Software Development (CS2500)

Lectures 56: Low-Level Thread Control

M. R. C. van Dongen

March 3, 2014

Outline

- Today we continue studying thread synchronisation.
- We start with `join( )`, which waits until a `thread` dies.
- We continue with *thread notification*.
  - Lets threads `wait( )` until a conditions is met.
  - Lets threads create conditions and `notify( )` waiting threads.
  - Waiting threads do not consume any CPU time.
- Conclude with *locks* and *semaphores*.
- Lecture mainly based on
  - [Oaks, and Wong 2004, Chapters 1–4] and
  - The `Java api` documentation.

The `join( )` Method

- `join( )` is an instance method defined in the `Thread` class.
- `thread.join( )` blocks and returns when `thread` has died.
- Risky method: requires exception handling.
- Main purpose is synchronisation.

Example

Java

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

    ...
}
```

Example (Continued)

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
    }
}
```

Example (Continued)

Java

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

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

Synchronisation Techniques

- So far we've studied a few synchronisation mechanisms.
  - Monitors: no more than one thread per object monitor.
    - Add code to object monitors with the keyword `syncrhonized`.
  - The `join( )` method waits until a thread has died.
- `join( )` notifies current thread about death of other thread.
- Can we generalise this (event) notification mechanism?

The `wait( )`-`notify( )` Mechanism

`object.wait( )`

`void wait( )` Blocks `Thread` and puts it in `object`'s wait queue.

- Calling `Thread` releases monitor ownership.
- *Interrupts and spurious wakeups are possible!*

`void notify( )` Notifies thread in `object`'s wait queue (if any).

1. JVM selects `Thread` from object's wait queue.
2. Current thread leaves object's monitor.
3. Selected `Thread` takes over the monitor.

The `wait( )-notify( )` Mechanism

`object.notify( )`

`void wait( )` Blocks `Thread` and puts it in `object`'s wait queue.

- Calling `Thread` releases monitor ownership.
- *Interrupts and spurious wakeups are possible!*

`void notify( )` Notifies thread in `object`'s wait queue (if any).

1. JVM selects `Thread` from object's wait queue.
2. Current thread leaves object's monitor.
3. Selected `Thread` takes over the monitor.

Example: `wait( )`

Java

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

Notifying a Waiting Thread

Java

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

Semaphores

- *Critical section:* code that shares common resources.
  - For example, a monitor.
- A *lock* is an object that implements a critical section.
- Locks are implemented as an interface: `Lock`.
- A lock's critical section is defined implicitly.
  - A thread tentatively enters the critical section by calling `lock( )`.
  - A thread leaves the critical section by calling `unlock( )`.

Example

Java

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

Non-Blocking Locking

Java

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

Comparison

- Locks don't need block-structured critical sections.
- Locks may be released in any order.
- Locks have non-blocking lock methods.
- Block-structured monitors are more robust and easier to use.

Semaphores

- 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.
  - Size equals maximum allowed threads in critical section.
- With size 1 we have a critical section with mutual exclusion.

Semaphores (Continued)

- 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( )`, ...
  - Calling `p( )` blocks if the critical section is full.
- You leave the section by calling instance method `v( )`.
  - Other names for `v( )` are `leave( )`, `up( )`, ...
  - Calling `v( )` may wake up a blocked thread.

Example

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;
    }
    ...
}
```

Example (Continued)

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( );
    }
}
```

Example (Continued)

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
}
```

Example (Continued)

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
}
```

Example (Continued)

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
}
```

Ordering Events

- Many applications require that events happen in a specific order.
- For example:
  1. One thread fills a buffer.
  2. Another thread reads what's in the buffer.
  3. The second thread should only start when the first thread is done.

First Technique: Semaphore/Locks

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( );
}
```

Second Technique: `wait( )/notify( )`

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( );
    }
}
```

Second Technique: `wait( )`/`notify( )` (Continued)

Java

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

Second Technique: `wait( )`/`notify( )` (Continued)

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" );
    }
};
```

For Wednesday

- Study the lecture notes.

Bibliography


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

Acknowledgements

- Lecture mainly based on
  - [Oaks, and Wong 2004, Chapters 1–4] and
  - The `Java api` documentation.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.