Software Development (CS2500)

Lectures 56: Low-Level Thread Control

M. R. C. van Dongen

March 3, 2014

Contents

1 Outline

Today we continue studying thread synchronisation. We start with the `join( )` command, which waits until a `thread` dies. We continue with *thread notification.* Thread notification involves two ideas. First, it lets threads *wait* until a certain condition is met. Second, it lets other threads create this condition and *notify* waiting threads when the condition is met. The mechanism that lets threads wait involves (1) temporarily suspending thread execution and (2) resuming thread execution when a thread is notified, thereby squeezing the maximum out of CPU performance. We conclude by studying two high-level synchronisation objects called *locks* and *semaphores.* This lecture is mainly based on [Oaks, and Wong 2004, Chapters 1–4] and the `Java` API documentation.

2 The `join( )` Method

The `join( )` method is an instance method defined in the `Thread` class. The call `thread.join( )` blocks and returns when `thread` has died.[1] The method is risky so it requires exception handling. The main purpose of `join( )` is to enforce thread synchronisation. For example, it lets a `main` thread first start a number of threads and then wait until the threads have done their job.

The following is an example that uses `Threads` to carry out some time-expensive computations. The attributes `input` and `result` are for the input and output of the computations.[2]

```java
public class Worker extends Thread {
    private final int input;
    private int result;

    …
}
```

In this example the `main( )` carries out the required computation. The following shows the `main( )`. The main purpose of the `main( )` is to create the `Worker` threads. Each `Worker` is constructed

[1]There's also an overloaded version that lets you wait for a maximum given number of milliseconds.
[2]Notice that we could have implemented this using a single attribute.

by giving it the input number for its computation. The `Workers` are started so they can compute their results concurrently. The `main( )` then uses `join( )` to collect and process the result of each `Worker` computation.

```java
public static void main( String[] args ) {
    final Worker[] results = new Worker[ 1000 ];
    for (int input = 0; input != results.length; input++) {
        results[ input ] = new Worker( input );
        results[ input ].start( );
    }
    try {
        for (Worker worker : results) {
            worker.join( );
            process( worker.result );
        }
    } catch (InterruptedException exception) {
        // ignored
    }
}
```

The following is the constructor of the `Worker` class. Since the `Worker`'s `run( )` method doesn't take any parameters, the input for the `Worker` is stored in an instance attribute. When the `Worker` is `run( )`, the `Worker` can collect the input and initiate the computation. This is done by calling `result = computation( input )`. The implementation of `computation( )` and `process( )` are omitted.

```java
private Worker( final int input ) {
    this.input = input;
}

@Override
public void run( ) {
    result = computation( input );
}
```

For similar reasons, the `Worker` stores the result of the computation in an instance attribute called `result`. That way, the `main` can collect the result when the call `join( )` returns. This can only happen when the each `Worker` has returned from its `run( )`, which is called by `start( )`.

This way of processing should save wall-clock time if the order of processing the results doesn't matter.

3 Notifying `Threads`

So far we've studied a few synchronisation mechanisms.

- Monitors limit the maximum number of threads in object monitors. You can add code to object monitors with the keyword `syncrhonized`.
- The `join( )` method waits until a thread has died.

In the last case the `join( )` call may be considered as a mechanism to *notify* the calling thread about an event: the death of another thread. As we shall see in this section, the `Java` language also provides a built-in mechanism that lets one thread inform other threads about other events. The events are called *conditions*.

To find out more about the built-in notification mechanism we must visit the `Object` class, which defines the following three interesting methods.[3] *All three instance methods can only be called in the object monitor of the object that owns the instance methods.*

`void wait( )` This call blocks the calling `Thread` and puts the `Thread` in the current object's *wait queue*.[4] The calling `Thread` immediately releases monitor ownership. The `Thread` can only resume its computation if some other `Thread` uses the current object to call the method `notify( )` or `notifyAll( )`.

Interrupts and spurious wakeups are possible!

`void notify( )` This call notifies some `Thread` in the current object's wait queue (if any). In the following we shall assume some `Thread` is waiting in the current object's wait queue. The mechanism for letting this thread resume its computation is complicated because the calling thread first has to leave the monitor. The following sequence explains what happens.

1. The JVM selects some waiting `Thread` from the current object's wait queue.
2. The current thread leaves the current object's monitor.
3. The selected `Thread` resumes exection and takes over monitor ownership.

It is important to understand that in Step 2 the calling thread doesn't have to leave the monitor immediately. In fact, the thread may stay in the monitor as long as it likes. However, usually, the thread leaves the monitor immediately because this reduces the time it takes to re-activate the notified thread's computation.

`void notifyAll( )` This method wakes all threads waiting for the condition to occur. Because only one thread may own the current object's monitor, waking the trherads is done one thread at a time.

The following is a typical example.

```java
synchronized(object) {
    while (!condition( )) {
        object.wait( );
    }
    // Perform action appropriate to condition( ).
}
```

It is important you understand the example. First of all the calling thread calls `object.wait( )` in a `while` loop. That way it can keep waiting on the condition in the presence of a spurious wakeup. Second, the `condition( )` is tested in the object monitor belonging to `object`. This is part of the `wait( )-notify( )` protocol.

The following code makes sure a waiting `Thread` is notified about a possible condition. The call is executed in a block that should be synchronized on the same object the waiting `Thread` used to call `wait( )`.

```java
if (condition) {
    synchronized(object) {
        object.notify( );
    }
}
```

[3]For simplicity we won't study the two overloaded versions of the `wait( )` method that are also provided by the `Object` class.

[4]The term *wait queue* is not standard.

4 Locks

A *critical section* is a collection of statements that share one or more common resources. For example, a monitor.

A *lock* is an object that implements a critical section. Locks are implemented as an interface called `Lock`, which is defined in the package `java.util.concurrent.locks`. A lock's critical section is defined implicitly. A thread tentatively enters the critical section by calling the lock's instance method `lock( )`. (The call blocks if the critical section is locked.) A thread leaves the critical section by calling the lock's instance method `unlock( )`.

The following is a typical example. The code for creating the lock is omitted.

```java
// enter critical section.
lock.lock();
try {
    // Use shared resources.
} finally {
    // leave critical section.
    lock.unlock();
}
```

The `Lock` interface also defines a non-blocking version of `lock( )`. The following shows how you use it.

```java
// enter critical section.
if (lock.tryLock()) {
    try {
        // Use shared resources.
    } finally {
        // leave critical section.
        lock.unlock();
    }
}
```

Locks generalise monitors. The following is a short comparison.

- Locks don't need block-structured critical sections. For example, locks can have overlapping critical sections. With two locks, you can lock the second lock in the first lock's critial section and unlock the second lock outside the first lock's critical section.
- Locks may be released in any order. With block-structured monitors a thread can only lock/acquire monitors in one order and unlock/release them in the opposite order.
- Locks have non-blocking lock methods. When a thread attempts to enter a locked monitor, the thread will always be blocked.
- Block-structured monitors are more robust and easier to use. For example, it is impossible to forget releasing a monitor's lock. On the other hand, the flexibility of locks may lead to errors such as forgetting to unlock a lock.

5 Semaphores

5.1 Introduction

Monitors and locks limit the allowed number of threads to 1. A *semaphore* is a generalised lock for a critical section. A semaphore has a *size*—a positive integer. A semaphore with initial size *n* may allow up to n concurrent threads in a critical section. With size 1 we have a critical section with mutual exclusion: a lock.

- A semaphore's critical section is also defined implicitly.
- To enter the critical section a thread must call `p( )`. Other names for `p( )` are `enter( )`, `down( )`, …. Let i be the initial value and let c be the current value of the semaphore's counter. When a thread calls `p( )` there are two posibilities:

  $c > 0$ When this happens' it means the semaphore's critical section is not full. The semaphore's counter is decremented atomically and the thread enters the semaphore's critical section.

  $c = 0$ The semaphore's critical section is full. When this happens, the thread is blocked and its `Thread` is put in the semaphore's wait queue.

  At any stage, the total number of threads in the monitor's critical section is equal to$i - c$.
- You leave the section by calling instance method `v( )`. Other names for `v( )` are `leave( )`, `up( )`, …. When a thread calls the method, this atomically increments the semaphores counter. If the semaphore's wait queue is non-empty, the semaphore will remove a `Thread` from the queue and wake up the `Thread`'s thread. When this happens, the thread enters the semaphore's critical section and the semaphore will atomically decrement its counter.

5.2 Possible Implementation

`Java` has packages that provide semaphores. In the following we implement our own user-defined semaphores. **For simplicity we assume that threads do not recursively acquire a semphore if they've already acquired it.**

The following class defines the user-defined semaphore. The `main( )` creates an instance of the `Semamphore` class. The `Sempahore` instance limits the number of threads that can acquire the instance to 2. The `main( )` then creates some "limited" threads that carry out the work concurrently. The semaphore limits the threads because it allows no more than two threads can acquire the semaphore.

```java
public class Semaphore {
    private int counter;

    public void main( String[] args ) {
        final Semaphore sem = new Semaphore( 2 );
        for (int i = 0; i != 6; i++) {
            final Thread thread = new LimitedThread( sem );
            thread.start( );
        }
    }

    public Semaphore( final int size ) {
        this.counter = size;
    }
    …
}
```

The following shows the `LimitedThread` class. The method `computation( )` simulates a computation.

```java
private static class LimitedThread extends Thread {
    private final Semaphore sem;

    private LimitedThread( final Semaphore sem ) {
        this.sem = sem;
    }

    @Override
    public void run( ) {
        sem.p( );
        computation( );
        sem.v( );
    }
}
```

Notice that the critical section is defined implicitly by the calls to the semaphore's instance methods `p( )` and `v( )`. Here the semaphore is implemented as an instance attribute. (We could also have used a `try-catch` block implementation similar to the one presented in the previous section.) By sharing the semaphore that is passed to the constructor, several `LimitedThread` instances chan share a critical section.

In the current implementation, it was decided to implement `LimitedThread` as an inner class. Before you continue, make sure you understand why this decision forces the class to become `static`.

Remember that `p( )` and `v( )` are semaphore instance methods for entering and leaving the critical section. The following are the missing pieces of the puzzle. In the following we implement the semaphore class with a `wait( )-notify( )` mechanism.

```java
public synchronized void v( ) {
    // counter >= 0
    counter++;
    notify( );
    // counter > 0
}

public synchronized void p( ) {
    // counter >= 0
    try {
        while (counter == 0) {
            wait( );
        }
        // counter > 0
        counter--;
    } catch (Exception exception) {
        // omitted
    }
    // counter >= 0
}
```

It is not difficult to see how it works. The value of the attribute `counter` is the number of threads that can acquire the semaphore without blocking. The `wait( )-notify( )` mechanism requires that the instanc methods `p( )` and `v( )` share the same monitor. This explains why both methods are `synchronized`.

Let's continue with `v( )`. When a thread releases the semaphore this means that one more thread can acquire the semaphore without blocking. Incrementing `counter` and notifying any waiting threads clearly is the correct way to implement this. *Notice the invariants are a great help in understanding how `p( )` and `v( )` work.*

Understanding the method `p( )` is not much more difficult. When `counter > 0`, this means the calling thread can acquire the semaphore without blocking. The `Semaphore` maintains the invariant that `counter` is non-negative. Therefore `counter` must be positive after the `while` loop, which is equivalent to saying the current thread shouldn't block if it gets past the `while` loop. The rest is easy. The current thread must `wait( )` while `counter == 0` because the condition `counter == 0` means any attempt to acquire the semaphore should block. Otherwise, the current thread acquires the semaphore and this means that `counter` must be decremented.

6 Ordering Events

Many applications require events happen in a specific order. For example:

- One thread fills a buffer.
- Another thread reads what's in the buffer.
- The second thread should only start when the first thread is done.

In the remainder of this section we shall study two techniques that let us order two events. We have two `Threads`, `first` and `last`. Using the techniques we make sure `first` carries out its task before `last`.

The first technique uses a semaphore or a lock. The idea is that we use a semaphore `sem` (lock `lock`) and reset its counter to zero (lock the lock `lock`) before we start the threads. We then let `first` increment the counter (unlock `lock`) and let `last` decrement the counter (lock `lock`). Of course, `last`'s attempt to decrement the counter will block until `first` as incremented it. The following is an example; it should always print `first` before printing `last`.

```java
public static void main( String[] args ) {
    final Semaphore sem = new Semaphore( 1 );
    sem.p( ); // decrement sem's counter
    final Thread first = new Thread( ) {
        @Override public void run( ) {
            try {
                System.out.println( "first" );
            } finally {
                sem.v( ); // increment sem's counter
            }
        }
    };
    final Thread last = new Thread( ) {
        @Override public void run( ) {
            sem.p( ); // decrement sem's counter
            System.out.println( "last" );
        }
    };
    last.start( );
    first.start( );
}
```

The second technique uses `Java`'s `wait( )` and `notify( )`. This technique requires we have some variable/attribute that is used to represent the condition that is needed by `last` before it can proceed with its task. In the following example, we use a `boolean` attribute `condition` of some user-defined class.

```java
public class LockExample {
    private boolean wait;

    public static void main( String[] args ) {
        final LockExample lock = new LockExample( );
        // Both threads can see lock.
        // lock.condition == false;

        final Thread first = new Thread( ) {
            @Override public void run( ) {
                // omitted
            }
        };
        final Thread last = new Thread( ) {
            @Override public void run( ) {
                // omitted
            }
        };
        last.start( );
        first.start( );
    }
}
```

```java
final Thread first = new Thread( ) {
    @Override public void run( ) {
        try {
            System.out.println( "first" );
        } finally {
            synchronized(lock) {
                lock.condition = true;
                lock.notify( );
            }
        }
    }
};
```

The second technique uses `Java`'s `wait( )` and `notify( )`. This technique requires we have some variable/attribute that is used to represent the condition that is needed by `last` before it can proceed with its task. In the following example, we use a `boolean` attribute `condition` of some user-defined class.

```java
final Thread last = new Thread( ) {
    @Override public void run( ) {
        synchronized(lock) {
            try {
                while (!lock.condition) {
                    lock.wait( );
                }
            } catch (InterruptedException exception) {
            }
        }
        System.out.println( "last" );
    }
};
```

7 For Friday

Study the lecture notes.

References

Oaks, Scott, and Henry Wong [2004]. *Java Threads.* Third Edition. O'Reilly. ISBN: 0-596-00782-5.

This lecture is based on [Oaks, and Wong 2004, Chapters 1–4] and the `Java api` documentation.