The Specific Notification Pattern: “Fair” Semaphore Semantics

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

- Understand the *Specific Notification* pattern
- Be aware of the semantics of “fair” semaphores

**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.

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:
An Overview of Fair Semaphore Semantics
Overview of Fair Semaphore Semantics

- Threads calling `acquire()` on a “fair” semaphore obtain permits in “first-in, first-out” (FIFO) order.

```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.

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:
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
Overview of Fair Semaphore Semantics

- Threads calling acquire() on a “fair” semaphore obtain permits in “first-in, first-out” (FIFO) order.
- FIFO ordering applies to internal points of execution within semaphore methods.

```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.

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 Fair Semaphore Semantics

- Threads calling acquire() on a “fair” semaphore obtain permits in “first-in, first-out” (FIFO) order.
- FIFO ordering applies to internal points of execution within semaphore methods.
  - e.g., one thread can invoke acquire() before another, but reach the ordering point after the other.

```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.

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 Fair Semaphore Semantics

- Threads calling acquire() on a “fair” semaphore obtain permits in “first-in, first-out” (FIFO) order
- FIFO ordering applies to internal points of execution within semaphore methods
- The Specific Notification pattern provides an effective model for implementing fair semaphore semantics

Specific Notification for Java Thread Synchronization

Tom Cargill
Consultant
Box 69, Louisville, CO 80027
www.sni.net/~cargill

Abstract

Java supports thread synchronization by means of monitor-like primitives. The weak semantics of Java’s signaling mechanism provides little control over the order in which threads acquire resources, which encourages the use of the Haphazard Notification pattern, in which an arbitrary thread is selected from a set of threads competing for a resource. For synchronization problems in which such arbitrary selection of threads is unacceptable, the Specific Notification pattern may be used to designate exactly which thread should proceed. Specific Notification provides an explicit mechanism for thread selection and scheduling.

0. Introduction

To study Java’s threads, I first tackled some of the classic exercises, like the “Dining Philosophers” and the “Readers and Writers.” The solutions that I obtained were reasonable, but I felt uncomfortable with the degree to which I had to depend on serendipitous treatment with respect to contention for locks and notifications. The solutions were free of deadlock, but were not fair in all circumstances. I thought I might threads could have active requests outstanding with an NNTP server. The fundamental correctness of this class depended on waiting threads being reactivated in exactly the right order to receive their responses from the server. In coding this class I applied the Specific Notification mechanism described below. With new insight, I returned to the earlier exercises and found that Specific Notification provided complete solutions to those problems. I therefore propose the Specific Notification pattern.

See www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf (especially Listing 3)
End of the Specific Notification Pattern: Fair Semaphore Semantics
The Specific Notification Pattern: Implementing a “Fair” Semaphore

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

• Understand the *Specific Notification* pattern

• Be aware of the semantics of “fair” semaphores

• Recognize how to implement a “fair” semaphore using the *Specific Notification* pattern

See [www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf) (especially Listing 3)
Visual Analysis of Fair Semaphore Acquire $(T_1 \& T_2)$
Visual Analysis of Fair Semaphore Acquire (T₁ & T₂)

FairSemaphore s = new FairSemaphore(1);

// Thread T1
s.acquire();

Create a fair semaphore with a single permit
Visual Analysis of Fair Semaphore Acquire (T₁ & T₂)

```java
FairSemaphore s = new FairSemaphore(1);
...
// Thread T1
s.acquire();
```

*Thread T₁ acquires the semaphore immediately since there are no waiters*
Critical Section

Acquire Lock

Synchronize access via intrinsic (monitor) lock to protect the object’s state from race conditions
Critical Section

If queue is empty & permit count is greater than 0 decrement count & return

This is the "fast path"
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

- Critical Section
- $m\text{WaitQ}$
- $m\text{Permits} = 0$
- $T_1$

Release intrinsic (monitor) lock & leave the monitor object's critical section
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

```java
FairSemaphore s = new FairSemaphore(1);
...
// Thread $T_2$
s.acquire();
```

Thread $T_2$ blocks in `acquire()` since no permits are available

Critical Section

Enter monitor object

FairSemaphore

mPermits

0
Create a new waitObj\_A that can be synchronized, queued in the wait queue, & waited upon by thread T\_2

This waiting happens outside of the FairSemaphore’s critical section
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

- **Acquire**

- **Lock**

- **waitObj**

- **mPermits**

- Must synchronize on **waitObj**
  to ensure that it’s not deleted prematurely in **release()**
Critical Section

FairSemaphore

waitObj_A

Critical Section

Acquire Lock

T_2

Must also synchronize on the intrinsic (monitor) lock to update wait queue safely

mPermits

0
Critical Section

Add $\text{waitObj}_A$ to the end of the wait queue, i.e., in FIFO order.
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

Release intrinsic (monitor) lock & leave the monitor object’s critical section

T2

FairSemaphore

Critical Section

mWaitQ

mPermits

0

waitObjA

Critical Section

22
wait() must be made with waitObj$_A$ synchronized, but w/out holding the intrinsic (monitor) lock to avoid “self-deadlock”
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

wait() atomically releases the waitObj$_A$ monitor lock & blocks
Here’s what happens when a Java `InterruptedException` (IE) is thrown in the `acquire()` method during a blocking call to `wait()` on a `waitObj`
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

Must reacquire intrinsic (monitor) lock in the catch clause to access the wait queue safely.
Critical Section

mWaitQ

mPermits 0

FairSemaphore

Try removing waitObj from wait queue (if it’s not on the queue another thread has released it, so give back permit via release())
Visual Analysis of Fair Semaphore Acquire ($T_1$ & $T_2$)

- Critical Section
- $mWaitQ$
- $mPermits = 0$
- $T_2$
- Rethrow the `InterruptedException`
Visual Analysis of Fair Semaphore Acquire ($T_3$)
Visual Analysis of Fair Semaphore Acquire (T₃)

FairSemaphore

Enter monitor object

Critical Section

Thread T₃ blocks in acquire() since there’s already a waiter

mWaitQ

mPermits

0

waitObj_B

FairSemaphore s = new FairSemaphore(1);

// Thread T3
s.acquire();
Visual Analysis of Fair Semaphore Acquire ($T_3$)

Create another $\text{waitObj}_B$ that can be synchronized, queued, & waited upon by thread $T_3$
Visual Analysis of Fair Semaphore Acquire ($T_3$)

Critical Section

Acquire Lock

waitObj$_B$

$T_3$

Synchronize on waitObj$_B$ to ensure it’s not deleted prematurely in release()
Critical Section

FairSemaphore

Critical Section

waitObjB

mWaitQ

mPermits

0

Release lock

Block on monitor condition

T_3

must synchronize on the intrinsic (monitor) lock to update wait queue safely
Visual Analysis of Fair Semaphore Acquire ($T_3$)

- Add new waitObj$_B$ to end of wait queue, i.e., in FIFO order
Critical Section

Release intrinsic (monitor) lock & leave the monitor object’s critical section

mWaitQ

waitObj_A

waitObj_B

mPermits

0

T_3

Visual Analysis of Fair Semaphore Acquire (T_3)
Critical Section

waitObj_A

waitObj_B

mWaitQ

mPermits

0

wait() on waitObj_B must be made w/ waitObj_B synchronized, but w/ out holding monitor lock to avoid “self-deadlock”
Visual Analysis of Fair Semaphore Acquire (T₃)

wait() atomically releases the waitObjₐ monitor lock & blocks until it is notified

wait() atomically releases the waitObjₐ monitor lock & blocks until it is notified
Visual Analysis of Fair Semaphore Release (T₄)
Thread \( T_4 \) releases longest waiting thread, i.e., thread \( T_2 \) waiting on \( \text{waitObj}_A \). 

\[
\text{FairSemaphore } s = \text{new FairSemaphore}(1); \\
\text{...} \\
// \text{Thread } T_4 \\
\text{s.release();}
\]

Thread \( T_4 \) could be thread \( T_1 \) or a different thread altogether.
Must synchronize on intrinsic (monitor) lock to update permit count & wait queue safely
Visual Analysis of Fair Semaphore Release ($T_4$)

If a "next" waiter (waitObj$_A$) is in wait queue another thread is waiting to acquire semaphore, so don’t increment permit count.
Visual Analysis of Fair Semaphore Release (T₄)

Must acquire the monitor lock on the next \textit{waitObj}_A
Visual Analysis of Fair Semaphore Release ($T_4$)

Inform thread blocked on next $waitObj_A$ in acquire() that a permit’s available
Visual Analysis of Fair Semaphore Release ($T_4$)

Unlock the next monitor object so thread $T_1$ waiting in acquire() can continue.
Visual Analysis of Fair Semaphore Release ($T_4$)

Release intrinsic (monitor) lock & leave the monitor object’s critical section
If there are no waiting threads increment the permit count by 1 so another thread can acquire the semaphore.
End of Implementing a Fair Semaphore with the Specific Notification Pattern