8. Condition Objects

Oscar Nierstrasz
Roadmap

> Condition Objects
  — Simple Condition Objects
  — The “Nested Monitor Problem”
  — Permits and Semaphores
Roadmap

> Condition Objects
  — Simple Condition Objects
  — The “Nested Monitor Problem”
  — Permits and Semaphores
Pattern: Condition Objects

**Intent:** Condition objects *encapsulate the waits and notifications* used in guarded methods.

**Applicability**

> To simplify class design by off-loading waiting and notification mechanics.

— Because of the limitations surrounding the use of condition objects in Java, in some cases the use of condition objects will increase rather than decrease design complexity!

> …
Pattern: Condition Objects

Applicability

…

> As an efficiency manoeuvre.
  —By isolating conditions, you can often avoid notifying waiting threads that could not possibly proceed given a particular state change.

> As a means of encapsulating special scheduling policies surrounding notifications, for example to impose fairness or prioritization policies.

> In the particular cases where conditions take the form of “permits” or “latches”. 
Condition Objects

A client that awaits a condition blocks until another object signals that the condition now may hold.

```java
public interface Condition {
    public void await (); // wait for some condition
    public void signal (); // signal that condition
}
```

Cf. java.util.concurrent.locks.Condition
A Simple Condition Object

We can encapsulate guard conditions with this class:

```java
public class SimpleConditionObject implements Condition {
    public synchronized void await () {
        try { wait(); }
        catch (InterruptedException ex) {} 
    }
    public synchronized void signal () {
        notifyAll ();
    }
}
```

NB: Careless use can lead to the “Nested Monitor Problem”
Roadmap

> Condition Objects
  — Simple Condition Objects
  — The “Nested Monitor Problem”
  — Permits and Semaphores
The Nested Monitor problem

We want to avoid waking up the wrong threads by separately notifying the conditions notMin and notMax:

```java
public class BoundedCounterNestedMonitorBAD extends BoundedCounterAbstract {
    protected Condition notMin = new SimpleConditionObject();
    protected Condition notMax = new SimpleConditionObject();
    public synchronized long value() { return count; }
    ...
```
public synchronized void dec() {
    while (count == MIN) {
        notMin.await();  // wait till count not MIN
        if (count-- == MAX)
            notMax.signal();
    }
}

public synchronized void inc() {  // can’t get in!
    while (count == MAX) {
        notMax.await();
        if (count++ == MIN)
            notMin.signal();  // we never get here!
    }
}
The Nested Monitor problem ...

Nested monitor lockouts occur whenever a blocked thread holds the lock for an object containing the method that would otherwise provide a notification to unblock the wait.

Where is the waits-for cycle?
2nd example — Nested Monitors in FSP

Nested Monitors typically arise when one synchronized object is implemented using another.

Recall our one Slot buffer in FSP:

\[
\text{const } N = 2 \\
\text{Slot} = (\text{put}[v:0..N] \rightarrow \text{get}[v] \rightarrow \text{Slot}).
\]

Suppose we try to implement a call/reply protocol using a private instance of Slot:

\[
\text{ReplySlot} = (\text{put}[v:0..N] \rightarrow \text{my}.\text{put}[v] \rightarrow \text{ack} \rightarrow \text{ReplySlot} \\
| \quad \text{get} \rightarrow \text{my}.\text{get}[v:0..N] \rightarrow \text{ret}[v] \rightarrow \text{ReplySlot}).
\]
Nested Monitors in FSP ...

**Our producer/consumer chain obeys the new protocol:**

Producer = ( put[0] -> ack
          -> put[1] -> ack

Consumer = ( get-> ret[x:0..N] -> Consumer ).

||Chain = (Producer | ReplySlot | my:Slot | Consumer).
Nested Monitors in FSP ...

But now the chain may deadlock:

Progress violation for actions:

\{\{\text{ack, get}\}, \text{my.}\{\text{get, put}\}[0..2], \{\text{put, ret}\}[0..2]\}\n
Trace to terminal set of states:

get

Actions in terminal set:

\{

\}
Solving the Nested Monitors problem

**You must ensure that:**

> Waits do not occur while synchronization is held on the host object.
  — Leads to a guard loop that reverses the faulty synchronization

> **Notifications are never missed.**
  — The entire guard wait loop should be enclosed within synchronized blocks on the condition object.

> **Notifications do not deadlock.**
  — All notifications should be performed only upon release of all synchronization (except for the notified condition object).

> **Helper and host state must be consistent.**
  — If the helper object maintains any state, it must always be consistent with that of the host
  — If it shares any state with the host, access must be synchronized.
Why must we sync on notMin?
Other solutions …

> Be sure to lock just a *single object*
  — i.e., either the host or the condition object

> Remove host synchronization (if safe or immutable)
The ReplySlot wrapper has no state of its own, so we can simply remove the synchronization!
Roadmap

> Condition Objects
  — Simple Condition Objects
  — The “Nested Monitor Problem”
  — Permits and Semaphores
**Pattern: Permits and Semaphores**

**Intent:** Bundle synchronization in a condition object when *synchronization depends on the value of a counter.*

**Applicability**
> When any given `wait` may proceed only if there have been *more signals than awaits.*
  > —I.e., when `await` decrements and `signal` increments the number of available “permits”.

> You need to guarantee the *absence of missed signals.*
  > —Unlike simple condition objects, semaphores work even if one thread enters its `await` after another thread has signalled that it may proceed (!)

> The host classes can arrange to invoke `Condition` methods outside of synchronized code.
Permits and Semaphores — design steps

> Define a class implementing `Condition` that maintains a permit count, and immediately releases await if there are already enough permits.
  — e.g., `BoundedCounter`
public class Permit implements Condition {
    private int count;
    Permit(int init) { count = init; }
    public synchronized void await() {
        while (count == 0) {
            try { wait(); }
            catch(InterruptedException ex) { };
        }
        count --;
    }
    public synchronized void signal() {
        count ++;
        notifyAll();
    }
}
As with all kinds of condition objects, their clients must avoid invoking `await` inside of synchronized code.

—You can use a `before/after design` of the form:

```java
class Host {
    Condition aCondition; ...
    public method m1() {
        aCondition.await();       // not synced
        doM1();                   // synced
        for each Condition c enabled by m1()
            c.signal();         // not synced
    }
    protected synchronized doM1() { ... }
}
```
Using permits

```java
public class Building{
    Permit permit;
    Building(int n) {
        permit = new Permit(n);
    }
    void enter(String person) { // NB: unsynchronized
        permit.await();
        System.out.println(person + " has entered the building");
    }
    void leave(String person) {
        System.out.println(person + " has left the building");
        permit.signal();
    }
}
```
Using permits

```java
public static void main(String[] args) {
    Building building = new Building(3);
    enterAndLeave(building, "bob");
    enterAndLeave(building, "carol");
    ...
}

private static void enterAndLeave(final Building building,
                                    final String person) {
    new Thread() {
        public void run() {
            building.enter(person);
            pause();
            building.leave(person);
        }
    }.start();
}
```

bob has entered the building
carol has entered the building
ted has entered the building
**bob has left the building**
alice has entered the building
ted has left the building
carol has left the building
elvis has entered the building
alice has left the building
elvis has left the building
**Variants**

**Permit Counters:** (Counting Semaphores)
- Just keep track of the number of “permits”
- Can use `notify` instead of `notifyAll` if class is final

**Fair Semaphores:**
- Maintain *FIFO queue of threads* waiting on a SimpleCondition

**Locks and Latches:**
- Locks can be *acquired and released* in separate methods
- Keep track of thread holding the lock so locks can be *reentrant*!
- A `latch` is set to true by `signal`, and *always stays true* (eg a future)
public class Semaphore {  // simple version
    private int value;
    public Semaphore (int initial) { value = initial; }
    synchronized public void up() {  // AKA V
        ++value;
        notify();  // wake up just one thread!
    }
    synchronized public void down() {  // AKA P
        while (value== 0) {
            try { wait(); } catch(InterruptedException ex) { };
        }
        --value;
    }
}
Using Semaphores

```java
public class BoundedCounterSem extends BoundedCounterAbstract { ... 
    protected Semaphore mutex, full, empty;
    BoundedCounterVSem() {
        mutex = new Semaphore(1);
        full = new Semaphore(0); // number of counters
        empty = new Semaphore(MAX-MIN); // number of empty slots
    }
    public long value() {
        mutex.down(); // grab the resource
        long val = count;
        mutex.up(); // release it
        return val;
    }
    public void inc() {
        empty.down(); // grab a slot
        mutex.down();
        count ++;
        mutex.up();
        full.up(); // release a counter
    }
    ... 
```
Using Semaphores ...

These would cause a nested monitor problem!

```java
... public void BADinc() {
    mutex.down(); empty.down(); // locks out BADdec!
    count ++;
    full.up(); mutex.up();
}
public void BADdec() {
    mutex.down(); full.down(); // locks out BADinc!
    count --;
    empty.up(); mutex.up();
}
}```
import java.util.concurrent.Semaphore;
public class BoundedCounterJUCSem extends BoundedCounterAbstract {
    protected Semaphore mutex;
    protected Semaphore full;
    protected Semaphore empty;

    BoundedCounterJUCSem() {
        mutex = new Semaphore(1); // one permit for critical section
        full = new Semaphore(0); // number of counters
        empty = new Semaphore((int)(MAX-MIN)); // number of empty slots
    }

    public void inc() {
        try {
            empty.acquire(); // grab a slot
            mutex.acquire();
        } catch (InterruptedException e) { }
        count ++;
        mutex.release();
        full.release(); // release a counter
        checkInvariant();
    }
}

What you should know!

> What are “condition objects”? How can they make your life easier? Harder?
> What is the “nested monitor problem”?
> How can you avoid nested monitor problems?
> What are “permits” and “latches”? When is it natural to use them?
> How does a semaphore differ from a simple condition object?
> Why (when) can semaphores use notify() instead of notifyAll()?
Can you answer these questions?

> Why doesn’t SimpleConditionObject need any instance variables?
> What is the easiest way to avoid the nested monitor problem?
> What assumptions do nested monitors violate?
> How can the obvious implementation of semaphores (in Java) violate fairness?
> How would you implement fair semaphores?
Attribution-ShareAlike 3.0

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

http://creativecommons.org/licenses/by-sa/3.0/