Chapter 3: Communication and Synchronization

Overview

Introduction and chapter structure

Chapter 2 gave some context for the Java concurrency model and explored thread creation in detail. This chapter focuses on how threads can communicate and synchronize their activities. Communication and synchronization between Java threads is supported via a simple form of monitor. The model is explored in detail, and its strengths and limitations are discussed.

Underlying Java's communication and synchronization approach is the Java Memory Model. The criticisms that have recently been made of this model are detailed, and the changes that have occurred in Java 1.5, as a result of these criticisms, are considered. Finally, ways in which threads can interact asynchronously with each other are addressed.

3.1 Synchronized Methods and Statements

Associated with each object there is a mutual exclusion lock. This lock cannot be accessed directly by the application, but it is affected by

- the method modifier `synchronized`, and
- block synchronization.

When a method is labeled with the `synchronized` modifier, access to the method can only proceed once the lock associated with the object has been obtained. Hence, synchronized methods have mutually exclusive access to the data encapsulated by the object, *if that data is only accessed by other synchronized methods*. Nonsynchronized methods do not require the lock, and can, therefore, be called at any time. Hence, to obtain full mutual exclusion, every method that accesses encapsulated data must be labeled synchronized. A simple shared integer is, therefore, represented by

```
class SharedInteger {
  public SharedInteger(int initialValue) {
    theData = initialValue;
  }

  public synchronized int read() {
    return theData;
  }

  public synchronized void write(int newValue) {
    theData = newValue;
  }

  public synchronized void incrementBy(int by) {
    theData = theData + by;
  }

  private int theData;
}
SharedInteger myData = new SharedInteger(42);
```

Block synchronization provides a mechanism whereby a block of code can be labeled as synchronized. The `synchronized` keyword takes as a parameter an object whose lock it needs to obtain before it can continue. Hence, synchronized methods are effectively implementable as (using the above `read` method as an example)

```
public int read() {
  synchronized (this) {
    return theData;
  }
  }
```

where **this** is the Java mechanism for obtaining the current object.

Warning Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms: that of encapsulating synchronization constraints associated with an object into a single place in the program. This is because it is not possible to understand the synchronization associated with a particular object, O, by just looking at O itself. It is necessary to look at all objects that name O in a synchronized statement. However, with careful use, this facility augments the basic model and allows more expressive synchronization constraints to be programmed.

Accessing synchronized data

Consider a simple class that implements a two-dimensional coordinate that is to be shared between two or more threads. This class encapsulates two integers, whose values contain the x and the y components. Writing to a coordinate is simple, the write method can be labeled as synchronized. Furthermore, the constructor method cannot (by definition) have any synchronization constraint:

```
  public class SharedCoordinate {
    public SharedCoordinate(int initX, int initY) {
      x = initX;
      y = initY;
    }

    public synchronized void write(int newX, int newY) {
      x = newX;
      y = newY;
    }
    ...

  private int x, y;
  }
```

The problem comes in deciding how to read the value of the coordinates. Functions in Java can return only a single value, and parameters to methods are passed by value. Consequently, it is not possible to have a single read method that returns both the x and the y components. If two synchronized functions readX and readY are used, it is possible for the value of the coordinate to be updated in between the calls to readX and readY. The result will be an inconsistent value of the coordinate.

There are essentially two ways of circumventing this problem. The first is to return a new Coordinate object whose values of the x and y fields are identical to the shared coordinate. This new object can then be accessed without fear of it being changed. Of course, the returned coordinate is only a snapshot of the shared coordinate, which might be changed by another thread immediately after the read method has returned. However, the individual field values now read will be consistent.

The following class illustrates the approach.

```
  public class SharedCoordinate {
    ...

    public synchronized SharedCoordinate read() {
      return new SharedCoordinate(x, y);
    }
    public int readX() { return x; }
      public int readY() { return y; }
  }
```

Once the returned coordinate has been used, it can be discarded and made available for garbage collection.

Using synchronized blocks

If efficiency is a concern, unnecessary object creation and garbage collection should be avoided. In the current example, this can be achieved by ensuring that any calls to readX and readY that need to see a consistent point value are encapsulated in a synchronized block.

```
  public class SharedCoordinate {
    public SharedCoordinate(int initX, int initY) {
      x = initX;
      y = initY;
    }

    public synchronized void write(int newX, int newY) {
      x = newX;
      y = newY;
    }

    public int readX() { return x; }

    public int readY() { return y; }

    private int x, y;
  }

  ...

  SharedCoordinate point1 = new SharedCoordinate(0,0);

  synchronized (point1) {
    SharedCoordinate point2 = new SharedCoordinate(
                   point1. readX(), point1. readY());
  }
```

In this example, synchronized updates are done within the class but there is now an onus on the user of the class to provide the synchronization on reading when needed (this is termed *conditionally thread-safe access*, see Section 4.8).

Java allows a thread to acquire a lock of an object even if it has already acquired the lock. Hence, the `readX` and `readY` methods could be made synchronized for added safety, and a thread that wishes to read the two values as an atomic operation can safely enclose the reads in a synchronized statement naming the object (without fear of deadlock).

Finding the locks held by a thread and a thread's state

In general, it is not possible to determine the group of locks that a thread currently holds. However, it is possible (as of Java 1.4) for a thread to determine if it holds the lock on a particular object. This is achieved by the static method `holdsLock` in the `Thread` class.

```
  package java.lang;

  public class Thread extends Object implements Runnable {
    ...
    public static boolean holdsLock(Object obj);
    ...
  }
```

As of Java 1.5, it is also possible to determine the current run-time state of a thread (and, therefore, whether it is waiting for a lock) via the `getState` method (also in the `Thread` class). This returns an enumeration object where the literals correspond to the states identified in Section 2.6., with the addition of a timed waiting state (that is, a waiting state with an associated timeout).

```
  package java.lang;

  public class Thread extends Object implements Runnable {
    ...
    // These are Java 1.5 extensions.
    public static final enum State{BLOCKED, NEW, RUNNABLE, TERMINATED, TIMED_WAITING, WAITING);
    public State getState();

  ...
  }
```

Static variables

Although synchronized methods or blocks allow mutually exclusive access to data in an object, this is not adequate if that data is `static`. Static data is shared between all objects created from the class. In Java, every class has an associated `Class` object, and it is this object's lock that must be obtained when accessing static data. The lock may be accessed either by labeling a static method with the **`synchronized`** modifier or by identifying the class's `Class` object in a synchronized block statement. In the latter case, the object associated with the class can be obtained by calling a class literal. Note that the class-wide lock is not obtained when synchronizing on an object of the class. Hence to obtain mutual exclusion over a static variable requires the following (for example):

```
class StaticSharedVariable {
  ...
  // The example shows two possible ways of acquiring
  // a lock on a Class object.

  public int Read() {
    synchronized (StaticSharedVariable.class) {
      return shared;
    }
  }

  public synchronized static void Write(int I) {
    shared = I;
  }
  private static int shared;
}
```

Obtaining the lock of a `Class` object does not affect the locks of any instances of the class. They are independent.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

3.2 Waiting and Notifying

To obtain condition synchronization requires further support. This comes from methods provided in the predefined `Object` class:

```
package java.lang;

public class Object {
  ...
  // The following methods all throw the unchecked
  // IllegalMonitorStateException.

  public final void notify();
  public final void notifyAll();

  public final void wait() throws InterruptedException;
  public final void wait(long millis)
        throws InterruptedException;
  public final void wait(long millis, int nanos)
        throws InterruptedException;
  ...
}
```

These methods are designed to be used only from within methods that hold the object lock (that is, they are synchronized). If called without the lock, the exception `IllegalMonitorStateException` is thrown.

The `wait` method always blocks the calling thread *and releases the lock associated with the object*. If a thread is holding several locks (for example, in a nested monitor call), only the lock associated with the object being waited on is released. All other locks are maintained. An optional timeout can be used to stop the calling thread waiting indefinitely. However, this can be difficult to use, see Section 4.2

Warning The `wait (0)` or `wait (0, 0)` method calls are defined to be the same as `wait ()` rather than do not wait. This is slightly counterintuitive as `wait (0,1)` will timeout, whereas a `wait (0,0)` will not. Hence, care must be taken if timeout values are being calculated.

Important note The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language (however, it is defined by the RTSJ; see Section 14.2). Note that `notify` does not release the lock, and hence the woken thread must still wait until it can obtain the lock before it can continue. To wake up **all** waiting threads requires use of the `notifyAll` method; again this does not release the lock, and all the awoken threads must wait and contend for the lock when it becomes free. If no thread is waiting, then `notify` and `notifyAll` have no effect.

A waiting thread can also be awoken if it is interrupted by another thread. In this case, the `InterruptedException` is thrown.

Condition variables

Although it appears that Java provides the equivalent facilities to other languages supporting monitors, there is one important difference. There are *no* explicit condition variables. Consequently, an object cannot partition the waiting states, and, therefore, cannot have fine control over notification. When a thread is awoken, it cannot necessarily assume that a notify was associated with its wait state. For many algorithms this limitation is not a problem, as the conditions under which threads are waiting are mutually exclusive.

For example, the bounded buffer traditionally has two condition variables: `BufferNotFull` and `BufferNotEmpty` each associated with the corresponding buffer state. If a thread is waiting for one condition, no other thread can be waiting for the other condition as the buffer cannot be both full and empty at the same time. Hence, one would expect that the thread can assume that when it wakes, the buffer is in the appropriate state.

Warning Unfortunately, this is not always the case. Java, in common with other monitor-like approaches (for example, POSIX mutexes), makes no guarantee that a thread woken from a `wait` will gain immediate access to the lock. Furthermore, a Java implementation is allowed to generate spurious wake-ups not related to the application.

Consider a thread that is woken after waiting on the `BufferNotFull` condition. Another thread could call the `put` method, find that the buffer has space and insert data into the buffer. When the woken thread eventually gains access to the lock, the buffer will again be full. Hence, it is usually essential for threads to reevaluate their conditions, as illustrated in the bounded buffer example below.

```
public class BoundedBuffer<Data> {
  public BoundedBuffer(int length) {
    size = length;
    buffer = (Data [] ) new Object [size] ;
    last = 0;
    first = 0;
  }

  public synchronized void put(Data item)
         throws InterruptedException {
    while (numberInBuffer == size) wait();
    last = (last + 1) % size;
    numberInBuffer++ ;
    buffer[last] = item;
    notifyAll();
  }

  public synchronized Data get()
         throws InterruptedException {
    while (numberInBuffer == 0) wait();
    first = (first + 1) % size;
    numberInBuffer--;
    notifyAll();
    return buffer[first];
  }

  private Data buffer[];
  private int first;
  private int last;
  private int numberInBuffer = 0;
  private int size;
}
```

Of course, if `notifyAll` is used to wake up threads, then it is more obvious that those threads must always reevaluate their conditions before proceeding.

Important note In general, many simple synchronization errors can be avoided in Java if

- all `wait` method calls are enclosed in `while` loops that evaluate the waiting condition
- the `notifyAll` method is used to signal changes in objects' states.

This approach, while safe, is potentially inefficient as spurious wake-ups will occur.

To improve performance, the `notify` method may be used when

- all threads are waiting for the same condition
- at most one waiting thread can benefit from the state change
- the JVM does not generate any wake-ups without an associated call to the `notify` and `notifyAll` methods on the corresponding object.

These first two requirements must, of course, also be met by any subclass.

The readers-writers problem

One of the standard concurrency control problems is the *readers-writers* problem. In this, many readers and many writers are attempting to access an object encapsulating a large data structure. Readers can read concurrently, as they do not alter the data; however, writers require mutual exclusion over the data, both from other writers and from readers. There are different variations on this scheme; the one considered here is where preference is always given to waiting writers. Hence, as soon as a writer is available, all new readers will be blocked until all writers have finished. Of course, in extreme situations this may lead to starvation of readers.

Important note The key to solving most concurrency control problems is to surround each operation with an *entry* and an *exit* protocol. The entry protocol determines if the conditions for the operation to proceed are right, and if not, blocks the calling thread until they are right. The exit protocol determines whether any blocked operations can now proceed. Data may be needed to keep track of the current state of the requested operations. This data must be accessed under mutual exclusion.

The solution to the readers-writers problem using standard monitors requires four monitor methods `startRead`, `stopRead`, `startWrite` and `stopWrite`. The first two methods implement the entry and exit protocol for the readers respectively. The second two implement the writers' protocols. The readers are consequently structured as follows:

```
startRead(); // entry protocol
  // Call object to read data structure.
stopRead(); // exit protocol
```

Similarly, the writers are structured:

```
startWrite(); // entry protocol
  // Call object to write data structure.
stopWrite(); // exit protocol
```

The code inside the monitor provides the necessary synchronization using two condition variables: `OkToRead` and `OkToWrite`. In Java, this cannot be directly expressed as there are no explicit condition variables. Two approaches for solving this problem are now considered. The first approach uses a single class:

```
public class ReadersWriters {
  // Preference is given to waiting writers.

  public synchronized void startWrite()
         throws InterruptedException {
    // Wait until it is ok to write.
    while(readers > 0 || writing) {
      waitingWriters++;
      wait();
      waitingWriters--;
    }
    writing = true;
  }

  public synchronized void stopWrite() {
    writing = false;
    notifyAll();
  }

  public synchronized void startRead()
         throws InterruptedException {
    // Wait until it is ok to read.
    while(writing || waitingWriters > 0) wait();
    readers++;
    }

    public synchronized void stopRead() {
      readers--;
      if(readers == 0) notifyAll();
    }
    private int readers = 0;
    private int waitingWriters = 0;
    private boolean writing = false;
  }
```

In this solution, on awaking after the `wait` request, a thread must reevaluate the conditions under which it can proceed. Although this approach will allow multiple readers or a single writer, arguably it is inefficient, as all threads are woken up every time the data becomes available. Many of these threads, when they finally gain access to the monitor, will find that they still cannot continue and, therefore, will have to wait again. It should also be noted that this solution is not tolerant to the `InterruptedException` being thrown. A waiting writer increments the `waitingWriters` count before waiting. If it is interrupted, the exception is propagated and the count is never decremented. One solution to this problem is to catch and propagate the exception.

```
  public synchronized void startWrite()
              throws InterruptedException {
    try {
      while(readers > 0 || writing) {
        waitingWriters++;
        wait();
        waitingWriters--;
      }
      writing = true;
    } catch(InterruptedException ie) {
      waitingWriters--; throw ie;
    }
  }
```

The alternative solution to the `InterruptedException` problem is to use the `finally` clause:

```
  public synchronized void startWrite()
         throws InterruptedException {
    while(readers > 0 || writing) {
      waitingWriters++;
      try {
        wait();
      } finally { waitingWriters--; }
    }
    writing = true;
  }
```

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

3.3 Implementing Condition Variables

An alternative solution to the readers-writers problem is to use another class to implement a simple condition variable. Consider

```
  public class ConditionVariable {
    public boolean wantToSleep = false;
    public boolean wakeUp = false;
  }
```

The general approach is to create instances of these variables (`OkToRead` and `OkToWrite`) inside another class and to use block synchronization. To avoid waiting in a nested monitor call, the flag `wantToSleep` is used to indicate whether the monitor wants to wait on the condition variable. The flag `wakeUp` is used to control the release of the threads and to cope with spurious JVM wakeups). The following algorithm illustrates the approach (in this instance preference is given to waiting readers):

```
  public class ReadersWriters2 {
    public void startWrite() throws InterruptedException {
      synchronized(OkToWrite) { // condition variable lock
      synchronized(this) { // monitor lock
        if(writing || readers > 0 || waitingReaders > 0) {
          waitingWriters++;
          OkToWrite.wantToSleep = true;
        } else {
          writing = true;
          OkToWrite.wantToSleep = false;
          OkToWrite.wakeUp = true;
          OkToRead.wakeUp = false;
        }
      } // Give up monitor lock.
      if(OkToWrite.wantToSleep) OkToWrite.wait();
      while (!OkToWrite.wakeUp) OkToWrite.wait();
      OkToWrite.wakeUp = false;
    } // Give up OkToWrite lock.
  }

  public void stopWrite() {
    synchronized(OkToRead) { // get locks in correct order
      synchronized(OkToWrite) {
        synchronized(this) {
          if(waitingReaders > 0) {
            writing = false;
            readers = waitingReaders;
            waitingReaders = 0;
            OkToRead.wakeUp = true;
            OkToRead.notifyAll();
          } else if(waitingWriters > 0) {
            waitingWriters--;
            OkToWrite.wakeUp = true;
            OkToWrite.notify();
          } else writing = false;
        } // Give up monitor lock.
      } // Give up OkToWrite lock.
    } // Give up OkToRead lock.
  }

  public void startRead() throws InterruptedException {
    synchronized(OkToRead){
      synchronized(this) {
        if(writing) {
          waitingReaders++;
          OkToRead.wantToSleep = true;
        } else {
          readers++;
          OkToRead.wantToSleep = false;
          OkToRead.wakeUp = true;
        }
      } // Give up monitor lock.
      if(OkToRead.wantToSleep) OkToRead.wait();
      while(!OkToRead.wakeUp) OkToRead.wait();
    } // Give up OkToRead lock.
  }

  public void stopRead() {
    synchronized(OkToWrite) {
      synchronized(this) {
        readers--;
        if(readers == 0 & waitingWriters > 0) {
          waitingWriters--;
          writing = true;
          OkToRead.wakeUp = false;
          OkToWrite.wakeUp = true;
            // Transfer the lock to first waiting writer.
          OkToWrite.notify();
        }
      }
    }
  }

  private int readers = 0;
  private int waitingReaders = 0;
  private int waitingWriters = 0;
  private boolean writing = false;

  private ConditionVariable OkToRead =
                new ConditionVariable();
    private ConditionVariable OkToWrite =
                new ConditionVariable();
  }
```

Every condition variable is represented by an instance of the `ConditionVariable` class declared inside the class that is acting as the monitor. Conditions are evaluated while holding the monitor lock and the lock on any condition variable that will be notified or waited on inside the monitor procedure. To ensure that no deadlock occurs, these locks should always be obtained in the same order. The Java language itself ensures that the locks are released in the reverse order to the one in which they were obtained. In this case, the acquisition order is always

```
  OkToRead
  OkToWrite
  ReadersWriters2
```

Note that in this example no while loops have been used around the testing of the conditions. This is because the conditions have explicitly been set so that if threads gain the lock for the first time ahead of those waking up, they will find that they cannot proceed and will enter the wait state. However, a while loop is still necessary to be protected against spurious JVM wake-ups.

Note also that this solution is not tolerant to the interrupted exception being thrown. As with the first readers-writers solution given in Section 3.2, the exception must be caught before being allowed to propagate. This is left as an exercise for the reader.

Java 1.5 and JSR 166 note

Within the Java Community Process there has been an activity [Java Community Process, JSR 166, 2002] that has proposed a set of concurrency-related utilities [Lea, 2004]. These have now been incorporated into `java.utils` in Java 1.5. One of the utilities developed is a general-purpose lock mechanism. This includes locks and condition variables that are accessed via the following interfaces.

```
  package java.util.concurrent.locks;
  public interface Lock {
    public void lock();
      // Uninterruptibly wait for the lock to be acquired.
    public void lockInterruptibly()
                   throws InterruptedException;
      // As above but interruptible.
    public Condition newCondition();
      // Create a new condition variable for use with the Lock.
    public boolean tryLock();
      // Returns true is lock is available immediately.
    public boolean tryLock(long time, TimeUnit unit)
                  throws InterruptedException;
      // Returns true is lock is available within a timeout.
      // See Section 4.2 for information on the TimeUnit class.
    public void unlock();
  }

  package java. util. concurrent. locks;

  public interface Condition {
    public void await() throws InterruptedException;
    /* Atomically releases the associated lock and
     * causes the current thread to wait until
     *  1. another thread invokes the signal method
     *     and the current thread happens to be chosen
     *     as the thread to be awakened; or
     *  2. another thread invokes the signalAll method;
     *  3. another thread interrupt the thread; or
     *  4. a spurious wake-up occurs.
     * When the method returns it is guaranteed to hold the
     * associated lock.
     */

    public boolean await(long time, TimeUnit unit)
           throws InterruptedException;
    public long awaitNanos(long nanosTimeout)
           throws InterruptedException;
    public void awaitUninterruptible();
      // As for await, but not interruptible.
    public boolean awaitUntil(java.util.Date deadline)
           throws InterruptedException;
      // As for await() but with a timeout, see Section 4.2
      // for information on TimeUnit class.
    public void signal();
      // Wake up one waiting thread.
    public void signalAll();
      // Wake up all waiting threads.
  }
```

A new lock and associated condition variables can be created via the `ReentrantLock` class:

```
  package java.util.concurrent.locks;
  public class ReentrantLock
         implements Lock, java.io.Serializable {
    public ReentrantLock();
    ...
    public void lock();
    public void lockInterruptibly() throws InterruptedException;
    public ConditionObject newCondition();
      // Create a new condition variable and associated it
      // with this lock object.
    public boolean tryLock();
    public boolean tryLock(long time, TimeUnit unit)
                  throws InterruptedException;
    public void unlock();
  }
```

Where `ConditionObject` is an inner class definition of the `ReentrantLock` class that implements the `Condition` interface.

Using these facilities it is possible to implement the bounded buffer using the familiar algorithm:

```
  import java.util.concurrent.*;
  public class BoundedBuffer<Data> {
    public BoundedBuffer(int length) {
      size = length;
      buffer = (Data[]) new Object[size];
      last = 0;
      first = 0;
      numberInBuffer = 0;
    }

    public void put(Data item)
           throws InterruptedException {
     lock.lock();
     try {
       while (numberInBuffer == size) notFull.await();
       last = (last + 1) % size;
       numberInBuffer++;
       buffer[last] = item;
       notEmpty.signal();
     } finally {
         lock.unlock();
    }
  }

  public synchronized Data get()
         throws InterruptedException {
     lock.lock();
     try {
       while (numberInBuffer == 0) notEmpty.await();
       first = (first + 1) % size;
       numberInBuffer--;
       notFull.signal();
       return buffer [first];
     } finally {
         lock.unlock();
     }
    }

    private Data buffer[];
    private int first;
    private int last;
    private int numberInBuffer;
    private int size;
    private Lock lock = new ReentrantLock();
    private final Condition notFull =
                  lock.newCondition();
    private final Condition notEmpty =
                  Lock.newCondition();
  }
```

Here, although it is still necessary to have the while loops, threads are only awoken when the condition on which they are waiting has been signalled.

The solution to the readers/writers problem is left as an exercise for the reader.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

3.4 Synchronization and the Java Memory Model

The previous sections have discussed how threads can safely communicate with each other using shared variables (and objects) encapsulated in monitors. As long as programmers ensure that all shared variables are accessed by threads only when they hold an appropriate monitor lock, they need not be concerned with issues such as multiprocessor implementations, compiler optimizations, whether processors execute instructions out-of-order, and so on. However, synchronization can be expensive, and there are times when a programmer might want to use shared variables without an associated monitor lock. One example is the so-called double-checked locking idiom [Schmidt and Harrison, 1997]. In this idiom, a singleton resource is to be created; this resource may or may not be used during a particular execution of the program. Furthermore, creating the resource is an expensive operation and should be deferred until it is required. A simple and intuitive implementation of this requirement is the following:

```
  public class ResourceController {

    public static synchronized Resource getResource() {
      if (resource == null) resource = new Resource();
      return resource;
    }

    private static Resource resource = null;
  }
```

The problem with this solution is that a lock is required on every access to the resource. In fact, it is only necessary to synchronize on creation of the resource, as the resource will provide its own synchronization when the threads use it. The double-checked locking idiom attempts to solve this with the following algorithm.

```
  public class ResourceController {

    public static Resource getResource() {
      if(resource == null) {
        synchronized (ResourceController.class) {
          if(resource == null) resource = new Resource();
        }
      }
      return resource;
    }

    private static Resource resource = null;
  }
```

Here, once the resource has been allocated, in theory, there is no need to execute the synchronized statement. In order to understand whether this program functions as intended, it is necessary to have a deeper understanding of both the relationship between Java threads and memory and the potential optimizations that a compiler or processor may perform.

The relationship between threads and memory is defined in the Java Language Specification Chapter 17 [Gosling, Joy and Steele, 1996] and is known as the Java Memory Model (JMM). Unfortunately, this model has come under much criticism over recent years because it is hard to understand [Pugh, 1999]; as a result it has been revamped in Java 1.5. In the JMM, each thread is considered to have access to its own working memory as well as the main memory that is shared between all threads. This working memory is used to hold copies of the data that resides in the shared main memory. It is an abstraction of data held in registers or data held in local caches on a multiprocessor system. The JVM transfers data between the main shared memory and a thread's local memory as and when required. It is a requirement that

- a thread's working memory is invalidated when the thread acquires an object's lock; that is, inside a synchronized method or statement any initial read of a shared variable must read the value from main memory,
- a thread's working memory is written back to the main memory when the thread releases a lock; that is, before a synchronized method or statement finishes, any variables written to during the method or statement must be written back to main memory.

Data may be written to the main memory at other times as well, however, the programmer just cannot tell when.

In order to give flexibility to compiler writers and JVM implementors, the JMM allows code to be optimized and reordered as long as it maintains "as-if-serial" semantics. That is, the result of executing the code is the same as the result that would be obtained if the code was executed sequentially. For sequential Java programs, the programmer will not be able to detect these optimizations and reordering. However, in concurrent systems, they will manifest themselves unless the program is properly synchronized.

Consider again the double-checked locking algorithm. Now suppose that a compiler implements the `resource = new Resource ()` statement logically as follows:

```
tmp = create memory for the Resource class
    // tmp points to memory
Resource.construct(tmp)
    // runs the constructor to initialize
resource = tmp // set up resource
```

Now as a result of optimizations or reordering, suppose the statements are executed in the following order

```
tmp = create memory for the Resource class
    // tmp points to memory
resource = tmp
Resource.construct(tmp)
    // run the constructor to initialize
```

It is easy to see that there is a period of time when the `resource` reference has a value, but the `Resource` object has not been initialized. It is now possible to construct an interleaving of the double-checked locking algorithm, when one thread is in the process of creating the resource, a second thread sees a partially created object (outside of the synchronized block) and tries to use it. It is not possible to predict what will happen as it depends on the resource itself [Pugh 2000].

Even more insidious problems may occur if the resource is fully initialized by thread, `T1`, but the initialization touches other objects. Now these objects may have been written back to memory when `T1` exits the synchronized statement, but another thread, `T2`, will see an initialized resource that potentially references objects that it already has in its local memory. Unfortunately, as it has not performed a lock operation, there is no requirement for the JVM to reload those objects, and so `T2` sees stale data.

Important note The double-checked locking algorithm illustrates that synchronized methods (and statements) in Java serve a dual purpose. Not only do they enable mutual exclusive access to a shared resource but they also ensure that data written by one thread (the writer) becomes visible to another thread (the reader). The visibility of data written by the writer is only guaranteed when it releases a lock that is subsequently acquired by the reader.

Volatile fields

Java allows fields to be defined as *volatile*. The Java Language Specification requires that a volatile field not be held in local memory and that all reads and writes go straight to main memory. Furthermore, operations on volatile fields must be performed in exactly the order that a thread requests. A further rule requires that volatile double or long variables must be read and written atomically.

Warning Objects and arrays are accessed via references and, therefore, marking them as volatile only applies to the references, not to the fields of the objects or the components of the arrays. It is not possible to specify that elements of an array are volatile.

Visibility and synchronization points

The Java 1.5 Java Memory Model has changed as a result of a Java Specification Request (JSR 133 [Java Community Process, JSR133]). One outcome is that more attention has been paid to the point at which changes made to shared data become visible to other threads. For example, threads are synchronized in situations other than via the synchronized method (statement). In particular, the following are considered additional synchronization points [Manson and Pugh, 2004]:

- ¿ when one thread starts another - changes made by the parent thread before the start requests are visible to the child thread when it executes;
- ¿ when one thread waits for the termination of another (for example, by using the `join` or `isAlive` methods in the `Thread` class) - changes made by the terminating thread before it terminates are visible to the waiting thread;
- ¿ when one thread interrupts another (see Section 3.5) - changes made by the interrupting thread before the interrupt request are made visible to the interrupted thread when it next tests the interrupted flag;
- ¿ when threads read and write to the same volatile field - changes made by the writer thread to shared data (before it writes to the volatile field) are made visible to a subsequent reader of the same volatile field.

The tightening up of the definition of synchronization points have removed many of the uncertainties of the old JMM and allows more precise semantics to be given to multithreaded Java programs.

Team LiB ◂ PREVIOUS NEXT ▸
Team LiB ◂ PREVIOUS NEXT ▸

3.5 Asynchronous Thread Control

Early versions of Java allowed one thread to affect another thread asynchronously through the following methods.

```
package java.lang;

public class Thread extends Object implements Runnable {
  ...
  // The following methods all throw the
  // unchecked SecurityException.
  public final void suspend(); // DEPRECATED
  public final void resume(); // DEPRECATED
  public final void stop(); // DEPRECATED
  public final void stop(Throwable except);// DEPRECATED
  ...
}
```

Suspend and resume

`Suspend` instructs the JVM to remove the associated thread from the set of runnable threads. It is typically called by a thread that is waiting for another thread to signal an event. When the other thread has caused the event, it instructs the JVM to add the suspended thread to the set of runnable threads by calling the suspended thread's `resume` method. Consider, for example, simple condition synchronization using a `flag`. One thread, `T2`, sets the flag, another thread, `T1`, waits until the `flag` is set and then clears it. This would be represented in Java:

```
boolean flag;
final boolean up = true;
final boolean down = false;
class FirstThread extends Thread {
  public void run() {
    ...
    if(flag == down) suspend();
    flag = down;
    ...
  }
}

class SecondThread extends Thread { // T2
  FirstThread T1;
  public SecondThread(FirstThread T) {
      super();
      T1 = T;
    }
    public void run() {
      ...
      flag = up;
      T1.resume();
      ...
    }
  }
```

Unfortunately (even ignoring those problems outlined in Section 3.4), this approach suffers from a *race condition*. Thread `T1` could test the `flag` and then the JVM could decide to preempt it and run `T2. T2` sets the `flag` and resumes `T1. T1` is, of course, not suspended, and so the `resume` has no effect. Now, when `T1` next runs, it thinks the `flag` is `down` and, therefore, suspends itself.

The reason for this problem is that the `flag` is a shared resource that is being tested for true or false, and a subsequent action is being performed that depends on the result of that test (the thread is suspending itself). This testing and suspending is not an atomic operation and therefore interference can occur from other threads. The correct way to program this interaction is to encapsulate the variable in a class and use synchronized methods with embedded `wait` and `notify` method calls as illustrated in Section 3.1.

The use of `suspend` by one thread to suspend another thread is even more dangerous, as the suspended thread may be holding resources needed by other threads. The resulting anarchy may lead to the system becoming deadlocked. It is for these reasons the methods are deprecated in the current version of Java.

As already mentioned (see Section 2.4), the `stop` method causes the associated thread to stop its current activity and throw a `ThreadDeath` exception, and similarly with the `stop (Throwable except)` method, only this time the exception passed as a parameter is thrown. Again, use of these methods is unsafe, and they have been deprecated in the current language. Consequently, they (and `suspend` and `resume`) should not be used.

Thread interruption

Conventional Java now supports only the following methods that give a *limited* form of asynchronous thread interaction.

```
  public class Thread extends Object implements Runnable {

    ...
    public void interrupt();
      // Throws unchecked SecurityException.
      // Send an interrupt to the associated thread.
    public boolean isInterrupted();
      // Returns true if associated thread has been
      // interrupted, the interrupt status is unchanged.
    public static boolean interrupted();
      // Returns true if the current thread has been
      // interrupted and clears the interrupt status.
    ...
  }
```

One thread can signal an interrupt to another thread by calling the `interrupt` method. The result of this depends on the current status of the interrupted thread.

- If the interrupting thread does not have the appropriate security permissions, the `SecurityException` is thrown in the interrupting thread.

- If the interrupted thread is blocked in the `wait`, `sleep` or `join` methods, it is made runnable and has the `InterruptedException` thrown.

- If the interrupted thread is executing, a status flag is set, indicating that an interrupt is outstanding. *There is no immediate effect on the interrupted thread*. Instead, the called thread must periodically test to see if it has been "interrupted" using the `isInterrupted` or the `interrupted` methods. If it does not test and subsequently attempts to block in the `wait`, `sleep` or `join` methods, then the `InterruptedException` is thrown immediately.

Warning Given that there is no guarantee that an interrupted thread will be interrupted in a timely manner, this facility is more of a synchronous notification method. It is not adequate for asynchronous notification in real-time systems. Consequently, thread interruption is one of the main areas that the RTSJ has addressed (see Chapter 7).

Team LiB PREVIOUS NEXT

Team LiB PREVIOUS NEXT

3.6 Summary

Communication and synchronization are fundamental to concurrent programming. Programming errors in these areas are notoriously difficult to detect. Apparently working programs can suddenly suffer from deadlock or livelock.

The Java model revolves around controlled access to shared data using a monitor-like facility. The monitor is represented as an object with synchronized methods and statements providing mutual exclusion. Condition synchronization is given by the `wait`, `notify` and `notifyAll` methods. True monitor condition variables are not directly given by the language, however, the package `java.util.concurrent. locks` provides this common locking paradigm. Therefore, they do not have to be reimplemented by the programmer.

It is important to realize that communication via nonvolatile data outside synchronized methods/statements is inherently unsafe unless the threads are synchronized by thread creation or by thread termination or by thread interruption.

Another key component of the Java communication and synchronization model is asynchronous thread control. This allows a thread to affect the progress of another thread without the threads agreeing in advance as to when that interaction will occur. There are two aspects to this: suspending and resuming a thread (or stopping it altogether), and interrupting a thread. The former are now deemed to be unsafe because of their potential to cause deadlock and race conditions. The latter is not responsive enough for real-time systems.

Team LiB PREVIOUS NEXT