internal state must be protected against race conditions

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/SimpleBlockingBoundedQueue
Code Analysis of SimpleBlockingBoundedQueue

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

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;
    }
  }

Code Analysis of SimpleBlockingBoundedQueue
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.

```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

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

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

        final E e = mList.poll();

        notifyAll();
        return e;
    }
}
```

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
Code Analysis of SimpleBlockingBoundedQueue

- `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.

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

This limitations 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!

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

This is due to the non-determinism of concurrency.
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

```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/Spurious_wakeup
**Code Analysis of SimpleBlockingBoundedQueue**

- 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 T₂ calls \texttt{put()}, which acquires the intrinsic lock & adds an item to the queue so it’s no longer empty.
A thread blocked on wait() won’t continue until it’s notified that the condition expression may be true.

Assuming that thread $T_1$ is blocked in take() the queue won’t be full!
A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true. `notifyAll()` is used here due to Java’s monitor object limitations, i.e., there’s only a single wait queue.

```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;
    }
    ...
}
```
Several steps occur when a waiting thread is notified

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

Code Analysis of SimpleBlockingBoundedQueue

```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

```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()

```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
  • re-evaluates the condition expression
  • continues after wait()

Code Analysis of SimpleBlockingBoundedQueue

```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)