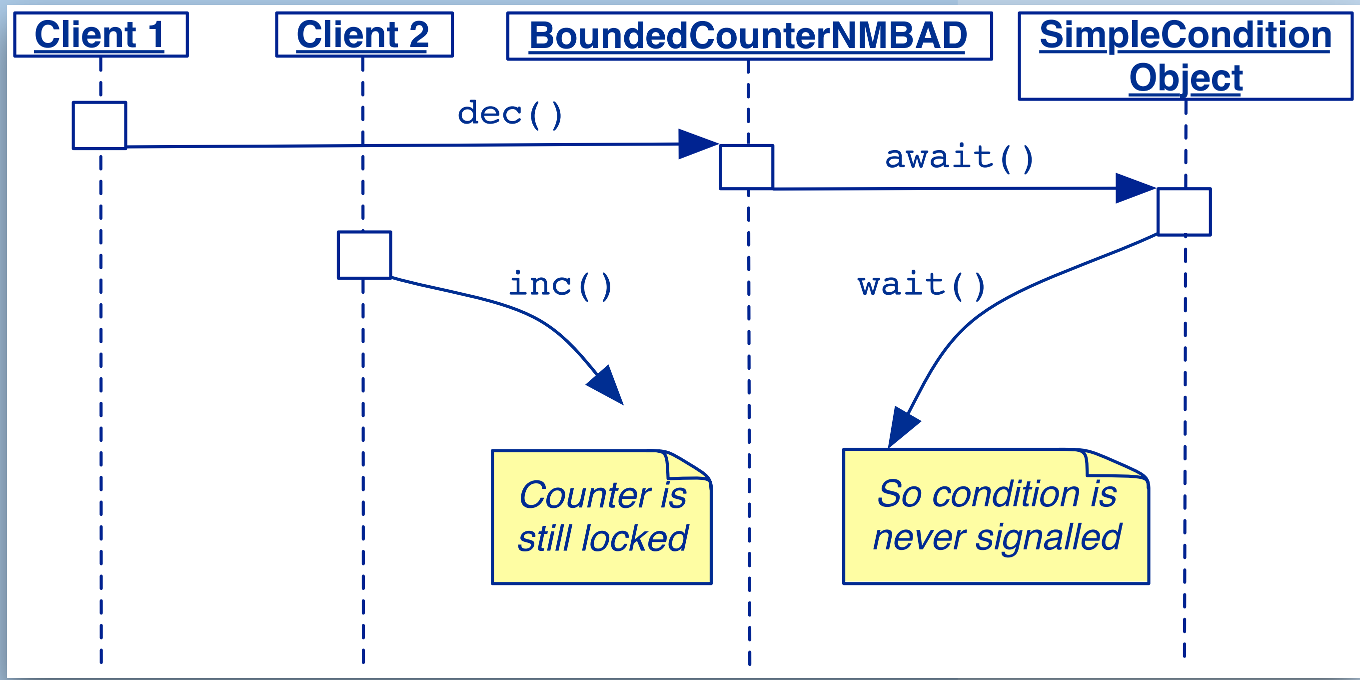
8. Condition Objects

Oscar Nierstrasz



Selected material © 2005 Bowbeer, Goetz, Holmes, Lea and Peierls

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!
- …

A “lock” object is an example of a condition object. Since Java supports monitors, making use of condition objects is in a sense a step backwards. On the other hand, in certain situations dedicated condition objects can nicely encapsulate specialized synchronization policies.

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”.

One drawback of Java monitors is that there is only a single notify method to wake up waiting threads, even if they are waiting for very different conditions. By encapsulating condition objects, we can offer the possibility for different threads to wait and by notified on distinct conditions.

Dedicated condition objects can also be used to selectively notify threads according to some fairness criterion (such as first-in, first-out).

Finally, some conditions have additional properties not captured by Java's object locks. A “permit” can be used by whatever object currently holds it. A “latch” has the property that once it is opened (or closed), it stays that way.

Condition Objects

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

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

Counter

Cf. java.util.concurrent.locks.Condition

A Simple Condition Object

We can encapsulate guard conditions with this class:

```
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:

```
public class BoundedCounterNestedMonitorBAD
 extends BoundedCounterAbstract {
   protected Condition notMin = new SimpleConditionObject();
   protected Condition notMax = new SimpleConditionObject();
   public synchronized long value() { return count; }
...
```

Counter

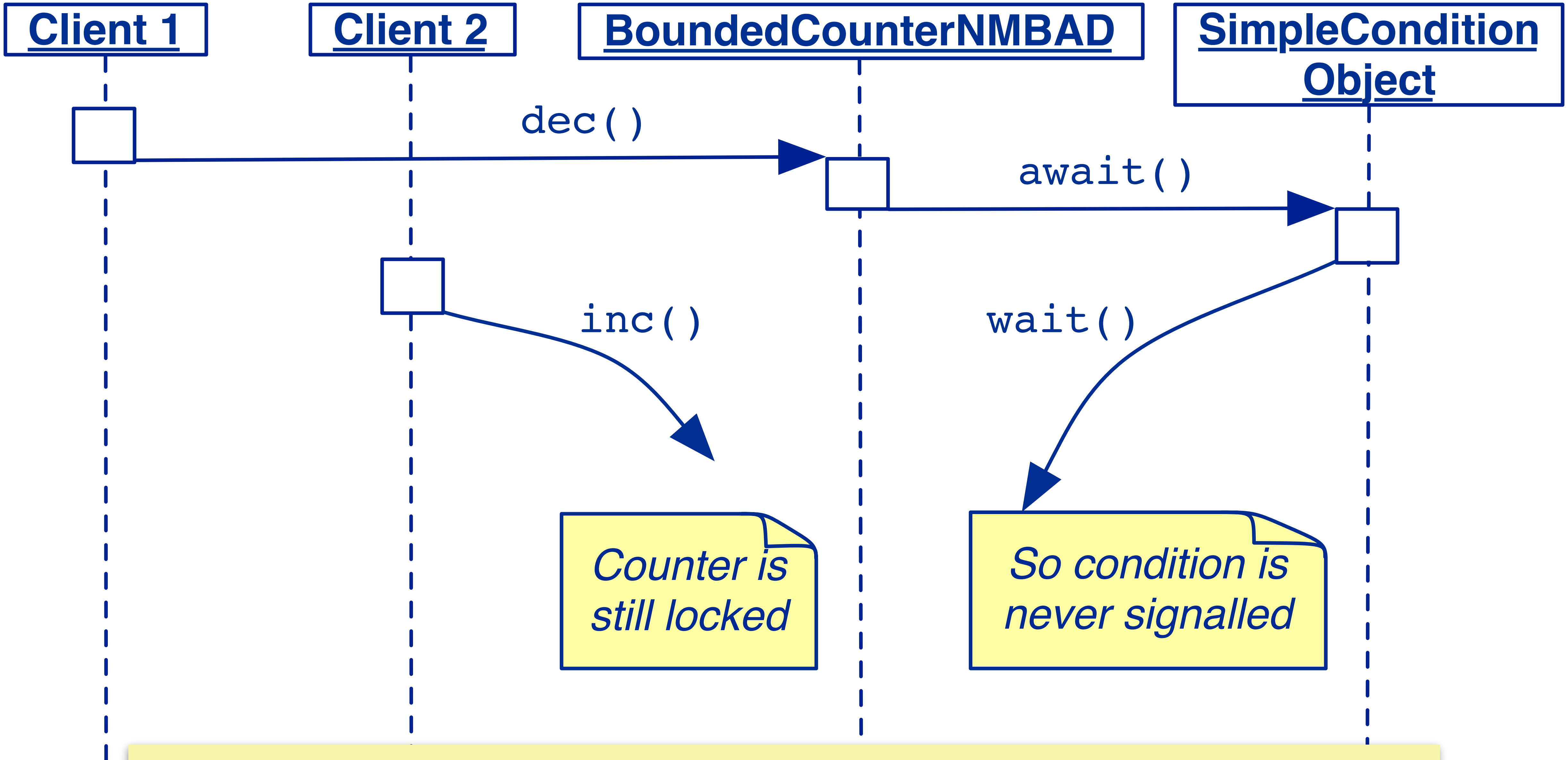
In this example we use condition objects to encapsulate the conditions that the `BoundedCounter` object is not currently at its minimum, respectively maximum value. The advantage of this design is that we can *separately signal threads waiting* for one condition or the other. With the classical monitor-based design, all threads are woken up by a `notifyAll` invocation, regardless of what condition they are actually waiting for.

```
   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!
   }
}
```

Unfortunately our design suffers from the *nested monitor problem*: the decrement and increment methods are synchronized as before, and they in turn make use of our synchronized condition objects. In other words, we have a monitor (`Condition`) nested inside another (`BoundedCounterNestedMonitorBAD`). Such nested monitors can quickly lead to a *deadlock*.

The problem is that if the condition `notMin` fails within the `dec` method, then the synchronization lock on the bounded counter *continues to be held*, since we `wait` on the `Condition`, not on the bounded counter. At this point no other thread can enter the monitor, and in particular no thread that would increment the counter and make the awaited condition true!

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.

Note that a nested monitor lockout is not a classical deadlock, as we do not have a waits-for cycle of processes holding resources wanted by other processes. Instead one process (C1) is holding the BC resource and is waiting for condition to come true. The other process (C2) wants in but it cannot get in (starvation). Nevertheless this is commonly thought of as a form of deadlock as each process is waiting for the other to make progress.

2^{nd} example — Nested Monitors in FSP

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

Recall our one Slot buffer in FSP:

```
const N = 2
Slot = (put[v:0..N] -> get[v] -> Slot).
```

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

```
ReplySlot  =  ( put[v:0..N] -> my.put[v] -> ack -> ReplySlot
              | get -> my.get[v:0..N] -> ret[v] -> ReplySlot ).
```

The idea here is to reuse the simple Slot process to implement a ReplySlot process supporting a more complex protocol. The problem is that the Slot process is synchronized, and requires put and get actions to strictly alternate. This will lead to a nested monitor problem.

Nested Monitors in FSP ...

Our producer/consumer chain obeys the new protocol:

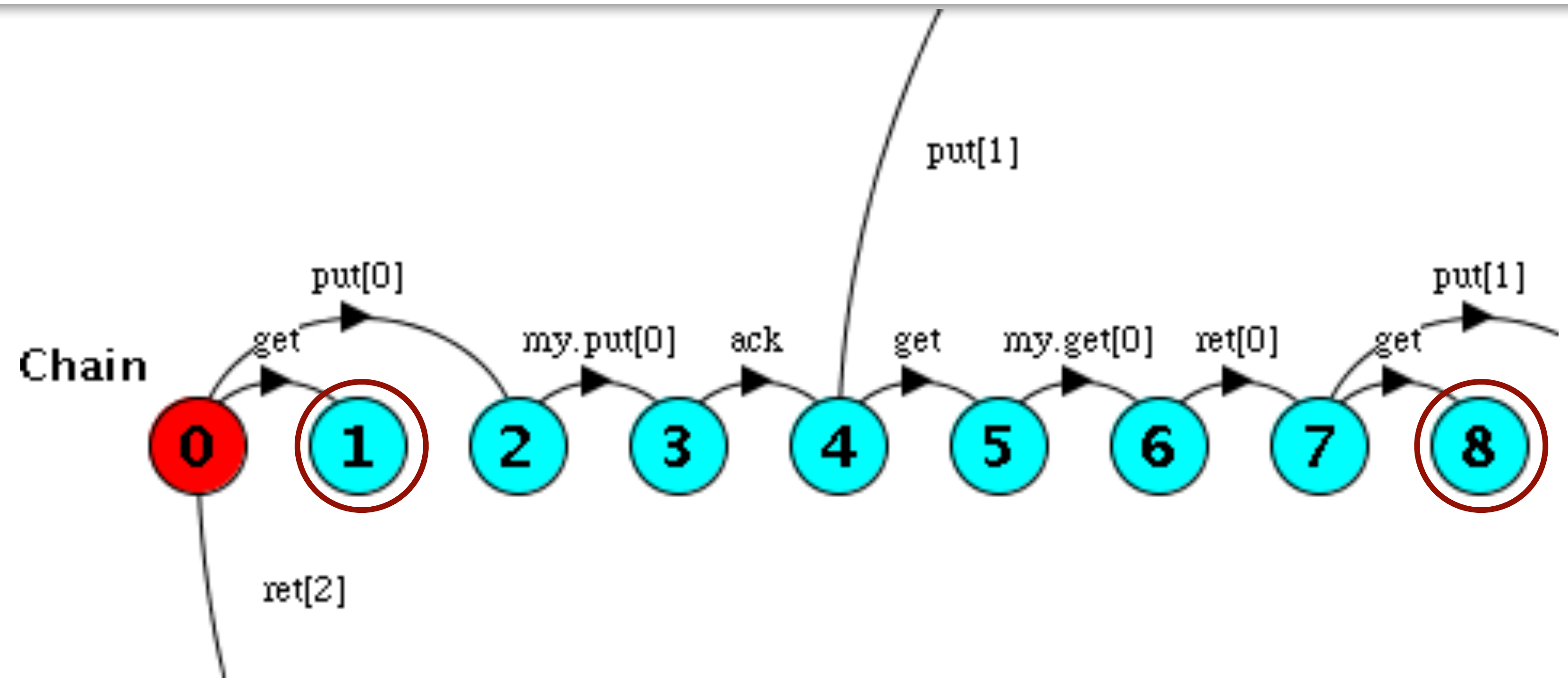
```
Producer =  ( put[0] -> ack
             ->put[1] -> ack
             ->put[2] -> ack -> Producer ).

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

||Chain = (Producer||ReplySlot||my:Slot||Consumer).
```

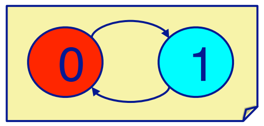
The Producer and Consumer follow the new protocol of ReplySlot (put is followed by ack and get by ret). We compose all three with a hidden Slot for the internal implementation of ReplySlot. It all looks very nice, but ...

Nested Monitors in FSP ...

But now the chain may deadlock:

```
Progress violation for actions:
   {{ack, get}, my.{get, put}[0..2], {put, ret}[0..2]}
Trace to terminal set of states:
   get
Actions in terminal set:
   {}
```





1-slotNM.lts

If we compose the processes and we see that there are several deadlocked states!

Notice that ReplySlot allows an initial get action. This then leads it to a state in which it wants to perform a get in its internal Slot, but this is not possible. We thereby reach a terminal set, i.e., a deadlocked state.

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.

The first point is the most important one: ensure that you do not hold the synchronization lock on the outer monitor while waiting for notification on the inner one. To fix this, you must *reverse the guard loop* so that you only wait on the inner monitor outside any synchronized method or block of the outer one.

Example solution

```
public class BoundedCounterCondition extends BoundedCounterAbstract {
public void dec() {                        // not synched!
      boolean wasMax = false;              // record notification condition
      synchronized(notMin) {               // synch on condition object
         while (true) {                    // new guard loop
            synchronized(this) {
               if (count > MIN) {          // check and act
                  wasMax = (count == MAX);
                  count--;
                  break;
               }
            }
            notMin.await();                // release host synch before wait
         }
      }
      if (wasMax) notMax.signal();         // release all syncs!
   } ...
}
```

Why must we sync on notMin?

This example is a bit convoluted, but it shows clearly how we avoid waiting on the condition objects within code that is synchronized on the outer monitor.

Note that we *need to synchronize on* `notMin`, otherwise we could *lose signals*, since we only do `notMin.await()` after checking the condition. (This is a *race condition*.)

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)

It is better to avoid such convoluted designs by instead using only conditions objects for all synchronization.

2nd example — removing synchronization

This version of ReplySlot has no state of its own,
so we can simply remove the synchronization!

```
||ReplySlot = (Putter||Getter).
Putter = ( put[v:0..N] -> my.put[v] -> ack -> Putter ).
Getter = ( get -> my.get[v:0..N] -> ret[v] -> Getter ).
```

```
Progress Check...
-- States: 54 Transitions: 90 Memory used: 6612K
No progress violations detected.
Progress Check in: 1ms
```

Our mistake was to make a choice between putting and getting. In this design we build the ReplySlot from independent Putter and Getter processes that share the hidden Slot (i.e., we remove the unneeded synchronization).

This eliminates the nested monitor problem and eliminates the deadlocked states.

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 `await` 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.

Unlike a simple lock, a permit or a semaphore counts the number of threads or processes that may enter a critical section.

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`

Note that our `BoundedCounter` class is already a form of permit! It offers a fixed number of “slots” to clients, and then blocks when all are occupied.

Example

```
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();
   }
}
```

Counter

The Permit class manages a fixed number (init) of “permits”. Each `await` attempts to grab a permit, and each `signal` releases one. When all permits are taken, `await` causes the invoking process to wait. This is useful in situations where some maximum number of processes are allowed to simultaneously access a shared resource.

Design steps ...

> As with all kinds of condition objects, clients must *avoid invoking await inside of synchronized code*.
>
> —You can use a *before/after design* of the form:

```
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() { ... }
}
```

Here you must ensure that there is no race condition between `aCondition.await()` and `doM1()`, i.e., it is not possible for the condition to change in between.

Using permits

```
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();
   }
}
```

Counter

Here we *only use permits* to synchronize access to the `Building`, so there can be no nested monitor problem.

Note the resemblance of the design of this class to `BoundedCounterBasic`.

Using permits

```
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
```

This Building can only hold three people. Anyone attempting to enter when the building is full must wait.

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* (e.g. a future)

Semaphores in Java

```
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;
   }
}
```

Counter

See also. java.util.concurrent.Semaphore

Why is it safe here to invoke `notify` *instead of* `notifyAll`*?*

Using Semaphores

```
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
  }
…
```

In this version of the `BoundedCounter` class, we *only use semaphores* for synchronization. We do not have any synchronized methods in the counter itself, but use a mutex semaphore to model the lock on the counter.

Note that in the increment method we *first* check the condition and *then* grab the mutex lock, rather than the other way around!

Using Semaphores ...

This would cause a nested monitor problem!

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

Reversing the order of usage of the semaphores would cause a nested monitor problem since we would hold the mutex lock on the counter while waiting for the condition to change.

The JUC version

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

The JUC version is very similar, except P and V are called `acquire` and `release`, and `acquire` might throw an `InterruptedException`.

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?*

creative commons

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/