# G52CON: Concepts of Concurrency

## Lecture 12 Synchronisation in Java II

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

## Outline of this lecture

- mutual exclusion in Java revisited
- problems with `synchronized`
  - example: backing off from lock attempts
  - example: non-block structured locking
  - example: read-write locks

## Fully synchronised objects

The safest (but not necessarily the best) 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 (i.e., no infinite loops after acquiring a lock)
- 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).

## Synchronization wrappers

- you can turn a `Collection` into a fully synchronized object using a *synchronization wrapper*
- each of the core collection interfaces, `Collection`, `Set`, `List`, `Map`, `SortedSet`, and `SortedMap`, has a static factory method

```
public static <T> Collection<T>
  synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T>
  synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V>
  synchronizedSortedMap(SortedMap<K,V> m);
```

## Ensuring synchronization

- the factory methods returns a synchronized `Collection` backed by the collection passed as argument
- to ensure mutual exclusion, *all* access to the backing collection must be through the synchronized collection
- simplest approach is not to keep a reference to the backing collection, e.g.:

```
List<T> list = Collections.synchronizedList(new
 ArrayList<T>());
```

- note that wrappers only synchronize interface methods

## Wrappers and iterators

- a synchronized wrapper does not make a collection thread safe for iteration

- iteration involves multiple calls into the collection, and you must manually synchronize on the wrapped collection when iterating over it, e.g.:

```
List<T> list = Collections.synchronizedList(new
 ArrayList<T>());
synchronized(list) {
    for (T e : list)
        foo(e);
}
```

- note that when iterating over a `Collection` view of a synchronized `Map` you should synchronize on the `Map` *not* the `Collection` view

## 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

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

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

## Example: semaphores

A semaphore $s$ is an integer variable which can take only non-negative values. Once it has been given its initial value, the only permissible operations on $s$ are the atomic actions:

$\boldsymbol{P}(s)$ : if $s > 0$ then $s = s - 1$, else suspend execution of the process that called $\boldsymbol{P}(s)$

$\boldsymbol{V}(s)$ : if some process $p$ is suspended by a previous $\boldsymbol{P}(s)$ on this semaphore then resume $p$, else $s = s + 1$

A *general semaphore* can have any non-negative value; a *binary semaphore* is one whose value is always 0 or 1.

## Exercise: semaphores in Java

```
class GeneralSemaphore {
    private long resource;

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


    // method to implement the P operation



    // method to implement the V operation



}
```

## Semaphores in Java

```
// Sample implementation - not the way it’s done in
// java.util.concurrent.Semaphore
class GeneralSemaphore {
    private long resource;

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

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

## Semaphores in Java 2

```
    public synchronized void P() throws
       InterruptedException {
         while (resource <= 0)
               wait();
         --resource;
    }
}
```

## 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 2a

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

## 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

## 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 still be prepared to wait indefinitely
- deadlocks are fatal—the only way to recover is to restart the application (next lecture)
- 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();
    Condtion newCondition(); // next lecture ...
}
```

## 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 `synchronzied`, 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 aquire 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

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 stuctured 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);
    }
```

# 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 ...
```

## 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 ...
    }
}
```

## 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

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

## 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 is 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
}
```

## `ReadWriteLock`s

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.

## The Next Lecture

*Synchronisation in Java III*

Suggested reading:

- Lea (2000), chapter 3.