

Java: Threads and Monitors

Java supports concurrency via *threads*. These threads are lightweight processes and they synchronise via monitors.

Threads

A Java thread is a lightweight process that has its own stack and execution context, but also has access to all variables in its scope. Threads are programmed by either extending the **Thread** class or by implementing the **Runnable** interface. Both of these are part of the standard **java.lang** package.

An instance of a thread is created by:

```
Thread myProcess = new Thread ( );
```

To start the new thread running you simply execute:

```
MyProcess.start ( );
```

The **start** method invokes a **run** method in the thread. Since we haven't defined the **run** method, the code above does nothing. We can define the **run** method by extending the **Thread** class.

```
class myProcess extends Thread
{
    public void run ( )
    {
        System.out.println ("Hello from the thread");
    }
}

myProcess p = new myProcess ( );
p.start ( );
```

If you don't need to keep a reference to the new thread we can do away with **p** and simply write:

```
new myProcess ( ).start ( );
```

As well as extending the **Thread** class, we can create lightweight processes by implementing the **Runnable** interface. The advantage of this approach is that you can make one of your own classes, or a system-defined class, into a process. You cannot do this with threads since Java only allows you to extend one class at a time.

Using the **Runnable** interface the previous example becomes:

Class myProcess implements Runnable

```
{  
    public void run ( )  
    {  
        System.out.println ("Hello from the thread");  
    }  
}
```

```
Runnable p1 = new myProcess ( );  
New Thread (p1). start ( );
```

The preferred way to terminate a thread is to allow the **run** method to terminate. There is a **stop** method that terminates a thread no matter what it is doing, but this has been deprecated in recent versions of Java (for very good reasons).

If a thread has nothing immediate to do (for example it updates the screen every second) then you should suspend the thread by putting it to sleep for a specified time. There are two sleep methods, **static void sleep (long milliseconds)** and **static void sleep (long milliseconds, int nanoseconds)**. Be careful because most operating systems have a minimum time resolution greater than 1 millisecond, so the JVM rounds the requested sleep period to the smallest time resolution the operating system supports.

The **join** method waits for the completion of the specified thread and provides some basic synchronisation. It can be supplied with a timeout period (like the **sleep** methods).

```
...
try
{
    otherThread.join (1000);    // wait for 1 sec
}
catch (InterruptedException e ) {}
...
```

Synchronisation in Threads

Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables. To prevent two threads from interfering with each other when updating a shared variable, Java provides synchronisation via a slimed-down monitor.

Java's keyword **synchronized** provides mutual exclusion and can be used with a group of statements or with an entire method.

The following class will potentially have problems if its **update** method is executed by several threads concurrently.

```

class Problematic
{
    private int data = 0;
    public void update ( )
    {
        data++;
    }
}

```

An executing instance of the **update** method needs exclusive access to the variable **data**.

```

class ExclusionByMethod
{
    private int data = 0;
    public synchronized void update ( )
    {
        data++;
    }
}

```

This is a simple monitor where the monitor's permanent variables are private variables in the class; and the monitor procedures are implemented as **synchronized** methods. There is only one lock per object in Java so when a synchronized method is invoked it waits to obtain the lock, execute the method, and then releases the lock.

Another way to implement mutual exclusion is to use the **synchronize** statement within the body of a method.

```

class ExclusionByGroup
{

```

```

private int data = 0;
public void update ( )
{
    synchronized (this) // lock this object for the
    {                     // following group of statements
        data++;
    }
}
}

```

The keyword **this** refers to the object invoking the **update** method. The lock is obtained on the invoking object.

A **synchronized** statement specifies that the following group of statements is executed as an atomic, non-interruptible, action. A **synchronized** method is equivalent to a monitor procedure.

While Java does not explicitly support condition variables, there is one implicitly declared for each synchronised object. The **wait** and **notify** methods are similar to the **wait** and **signal** operations in SR but they can only be executed within **synchronized** portions of the code (i.e. when an object is locked).

The **wait** method releases the lock on an object and suspends the executing thread. There is a single delay queue per object (usually a FIFO but not necessarily).

The **notify** method awakens the thread at the front of the object's delay queue. **notify** has *signal and continue* semantics, so the thread invoking **notify** continues to hold the lock on the object. The awakened

thread will execute at some future time when it can reacquire the lock on the object. Java also has a **notifyAll** method that is similar to **signal_all** in SR.

Readers/Writers Problem in Java

```
class ReadersWriters
{
    private int data = 0;    // our database
    private int nr = 0;

    private synchronized void startRead ( )
    {
        nr++;
    }

    private synchronized void endRead ( )
    {
        nr--;
        if (nr == 0)
            notify ( );    // awaken a waiting writer
    }
}
```

```
public void read ( )
{
    startRead ( );
    System.out.println ("read: " + data);
    endRead ( );
}
```

```
public synchronized void write ( )
{
    while (nr > 0)
        try
        {
            wait ( );           // delay any active readers
        }
        catch (InterruptedException ex)
        {
            return;
        }
    data++;
    System.out.println ("write: " + data);
    notify ( );               // awaken a waiting writer
}
}
```

```
class Reader extends Thread
{
    int rounds;
    ReadersWriters RW;
```

```

Reader (int rounds, ReadersWriters RW)
{
    this.rounds = rounds;
    this.RW = RW;
}

public void run ( )
{
    for (int i = 0; i < rounds; i++)
        RW.read ( );
}
}

```

```

class Writer extends Thread
{
    int rounds;
    ReadersWriters RW;
}

```

```

Reader (int rounds, ReadersWriters RW)
{
    this.rounds = rounds;
    this.RW = RW;
}

public void run ( )
{
    for (int i = 0; i < rounds; i++)
        RW.write ( );
}
}

```

```

class RWProblem
{

```

```

static ReadersWriters RW = new ReadersWriters ( );

public static void main (String[ ] args)
{
    int rounds = Integer.parseInt (args[0], 10);
    new Reader (rounds, RW).start ( );
    new Writer (rounds, RW).start ( );
}
}

```

This piece of Java code gives reader preference over writers. As an exercise can you figure out how to make it fair?

Threads communicate primarily by sharing access to memory. This is very efficient, but inherently dangerous:

- Thread interference or interleaving
- Memory consistency errors (different threads have inconsistent views of what should be the same data)

So careful synchronization is needed, as we have seen.

Java: Higher Level Concurrency

Up to now, we have focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but we need higher-level building blocks for more advanced tasks (especially for massively parallel applications exploiting current multiprocessor and multi-core systems).

In this section we'll examine some high-level concurrency features introduced in Java 5.0. Most of these features are implemented in the new `java.util.concurrent` packages. There are also new concurrent data structures in the Java Collections Framework.

- Lock **objects** (described more fully below) support locking idioms that simplify many concurrent applications. Though similar, they should not be confused with their *implicit* cousins seen above.
- `Executors` define a high-level API for launching and managing threads. `Executor` implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
- Concurrent collections include concurrent support for the management of large collections of data in `HashTables`, different kinds of `Queues` etc; they can greatly reduce the need for synchronization.
- Atomic variables (eg `AtomicInteger`) are defined in the `java.util.concurrent.atomic` package and support atomic operations on single variables have features that minimize synchronization and help avoid memory consistency errors.

Lock Objects

Lock objects work very much like the implicit locks used by synchronized code in that only one thread can own a Lock object at a time¹.

The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock (so that livelock, starvation and deadlock are not a problem):

- The `tryLock()` method returns if the lock is not available immediately or before a timeout expires (if specified).
- The `lockInterruptibly()` method returns if another thread sends an interrupt before the lock is acquired.

Lock objects also support a `wait/notify` mechanism, through their associated `Condition` objects and thus replace basic monitor methods (`wait()`, `notify()` and `notifyAll()`) with specific objects:

- A `Lock` in place of synchronized methods and statements.
- A `Condition` in place of `Object`'s monitor methods.

A `Condition` instance is intrinsically bound to a lock.

¹ A thread cannot acquire a lock owned by another thread. But a thread can acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables **Reentrant Synchronization**.

To obtain a Condition instance for a particular Lock instance use its newCondition() method.

Example of Lock and Condition

```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty= lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```