Java Semaphore (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

• Appreciate the concept of semaphores
Learning Objectives in this Part of the Module

- Appreciate the concept of semaphores
- Recognize the two types of semaphores
Learning Objectives in this Part of the Module

• Appreciate the concept of semaphores
• Recognize the two types of semaphores
• Know a human known use of semaphores
Overview of Semaphores
Overview of Semaphores

- A semaphore is conceptually an “object” that can be atomically incremented & decremented to control access to a shared resource.

See en.wikipedia.org/wiki/Semaphore_(programming)
Overview of Semaphores

- A semaphore is conceptually an “object” that can be atomically incremented & decremented to control access to a shared resource
  - e.g., originally used to control access to a shared railroad track

See en.wikipedia.org/wiki/Railway_semaphore_signal
Overview of Semaphores

• Concurrent programs use semaphores to coordinate interactions between multiple threads

See tutorials.jenkov.com/java-concurrency/semaphores.html
Overview of Semaphores

- Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,
  - A semaphore can control the access of threads to a limited # of resources

See [www.youtube.com/watch?v=RAv71VbdkBc](http://www.youtube.com/watch?v=RAv71VbdkBc) for the Semaphore anthem ;-)
Overview of Semaphores

- Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,
  - A semaphore can control the access of threads to a limited # of resources
  - It records a count ("permits") of how many units of a resource are available
Overview of Semaphores

- Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,
  - A semaphore can control the access of threads to a limited # of resources
  - It records a count ("permits") of how many units of a resource are available
  - It provides operations to adjust the permit count atomically as units are acquired or released
Overview of Semaphores

Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,

- A semaphore can control the access of threads to a limited # of resources
- It records a count ("permits") of how many units of a resource are available
- It provides operations to adjust the permit count atomically as units are acquired or released
Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,

- A semaphore can control the access of threads to a limited number of resources.
- It records a count ("permits") of how many units of a resource are available.
- It provides operations to adjust the permit count atomically as units are acquired or released.
- Threads can wait (timed or blocking) until a unit of the resource is available.
Concurrent programs use semaphores to coordinate interactions between multiple threads, e.g.,

- A semaphore can control the access of threads to a limited # of resources.
- It records a count ("permits") of how many units of a resource are available.
- It provides operations to adjust the permit count atomically as units are acquired or released.
- Threads can wait (timed or blocking) until a unit of the resource is available.
- When a thread is done with a resource the permit count is incremented atomically & another waiting thread can acquire it.
Overview of Semaphores

• There are two types of semaphores
Overview of Semaphores

• There are two types of semaphores
  • Counting semaphores

See javarevisited.blogspot.com/2012/05/counting-semaphore-example-in-java-5.html
Overview of Semaphores

- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter \( N \) with precise meaning
Overview of Semaphores

- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter (N) with precise meaning
      - **Negative**
        - exactly -N threads queued waiting to acquire semaphore
There are two types of semaphores

**Counting semaphores**
- Have # of permits defined by a counter (N) with precise meaning
  - **Negative**
  - **Zero** == no waiting threads
    - an acquire operation will block the invoking thread until the counter N is positive
Overview of Semaphores

• There are two types of semaphores
  • **Counting semaphores**
    • Have # of permits defined by a counter (N) with precise meaning
      • **Negative**
      • **Zero** == no waiting threads
    • **Positive** == no waiting threads
      • an acquire operation will not block the invoking thread
Overview of Semaphores

• There are two types of semaphores
  • Counting semaphores
  • Binary semaphores

See howtodoinjava.com/core-java/multi-threading/binary-semaphore-tutorial-and-example
Overview of Semaphores

- There are two types of semaphores
  - Counting semaphores
  - **Binary semaphores**
    - Has only 2 states: acquired (0) & not acquired (1)
Overview of Semaphores

• There are two types of semaphores
  • Counting semaphores
  • **Binary semaphores**
    • Has only 2 states: acquired (0) & not acquired (1)
    • Restricts the counter N to the values 0 & 1

In practice, binary semaphores are often implemented via counting semaphores.
Overview of Semaphores

- We’ll analyze examples of counting & binary semaphores later
Overview of Semaphores

- We’ll analyze examples of counting & binary semaphores later, e.g.
- The PalantiriSimulator app uses a counting semaphore

See github.com/douglasraigschmidt/CS891/tree/master/assignments
Overview of Semaphores

- We’ll analyze examples of counting & binary semaphores later, e.g.
  - The PalantiriSimulator app uses a counting semaphore
  - The Ping/Pong app uses a pair of binary semaphores

See github.com/douglascraigschmidt/LiveLessons/tree/master/PingPongApplication
Human Known Use of Semaphores
Human Known Uses of Semaphores

- A human known use of counting semaphores applies them to schedule access to beach volleyball courts

See en.wikipedia.org/wiki/Corona_del_Mar_State_Beach
Human Known Uses of Semaphores

- A human known use of counting semaphores applies them to schedule access to beach volleyball courts.
- A bag full of balls is used to limit the number of teams that can concurrently play volleyball.
End of Java Semaphores
(Part 1)
Java Semaphore (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 Lesson

- Appreciate the concept of semaphores
- Recognize the two types of semaphores
- Know a human known use of semaphores
- Understand the structure & functionality of Java Semaphore & its methods
Overview of Java Semaphores
Overview of Java Semaphores

- Implements a variant of counting semaphores

```java
public class Semaphore
    implements ...
```

### Class Semaphore

```java
java.lang.Object
    java.util.concurrent.Semaphore
```

**All Implemented Interfaces:**

- Serializable

```java
public class Semaphore
extends Object
implements Serializable
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphore are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
Overview of Java Semaphores

• Implements a variant of counting semaphores

```java
public class Semaphore implements ... {
...
```

Semaphore doesn’t implement any synchronization-related interfaces

### Class Semaphore

```java
class Semaphore
extends Object
implements Serializable
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:
Overview of Java Semaphores

• Constructors create semaphore with a given # of permits

```java
public class Semaphore implements ... {
    ...
    public Semaphore (int permits) {
        ...
    }
    ...
    public Semaphore (int permits, boolean fair) {
        ...
    }
    ...
}
```
Overview of Java Semaphores

- Constructors create semaphore with a given # of permits
- This # is *not* a maximum, it’s just an initial value

```java
public class Semaphore implements ...
{
    ...
    public Semaphore (int permits) {
        ...
    }
    ...
    public Semaphore (int permits, boolean fair) {
        ...
    }
    ...
}
```

See [stackoverflow.com/questions/7554839/how-and-why-can-a-semaphore-give-out-more-permits-than-it-was-initialized-with](http://stackoverflow.com/questions/7554839/how-and-why-can-a-semaphore-give-out-more-permits-than-it-was-initialized-with)
Overview of Java Semaphores

• Constructors create semaphore with a given # of permits
  • This # is not a maximum, it’s just an initial value
  • The initial permit value can be negative!!

```java
public class Semaphore implements ... {
    ...
    Semaphore s = new Semaphore(-1);
    ...
}
```

In this case, all threads will block trying to acquire the semaphore until some thread(s) increment the permit value until it’s positive
Overview of Java Semaphores

- Applies the *Bridge* pattern

```java
public class Semaphore
    implements ...
{
...
```

Decouples its interface from its implementation so fair & non-fair semantics can be supported uniformly

See [en.wikipedia.org/wiki/Bridge_pattern](en.wikipedia.org/wiki/Bridge_pattern)
Overview of Java Semaphores

- Applies the Bridge pattern
- Locking handled by Sync Implementor hierarchy

```java
public class Semaphore
    implements ...
{
    ...
    /** Performs sync mechanics */
    private final Sync sync;
}
```
Overview of Java Semaphores

- Applies the *Bridge* pattern
- Locking handled by *Sync* Implementor hierarchy
- Reuses functionality from *AbstractQueuedSynchronizer*
- Many Java synchronizers that rely on FIFO wait queues use this framework

```java
public class Semaphore
    implements ...
{
    ...

    /** Performs sync mechanics */
    private final Sync sync;

    /**
     * Synchronization implementation
     * for semaphore
     */
    abstract static class Sync extends
        AbstractQueuedSynchronizer {
        ...
    }
}
```

See gee.cs.oswego.edu/dl/papers/aqs.pdf
Overview of Java Semaphores

• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
• Reuses functionality from AbstractQueuedSynchronizer
• Optionally implement fair or non-fair lock acquisition model

```java
public class Semaphore
    implements ...
{
    ...

    public Semaphore
        (int permits,
         boolean fair) {
        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);
    }
    ...
```

The Semaphore fair & non-fair models follow the same pattern used by the “Java ReentrantLock”
• Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
• Reuses functionality from AbstractQueuedSynchronizer
• Optionally implement fair or non-fair lock acquisition model

```java
public class Semaphore
    implements ...
{

    public Semaphore
        (int permits,
         boolean fair) {

        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);

    }

    ...

    This param determines whether FairSync or NonfairSync is used
```
Overview of Java Semaphores

- Applies the Bridge pattern
- Locking handled by Sync Implementor hierarchy
- Reuses functionality from AbstractQueuedSynchronizer
- Optionally implement fair or non-fair lock acquisition model

```java
public class Semaphore implements ... {
    public Semaphore (int permits, boolean fair) {
        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);
    }
    ...
}
```

Ensures strict “FIFO” fairness, at the expense of performance
Overview of Java Semaphores

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Reuses functionality from AbstractQueuedSynchronizer
- Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore implements ... {

    public Semaphore (int permits, boolean fair) {
        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);
    }

    ...
```

*Enables faster performance at the expense of fairness*
Overview of Java Semaphores

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Reuses functionality from `AbstractQueuedSynchronizer`
- Optionally implement fair or non-fair lock acquisition model

```java
public class Semaphore implements ... {
    ...

    public Semaphore
            (int permits,
             boolean fair) {
        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);
    }

    public Semaphore
            (int permits) {
        sync = new
            NonfairSync(permits);
    }

    ...

    The default behavior favors performance over fairness
```
Overview of Java Semaphores

- Acquiring & releasing permits from/to a semaphore need not be “fully bracketed”
  - i.e., a thread that acquires a semaphore need not be the one that releases it

See example in part 3 of this lesson
Overview of Key Java Semaphore Methods
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore

```java
public class Semaphore
    implements ... {

    ...

    public void acquire() { ... }

    public void acquireUninterruptibly()
    { ... }

    public boolean tryAcquire
        (long timeout,
         TimeUnit unit)
    { ... }

    public void release() { ... }

    ...

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html
```
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore

```java
public class Semaphore
    implements ...
{
    ...
    public void acquire() { ... }

    public void
        acquireUninterruptibly()
    { ... }

    public boolean tryAcquire
        (long timeout,
             TimeUnit unit)
    { ... }

    public void release() { ... }
    ...
```

These methods forward to their implementor methods, most of which are inherited from the AbstractQueuedSynchronizer framework.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
- acquire() obtains a permit from the semaphore
- Can be interrupted

```java
public class Semaphore
    implements ...
{
    ...
    public void acquire()
    {
        sync.
        acquireSharedInterruptibly(1);
    }
    ...
}
```

See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - `acquire()` obtains a permit from the semaphore
  - `acquireUninterruptibly()` also obtains a permit from the semaphore
  - Cannot be interrupted

```java
public class Semaphore implements ... {
    ...
    public void acquireUninterruptibly() {
        sync.acquireShared(1)
    }
    ...
}
```
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if it’s available at invocation time

```java
public class Semaphore
    implements ...
{
    ...

    public boolean tryAcquire()
        ...
    {
        sync.
        nonfairTryAcquireShared(1)
        >= 0;
    }

    ...
}
```
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if it’s available at invocation time

```java
public class Semaphore implements ...
{
    ...
    public boolean tryAcquire()
    {
        sync.
        nonfairTryAcquireShared(1)
        >= 0;
    }
    ...
}
```

Untimed tryAcquire() methods will “barge”, i.e., they don’t honor the fairness setting & take any permits available
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - `acquire()` obtains a permit from the semaphore
  - `acquireUninterruptibly()` ignores interrupts
  - `tryAcquire()` obtains a permit if it’s available at invocation time
  - `release()` increments the permit count by 1

```java
public class Semaphore implements ... {
    ...
    public void release() {
        sync.releaseShared(1);
    }
    ...
}
```

Recall it’s valid for the permit count to exceed the initial permit count!!
Overview of Other Java Semaphore Methods
### Overview of Other Java Semaphore Methods

- There are many other Semaphore methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void acquire(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all are available, or thread interrupted</td>
</tr>
<tr>
<td><code>void acquireUninterruptibly(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all available</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits)</code></td>
<td>Acquires given # of permits from semaphore, only if all are available at the time of invocation</td>
</tr>
<tr>
<td><code>void release(int permits)</code></td>
<td>Releases # of permits, returning them to semaphore</td>
</tr>
<tr>
<td><code>boolean tryAcquire(long timeout, TimeUnit unit)</code></td>
<td>Acquires a permit from semaphore, if one is available within given waiting time &amp; thread has not been interrupted</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits, long timeout, TimeUnit unit)</code></td>
<td>Acquires given # of permits from semaphore, if all available within given waiting time &amp; current thread has not been interrupted</td>
</tr>
</tbody>
</table>
### Overview of Other Java Semaphore Methods

- There are many other Semaphore methods
- Some methods can acquire or release multiple permits at a time

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void acquire(int permits)</td>
<td>Acquires # of permits from semaphore, blocking until all are available, or thread interrupted</td>
</tr>
<tr>
<td>void acquireUninterruptibly(int permits)</td>
<td>Acquires # of permits from semaphore, blocking until all available</td>
</tr>
<tr>
<td>boolean tryAcquire(int permits)</td>
<td>Acquires given # of permits from semaphore, only if all are available at the time of invocation</td>
</tr>
<tr>
<td>void release(int permits)</td>
<td>Releases # of permits, returning them to semaphore</td>
</tr>
<tr>
<td>boolean tryAcquire(long timeout, TimeUnit unit)</td>
<td>Acquires a permit from semaphore, if one is available within given waiting time &amp; thread has not been interrupted</td>
</tr>
<tr>
<td>boolean tryAcquire(int permits, long timeout, TimeUnit unit)</td>
<td>Acquires given # of permits from semaphore, if all available within given waiting time &amp; current thread has not been interrupted</td>
</tr>
</tbody>
</table>
Overview of Other Java Semaphore Methods

- There are many other Semaphore methods
  - Some methods can acquire or release multiple permits at a time
  - Likewise, some of these methods use timeouts

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void acquire(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all are available, or thread interrupted</td>
</tr>
<tr>
<td><code>void acquireUninterruptibly(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all available</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits)</code></td>
<td>Acquires given # of permits from semaphore, only if all are available at the time of invocation</td>
</tr>
<tr>
<td><code>void release(int permits)</code></td>
<td>Releases # of permits, returning them to semaphore</td>
</tr>
<tr>
<td><code>boolean tryAcquire(long timeout, TimeUnit unit)</code></td>
<td>Acquires a permit from semaphore, if one is available within given waiting time &amp; thread has not been interrupted</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits, long timeout, TimeUnit unit)</code></td>
<td>Acquires given # of permits from semaphore, if all available within given waiting time &amp; current thread has not been interrupted</td>
</tr>
</tbody>
</table>

Ironically, the timed `tryAcquire()` methods do honor the fairness setting, so they don’t “barge”
End of Java Semaphores (Part 2)