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 have to resign myself to tolerating some unfairness in Java. Next, I built a multi-threaded NNTP[1] client, in which several 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.

Section 1 summarizes the semantics of Java's thread synchronization mechanisms, contrasting them with classical monitors; this section may be omitted by readers who have a detailed

[1] B. Kantor, P. Lapsley, Network News Transfer Protocol, Internic RFC 977, 1986.

knowledge of Java. Section 2 summarizes Java's conventional Haphazard Notification pattern, and discusses its limitations. Section 3 presents the Specific Notification pattern, and explains how it addresses those limitations. Section 4 offers additional examples of Specific Notification. Section 5 speculates that Specific Notification may be a member of a pattern language.

1. Java Thread Synchronization

The semantics of Java's thread synchronization primitives[2] approximate those of classical monitors.[3] A Java "lock" object corresponds to a monitor; Java's `synchronized` method[4] corresponds to a monitor procedure; Java's `wait` and `notify` methods on a lock object correspond to the `wait` and `signal` operations of a monitor.

There are two significant differences between Java's semantics and classical monitors. First, Java locks have no associated condition variables. The `wait` and `notify` methods operate on the lock object itself, not on distinct condition variables. (Alternatively, a Java lock may be viewed as a monitor with a single, implicit condition variable.) Therefore, a waiting thread cannot be notified that a specific condition has been satisfied by another thread. Second, context switching between a notifying thread and a notified thread is not fair.[5] Among a set of threads awaiting notification, the selection of the thread that receives the notification is arbitrary. Therefore, an unlucky thread can starve forever while waiting to acquire a resource, if there is sufficient ongoing competition from other threads. Moreover, between the execution of a `notify` operation and the acquisition of the lock by the waiting thread that receives the notification, another thread may intervene to acquire the lock. The impact of these semantics is that a thread resuming from a `wait` operation cannot assume that it will find a specific state as left by the notifying thread. A thread continuing from a `wait` can assume that the lock invariant holds, but no more.

In general, thread priority has no bearing on Java's weak notification semantics. For example, a higher priority thread is *not* favored over a lower priority one in the selection of which should receive a notification. To illustrate the consequence of this, consider that two lower priority threads might starve a higher priority thread, if notification always happens to pass from one of the lower priority threads to the other. Thread priorities may be used to control the relative progress of threads when they are *not* interacting, but not when the are. For this reason, thread priority is ignored until Section 5.

2. Haphazard Notification

The semantics of Java locks naturally lead to patterns in which a `synchronized` method waits in a loop until a required guard condition is met:

```
synchronized void f() {
  while( !guardCondition() )
    wait(); // 6
  executeOperation();
  notifyAll();
}
```

[2] J. Gosling, et al., The Java Language Specification, Addison-Wesley, 1996, http://www.aw.com/cp/javaseries.html.

[3] C.A.R Hoare, Monitors: An Operating System Structuring Concept. CACM 17:10, pp. 549–557, 1974.

[4] In fact, the `synchronized` method is merely syntactic sugar for the true primitive: the `synchronized` statement.

[5] For a discussion of "fairness" see G. R. Andrews, Concurrent Programming, Addison-Wesley, 1991, pp. 83–86

[6] The exception that may be thrown by `wait` is ignored in this code fragment.

The `notifyAll` primitive is a variation of `notify` in which all threads currently waiting on the lock are notified, and may start competing for the lock. In general, the `notifyAll` form must be used because there is no way to guarantee that a `notify` operation would resume a thread that is in a position to make progress. The use of `notifyAll` results in each waiting thread eventually[7] retesting its guard condition. Those threads that may proceed to use the resource will in turn notify all waiting threads.

Gosling et al.[8] recommend this form as "good practice." Arnold and Gosling [9] state that "The condition test should *always* be in a loop." Lea[10] describes it as the "standard coding idiom." The pattern has also been recommended for use in other languages. Lampson and Redell[11] state that, in Mesa, "The proper pattern of code for waiting is therefore:

```
WHILE NOT <OK to proceed> DO
    WAIT c
ENDLOOP
```

[7] Strictly, Java does not guarantee that every thread progresses. Relative thread priorities and the availability of processors determine which threads make progress.

[8] J. Gosling, et al., The Java Language Specification, Addison-Wesley, 1996, Chapter 16.

[9] K. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, 1996, Chapter 9.

[10] D. Lea, Concurrent Programming in Java, in preparation.

[11] B.W. Lampson, D.D. Redell, Experience with Processes and Monitors in Mesa, CACM 23:2, 105–11, 1981.

Birrell[12] offers "... the following general pattern, which I strongly recommend for all your uses of condition variables.

```
WHILE NOT expression DO
    Thread.Wait(m,c)
END;
```

Listing 1 shows a Java solution to Dijkstra's "Dining Philosophers" problem[13] using Haphazard Notification. The solution is deadlock-free, because even-numbered philosophers pick up their left fork first, and odd-numbered philosophers pick up their right fork first.[14] A weakness of this solution is that it does not guarantee fairness: a philosopher may wait without bound to acquire a given fork while another philosopher repeatedly acquires and releases that fork. The `putDown` method of class `Fork` uses `notify`, rather than `notifyAll`, as an optimization, since there is no point in notifying more that one waiting thread. However, as discussed above, the use of `notify` does *not* guarantee that the notified thread will be the next thread to acquire the lock. In particular, the thread that performs the notification may acquire the lock again for itself before the notified thread manages to do so.

The solution would become fair if a mechanism can be found that forces competing threads to alternate in their acquisition of a fork. How should this be accomplished?

[12] A.D. Birrell, An Introduction to Programming with Threads, Tech. Report #35, Systems Research Center, Digital Equipment, Palo Alto, CA, 1989.

[13] E.W. Dijsktra, Hierarchical Ordering of Sequential Processes, Acta Informatica, 1:2, 115–138, 1971.

[14] T.A. Cargill, A Robust Distributed Solution to the Dining Philosophers Problem, Software — Practice and Experience, 12, 965–969, 1982.

3. Specific Notification Pattern

The Specific Notification pattern addresses the arbitrary scheduling of Haphazard Notification.

3.1 Problem

A family of threads must cooperate to synchronize their use of a shared resource. In general, a thread must defer its use of the resource until the resource achieves an appropriate state. Many threads may wait for the resource concurrently, and for diverse reasons.

3.2 Context

Specific Notification is applicable when the arbitrary selection of a thread by `notify`, or the arbitrary scheduling of threads following `notify` and `notifyAll` provide insufficient control with respect to the order in which threads acquire a resource. The lack of ordering may be unsatisfactory merely because it is unfair, or because an application-specific policy constrains the order in which threads must acquire the resource. (An application-specific policy is illustrated in Section 4.)

3.3 Solution

Remove thread selection and scheduling from the purview of the built-in primitives, and place it explicitly under program control. To do this, create a *separate lock object for each set of threads that must be notified together.* Notification may then be applied to a specific lock object, which results in the activation of precisely the corresponding set of threads. If the set of threads is known to be a singleton, then exactly one thread is activated, by the use of `notify`. For an arbitrary set of threads, `notifyAll` should be used.

In general, the only property shared by the threads waiting for a specific notification lock is that they are to be reactivated simultaneously. The number of lock objects is therefore determined by the number of such sets of threads. For some resources, a fixed set of locks may suffice; for other resources, the set of locks may be determined dynamically, and may even grow without bound.

3.4 Example

Listing 2 shows the `Fork` class from Listing 1 rewritten to use Specific Notification. Each `Fork` object has a `Vector`[15] of lock objects called `snq` (specific notification queue). As a thread enters the `pickUp` method (which is *not* a `synchronized` method), it creates a new object, `snl` (specific notification lock), for exclusive use as its specific notification lock. If the thread must `wait`, it must leave itself holding the lock on `snl`. It therefore locks `snl` *before* locking `this` (the `Fork` object). The thread cannot block when acquiring `snl`, because the identity of `snl` is local. The thread may block in trying to acquire `this`, just as it might have blocked when entering the `synchronized` version of `pickUp` (Listing 1). Having acquired the lock on the `Fork` object, the thread determines whether or not it may proceed to use the `Fork` object. If any other thread has a lock object in the specific notification

[15] Class `Vector` is a member of the standard `java.util` package.

queue, this thread must `wait`. Before waiting, the thread releases the lock on the `Fork` object. The `wait` operation is applied specifically to `snl`. Any `snl` object is used at most once as a lock.

The `putDown` method, which remains `synchronized`, is simpler. First, a thread that is relinquishing the `Fork` object removes its own specific notification lock from the head of the specific notification queue. If the queue is non-empty, then there is at least one thread waiting to acquire this `Fork` object. Before notifying the thread at the head of the queue, the active thread must acquire the corresponding specific notification lock. The `notify` operation activates exactly the designated thread. No other thread may intervene ahead of the designated thread.

3.5 Costs

Specific Notification incurs a programming cost, which must not be overlooked. The programmer must assume greater responsibility in terms of maintaining the integrity of shared objects and the progress of threads. The benefits of increased control over threads are paid for with greater diligence in design and coding. If the behavior of Haphazard Notification is acceptable, there is no reason to introduce this burden.

The execution costs of Specific Notification are hard to characterize, but are generally better conditioned than those of Haphazard Notification. Where thread contention is low, Haphazard Notification is efficient because `notifyAll` operations act on queues that are expected to be empty. Under low contention, Specific Notification goes further, by eliminating redundant notifications entirely, usually at the cost of more mechanism when notification is required. Under high contention, the performance of Haphazard Notification degrades: in general, of a large set of threads that are activated to re-evaluate their guard conditions, only a few (usually zero or one) actually make useful progress. The performance of Specific Notification does not degrade under high contention: the notification operations performed remain exactly those that are required.

4. Further Examples

This section offers two further examples of Specific Notification.

4.1 Reader and Writers

Listing 3 shows part of a solution to the "Readers and Writers" problem[16]. Class `Resource` has methods `beforeRead`, `beforeWrite`, `afterRead` and `afterWrite`, which must be called before and after their respective operations. The integer fields `readers` and `writers` record the number of active readers and writers, respectively, preserving the invariants `(readers==0` or `writers==0)` and `writers<=1`. The `readersWaiting` field records the number of readers waiting to acquire the resource. All reader threads wait on the `Resource` object itself and are notified collectively by `notifyAll` in method `afterWrite`. This illustrates Specific Notification of a set of threads. Note that `beforeRead` does not use a `while`

[16] Courtois, Heymans, Parnas, Concurrent Control with "readers" and "writers," CACM 14:10, 667–668, Oct., 1971.

loop to test its guard condition, as it would under Haphazard Notification. Exactly the set of threads notified from `afterWrite` become the active readers (possibly joined by later arrivals). Fairness among writer threads is achieved by using Specific Notification to individual threads. Writer threads wait on lock objects that are enqueued in a `Vector` called `writerQ`. The Specific Notification is performed in the method `notifyWriter`, which is called from both `afterRead` and `afterWrite`, when needed.

4.2 NNTP Client

Listing 4 is abstracted from a multi-threaded NNTP client, which uses Specific Notification. Each thread that enters the client object with a query for the NNTP server calls `sendAndWait` to transmit a sequence of commands to the server. Responses from the server are returned in the order they were transmitted. Client threads must therefore be directed to read their responses in the same order that they sent their requests. After transmitting its request, if another thread is seen ahead, the thread waits in `sendAndWait` on a Specific Notification lock placed in a queue. After waiting (if necessary) and then reading a response from the server, each thread calls `afterReading` to `notify` exactly the thread behind it, if there is one.

5. Speculation

The Specific Notification pattern may be a member of a pattern language that addresses Java's weak context switching semantics in general. Another problem is to arrange that a set of *independent* threads, which have no intrinsic reason to communicate, share a set of processors such that all of the threads make progress. A Java virtual machine may timeslice the execution of threads at the same priority, but is not obliged to do so. The semantics of the `Thread.yield` primitive are not strong enough to guarantee that N (or more) threads can share N-1 processors fairly. As with Specific Notification, the solution is to program explicitly with the weak primitives such that the virtual machine is forced to produce the desired effect. The problem can be solved by adding a high priority thread that awakens periodically and adjusts the relative priorities of the other threads, so that each is guaranteed to dominate from time to time, and therefore make progress.

6. Summary

Using Specific Notification, a Java program takes responsibility for explicitly determining the set of threads to be activated by a `notify` operation, rather than subject itself to the arbitrary built-in semantics. The additional implementation complexity is warranted in programs where correctness or fairness considerations make haphazard synchronization intolerable.

7. Acknowledgments

Discussions with Doug Lea helped to clarify my thinking on several topics addressed in this paper. Thanks also to Adam McClure and Hans Rohnert for their comments on earlier drafts.

Listing 1: Dining Philosophers using Haphazard Notification

```
class Fork {
private boolean free = true;
synchronized void pickUp(){
   while( !free ) {
      try {
         wait();
      }
      catch(Exception exc) {
         Thread.currentThread().stop();
      }
   }
   free = false;
}
synchronized void putDown() {
   free = true;
   notify();
}
}

class Phil implements Runnable {
private Fork a, b;
Phil(Fork a, Fork b) {
   this.a = a;
   this.b = b;
}
public void run(){
   while( true ){
      // thinking
      a.pickUp();
      b.pickUp();
      // eating
      b.putDown();
      a.putDown();
   }
}
}

class Dining {
public static void main(String[] argv){
   int size = 5;
   Fork[] forks = new Fork[size];
   for( int i=0; i<size; ++i )
      forks[i] = new Fork();
   for( int i=0; i<size; ++i ){
      Fork a = forks[i];
      Fork b = forks[(i+1)%size];
      Phil p;
      if( i%2==0 )
         p = new Phil(a, b);
      else
         p = new Phil(b, a);
      new Thread(p).start();
   }
}
}
```

Listing 2: Class `Fork` using Specific Notification

```
class Fork {
private Vector snq = new Vector();
void pickUp(){
   Object snl = new Object();
   boolean mustWait;
   synchronized( snl ) {
      synchronized( this ) {
         mustWait = !snq.isEmpty();
         snq.addElement(snl);
      }
      if( mustWait ) {
         try {
            snl.wait();
         }
         catch(Exception exc) {
            Thread.currentThread().stop();
         }
      }
   }
}
synchronized void putDown() {
   snq.removeElementAt(0);
   if( !snq.isEmpty() )
      synchronized(snq.firstElement() ) {
         snq.firstElement().notify();
      }
}
}
```

Listing 3: Readers and Writers using Specific Notification

```
class Resource {
private int writers = 0, readers = 0, readersWaiting = 0;
private Vector writerQ = new Vector();

synchronized void beforeRead() {
   if( writerQ.size()==0 && writers==0 ) {
      readers += 1;
      return;
   }
   readersWaiting += 1;
   try {
      wait();
   } catch(Exception e) { Thread.currentThread().stop(); }
}

void beforeWrite() {
   Object snl = new Object();
   synchronized(snl) {
      synchronized(this) {
         if( writerQ.size()==0 && writers+readers==0 ) {
            writers += 1;
            return;
         }
         writerQ.addElement(snl);
      }
      try {
         snl.wait();
      } catch(Exception e) { Thread.currentThread().stop(); }
   }
}

synchronized void afterRead() {
   readers -= 1;
   if( readers==0 )
      notifyWriter();
}

synchronized void afterWrite() {
   writers -= 1;
   readers = readersWaiting;
   readersWaiting = 0;
   if( readers > 0 )
      notifyAll();
   else
      notifyWriter();
}

private void notifyWriter() {
   if( writerQ.size() > 0 ) {
      Object snl = writerQ.firstElement();
      writerQ.removeElementAt(0);
      synchronized(snl) {
         snl.notify();
      }
      writers += 1;
   }
}
```

Listing 4: Methods Excerpted from an NNTP Client

```
private void sendAndWait(String groupName, String cmd) {
   Object lock = new Object();
   synchronized(lock){
      boolean ready;
      synchronized(this){
         sendAsync("group "+groupName);
         sendAsync(cmd);
         ready = snq.isEmpty();
         snq.addElement(lock);
      }
      if( !ready )
         try {
            lock.wait();
         }
         catch(Exception e) { fatal(e); }
   }
}

private synchronized void afterReading() {
   snq.removeElementAt(0); // remove me
   if( !snq.isEmpty() )
      synchronized(snq.elementAt(0)) {
         snq.elementAt(0).notify();
      }
}
```