

G52CON: Concepts of Concurrency

Lecture 12 Synchronisation in Java II

Brian Logan
School of Computer Science & IT
bsl@cs.nott.ac.uk

Outline of this lecture

- mutual exclusion in Java revisited
- problems with synchronized
 - example: non-block structured locking
 - example: read-write locks
- condition synchronisation in Java revisited
- problems with `notifyAll()`
 - example: improving the `Bounded Buffer` solution
- exercise 5: Dining Philosophers Problem

Fully synchronised objects

The safest design strategy based on mutual exclusion is to use *fully synchronized objects* in which:

- all methods are synchronized
- there are no public fields or other encapsulation violations
- all methods are finite
- all fields are initialised to a consistent state in constructors
- the state of the object is consistent at both the beginning and end of each method (even in the presence of exceptions).

Problems with synchronized

synchronized methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for synchronisation

One way these problems can be overcome is by using *utility classes* to control locking.

Problems with synchronized

synchronized methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for synchronisation

One way these problems can be overcome is by using *utility classes* to control locking.

Example: GeneralSemaphore

The `GeneralSemaphore` class is fully synchronized:

- when the `P()` method is invoked on an instance of the `GeneralSemaphore` class, `s`, the invoking thread attempts to obtain the lock on `s`
- there is no way to back off if the lock is already held by another thread, to give up after waiting for a specified time, or to cancel the lock attempt following an `interrupt`
- this can make it difficult to recover from liveness problems.

Semaphores in Java

```
class GeneralSemaphore {
    protected long resource;

    public GeneralSemaphore (long r) {
        resource = r;
    }

    public synchronized void V() {
        ++resource;
        notify();
    }
}
```

Semaphores in Java 2 (pre Java 5)

```
public synchronized void P() throws
InterruptedException {
    try {
        while (resource <= 0) {
            wait();
        }
        --resource;
    } catch (InterruptedException ie) {
        notify();
        throw ie;
    }
}
```

Handling interrupts

Threads should periodically check their interrupt status, and if interrupted, shut down:

- a good place to check is before calling a synchronized method, e.g.:

```
// other code ...  
s.P();
```

as we may spend a long time contending for the lock on `s`

- this can result in threads being unresponsive to interrupts

Semaphores in Java 2a (pre Java 5)

```
public void P() throws InterruptedException {  
    if (Thread.interrupted())  
        throw new InterruptedException();  
    synchronized(this) {  
        try {  
            while (resource <= 0)  
                wait();  
            --resource;  
        } catch (InterruptedException ie) {  
            notify();  
            throw ie;  
        }  
    }  
}
```

Semaphores in Java 2b (Java 5)

```
public void P() throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    synchronized(this) {
        while (resource <= 0)
            wait();
        --resource;
    }
}
```

Backing off from lock attempts

Even if we check for interrupts before attempting to acquire a lock on a synchronized method or block

- a thread which is trying to acquire the lock must be prepared to wait indefinitely
- deadlocks are fatal—the only way to recover is to restart the application
- however, we can implement more flexible locking protocols using utility classes

The Lock interface

`java.util.concurrent` defines a `Lock` interface and a number of utility classes (e.g., `ReentrantLock`) which implement the interface:

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit) throws
        InterruptedException;
    void unlock();
    Condition newCondition();
}
```

Replacing synchronized blocks

A `Lock` can be used to replace blocks of the form:

```
synchronized(this) { /* body */ }
```

with a before/after construction, e.g.:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // body: catch & handle exceptions if necessary
} finally {
    lock.unlock();
}
```

Backing off from lock attempts with `Lock`

Unlike `synchronized`, the `Lock` interface supports:

- *polled* lock acquisition: `tryLock()` allows control to be regained if all the required locks can't be acquired
- *timed* lock acquisition: `tryLock(timeout)` allows control to be regained if the time available for an operation runs out
- *interruptible* lock acquisition: `lockInterruptibly` allows an attempt to acquire a lock to be interrupted

Problems with `synchronized`

`synchronized` methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an `interrupt`
- **synchronisation within methods and blocks limits use to strict block structured locking**
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for synchronisation

One way these problems can be overcome is by using *utility classes* to control locking.

Locking is block structured

- `synchronized` methods and blocks limits use to strict block structured locking
- a lock is always released in the same block as it was acquired, regardless of how control exits the block
- e.g., a lock can't be acquired in one method or block and released in another
- this prevents potential coding errors, but can be inflexible
- again we can use utility classes to implement non-block structured locking

Example: `ListUsingLocks`

- for example, we can use `Lock` objects to lock the nodes of linked list during operations that traverse the list
- the lock for the next node must be obtained while the lock for the current node is still being held
- after acquiring the next lock, the current lock is released
- this allows extremely fine-grained locking and increases potential concurrency
- only worthwhile in situations where there is a lot of contention.

Example: ListUsingLocks

```
class ListUsingLocks {
    static class Node {
        Object item;
        Node next;
        Lock lock = new ReentrantLock();

        Node(Object x, Node n) { item = x; next = n; }
    }

    protected Node head;

    protected synchronized Node getHead() { return head; }

    public synchronized add(Object x) {
        head = new Node(x, head);
    }
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

19

ListUsingLocks 2

```
boolean search(Object x) throws InterruptedException {
    Node p = getHead();
    if (x == null || p == null) return false;

    Node nextp;
    p.lock.lock();
    for (;;) {
        try {
            if (x.equals(p.item)) return true;
            if ((nextp = p.next()) == null) return false;
            nextp.lock.lock();
        } finally {
            p.lock.unlock();
        }
        p = nextp;
    }
} // other methods omitted ...
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

20

ListUsingSynchronized

```
// Broken, do not use ...
boolean search(Object x) throws InterruptedException {
    Node p = getHead();
    if (x == null || p == null) return false;

    Node nextp;
    for (;;) {
        synchronized(p) {
            if (x.equals(p.item)) return true;
            if ((nextp = p.next()) == null) return false;
            synchronized(nextp) {
                // can't release the lock on p here ...

            } // lock on nextp will be released here
            p = nextp;
        } // lock on 'p' will be released here ...
    }
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

21

Problems with synchronized

synchronized methods and blocks have a number of limitations:

- there is no way to back off from an attempt to acquire a lock, e.g., to timeout or cancel a lock attempt following an interrupt
- synchronisation within methods and blocks limits use to strict block structured locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for synchronisation

One way these problems can be overcome is by using *utility classes* to control locking.

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

22

Altering the semantics of a lock

With `synchronized` there is no way to alter the semantics of a lock, e.g., read vs write protection

- this makes it difficult to solve *selective mutual exclusion* problems, like the Readers and Writers problem
- again, these problems can be overcome by using *utility classes* to control locking

The `ReadWriteLock` interface

- `java.util.concurrent` defines a `ReadWriteLock` interface and a number of utility classes (e.g., `ReentrantReadWriteLock`) which implement the interface:

```
public interface ReadWriteLock {  
    Lock readLock(); // returns the read lock  
    Lock writeLock(); // returns the write lock  
}
```

ReadWriteLocks

A `ReadWriteLock` maintains a pair of associated `Locks`, one for read-only operations and one for writing:

- the `readLock` may be held simultaneously by multiple reader threads, so long as there are no writers
- the `writeLock` is exclusive
- since the `readLock` and `writeLock` implement the `Lock` interface, they support polled, timed and interruptible locking

Read-Write locks

A read-write lock can allow for a greater level of concurrency in accessing shared data than that permitted by a mutual exclusion lock if:

- the methods in a class can be separated into those that only read internally held data and those that read and write
- reading is not permitted while writing methods are executing
- the application has more readers than writers
- the methods are time consuming, so it pays to introduce more overhead in order to allow concurrency among reader threads.
- e.g, in accessing a dictionary which is frequently read but seldom modified

Example RWDictionary

```
class RWDictionary {
    private final Map<String, Data> m =
        new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    // methods follow ...
}
```

Example RWDictionary readers

```
// Reader method (does not update the map)
public Data get(String key) {
    r.lock();
    try { return m.get(key); }
    finally { r.unlock(); }
}
```

Example RWDictionary writers

```
// Writer method (changes the map)
public Data put(String key, Data value) {
    w.lock();
    try { return m.put(key, value); }
    finally { w.unlock(); }
}
}
```

Mutual exclusion summary

- all the synchronisation primitives we have looked at are equivalent in the sense that they all have the same expressive power
- while it is often helpful to take advantage of the higher level of abstraction offered by monitors, there are situations when other forms of synchronisation are required and we can implement any of these using any of the primitives.
- more complex forms of locking can and are defined in terms of primitives like `Lock`.
- at each level of abstraction we see this pattern of acquiring and releasing locks.

Condition synchronisation in Java

Condition Synchronisation can be implemented using the methods `wait()`, `notify()` and `notifyAll()`:

- to delay a thread until some condition is true, write a loop that causes the thread to `wait()` (block) if the delay condition is false
- ensure that every method which changes the truth value of the delay condition notifies threads waiting on the condition (using `notify()` or `notifyAll()`), causing them to wake up and re-check the delay condition.

Context switching in Java

When a thread blocks and/or another is scheduled, the JVM must perform a *context switch*:

- this involves saving the registers of the suspended thread and loading the registers of the newly scheduled thread
- which takes time
- a concurrent program, runs faster if we can reduce the number of context switches.

Condition variables in Java

In Java, each object has a single implicit condition variable:

- a `notifyAll()` intended to inform threads about one condition also wakes up threads waiting for unrelated conditions, resulting in large numbers of context switches
- context switching can be minimised by delegating operations with different `wait()` conditions to different helper objects
- such helper objects serve as *condition variables*—places to put threads that need to wait and be notified.

Bounded buffer in Java

```
class BoundedBuffer {
    // Private variables ...
    Object[] buf;
    int out = 0, // index of first full slot
    int in = 0, // index of first empty slot
    int count = 0; // number of full slots

    public BoundedBuffer(int n) {
        buf = new Object[n];
    }

    // continued ...
}
```

Bounded buffer in Java 2

```
// Monitor procedures ...
public synchronized void append(Object data) {
    try {
        while(count == n) {
            wait();
        }
        catch (InterruptedException e) {
            return;
        }
        buf[in] = data;
        in = (in + 1) % n;
        count++;
        notifyAll();
    }
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

35

Bounded buffer in Java 3

```
public synchronized Object remove() {
    try {
        while(count == 0) {
            wait();
        }
        catch (InterruptedException e) {
            return null;
        }
        Object item = buf[out];
        out = (out + 1) % n;
        count--;
        notifyAll();
        return item;
    }
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

36

Bounded buffer with semaphores

```
class BoundedBufferWithSemaphores {
    // Private variables ...
    BufferArray buf;
    GeneralSemaphore empty;
    GeneralSemaphore full;

    public BoundedBufferWithSemaphores(int n) {
        buf = new BufferArray(n);
        empty = new GeneralSemaphore(n);
        full = new GeneralSemaphore(0);
    }

    // continued ...
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

37

Bounded buffer with semaphores 2

```
public void append(Object data)
    throws InterruptedException {
    empty.P();
    buff.append(data);
    full.V();
}

public Object remove()
    throws InterruptedException {
    full.P();
    Object data = buff.remove();
    empty.V();
}
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

38

Bounded Buffer with Semaphores 3

```
class BufferArray {
    Object[] array; int in = 0; int out = 0;

    BufferArray(int n) { array = new Object[n]; }

    synchronized void append(Object data) {
        array[in] = data;
        in = (in + 1) % array.length;
    }

    synchronized Object remove() {
        Object data = array[out];
        array[out] = null;
        out = (out + 1) % array.length;
        return data;
    }
}
```

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

39

Quadratic to linear

- `BoundedBufferWithSemaphores` is likely to run more efficiently than the `BoundedBuffer` class when many threads are using the buffer
- it uses two different underlying wait sets
- the semaphores only wake one thread on each operation, eliminating the unnecessary context switching caused by using `notifyAll()` instead of `notify()`
- this reduces the worst case number of wakeups from a quadratic function of the number of invocations to linear

© Brian Logan 2006

G52CON Lecture 12: Synchronisation in Java
II

40

The Condition interface

- Condition factors out the Object condition synchronisation methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object

```
public interface Condition {  
    // Key methods only ...  
    void await() throws InterruptedException  
    void signal()  
    void signalAll()  
}
```

The Condition interface

- because access to the shared condition occurs in different threads, it must be protected by a Lock
- each Condition instance is intrinsically bound to a Lock—to obtain a Condition instance for a particular Lock instance use its newCondition() method.
- waiting for a Condition *atomically* releases the associated lock and suspends the current thread, just like Object.wait()
- supports interruptible, non-interruptible, and timed waits

The class `ArrayBlockingQueue<E>`

- `ArrayBlockingQueue` implements a bounded buffer backed by a fixed-sized array
- attempts to put an element into a full queue block;
- attempts to take an element from an empty queue also block;
- supports an optional *fairness policy* for ordering waiting producer and consumer threads—a queue constructed with *fairness* set to true grants threads access in FIFO order;
- fairness generally decreases throughput but reduces variability and avoids starvation.

Exercise 5: Dining Philosophers Problem

The *Dining Philosophers* problem illustrates mutual exclusion between processes which compete for overlapping sets of shared variables

- five philosophers sit around a circular table
- each philosopher alternately thinks and eats spaghetti from a dish in the middle of the table
- the philosophers can only afford five forks—one fork is placed between each pair of philosophers
- to eat, a philosopher needs to obtain mutually exclusive access to the fork on their left and right

Using any of the concurrency primitives covered so far, devise a solution to the dining philosophers problem for 5 philosophers which avoids *starvation*—e.g., each philosopher acquires one fork and refuses to give it up.

The next lecture

Message Passing

Suggested reading:

- Andrews (2000), chapter 7;
- Burns & Davies (1993), chapter 4;
- Andrews (1991), chapters 7 & 8.