G52CON:
Concepts of Concurrency

Lecture 13 Synchronisation in Java III

Brian Logan

School of Computer Science

bsl@cs.nott.ac.uk

Outline of this lecture

- condition synchronisation in Java revisited
- problems with `notifyAll()`
  - example: improving the `BoundedBuffer` solution
- deadlock revisited
- resource ordering
- acquiring multiple locks in Java
  - example: swapping values in synchronized objects

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).

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.

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 on and be notified of a particular condition

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();
}
```

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;
}
}
```

Problems with the `BoundedBuffer`

- there are two different conditions a thread may be waiting on:
  - the buffer being not full (producer threads)
  - the buffer being not empty (consumer threads)
- but a `BoundedBuffer` object has only one wait set, so we must use `notifyAll()`
- e.g., if the buffer is full, a consumer taking one item will wake all waiting producers, even though only one can proceed

Bounded buffer with semaphores

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

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

    // continued ...
```

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();
    }
}
```

Bounded buffer with semaphores 3

- operations with different `wait()` conditions are delegated to different *helper objects* – the `GeneralSemaphores`
- underlying array operations are isolated in a simple `BufferArray` class
- `BufferArray` uses `synchronized` methods to ensure mutually exclusive access to the underlying array
- only one thread can access the buffer at a time

Bounded Buffer with Semaphores 4

```
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;
    }
}
```

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

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 utility classes `ReentrantLock`, `ReentrantReadWriteLock`, etc. implement the `Condition` interface

Condition methods

- `void await()`: causes the invoking thread to wait until it is signalled or interrupted
- `boolean await(long time, TimeUnit unit)`: causes the invoking thread to wait until it is signalled or interrupted, or the specified waiting time elapses
- `void awaitUninterruptibly()`: causes the invoking thread to wait until it is signalled
- `void signal()`: wakes one thread waiting on the condition
- `void signalAll()`: wakes all threads waiting on the condition

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

Example: Bounded buffer with Conditions

```
class BoundedBuffer {

    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    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 with Conditions 2

```
public void append(Object data)
        throws InterruptedException {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        buf[in] = data;
        in = (in + 1) % n;
        count++;
        notEmpty.signal();
    } finally {
            lock.unlock();
    }
}
```

Bounded buffer with Conditions 3

```
public Object remove() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object item = buf[out];
        out = (out + 1) % n;
        count--;
        notFull.signal();
        return item;
    } finally {
        lock.unlock();
    }
}
```

Other `java.util.concurrent` utility classes

- utility classes implementing the `Lock` and `Condition` interfaces are used internally in the implementation of other classes in `java.util.concurrent`

- e.g., the `BlockingQueue<E>` interface defines a `Queue` with operations that:

  – wait for the queue to become non-empty when retrieving an element

  – wait for space to become available in the queue when storing an element

- if inserting, removing or examining an element can't proceed, methods may either: throw an exception, return a special value, block, or timeout

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.

Condition synchronisation summary

- simple condition synchronisation can be implemented using the methods `wait()`, `notify()` and `notifyAll()`
- however a single wait set can be inefficient if threads wait on different conditions
- context switching can be minimised by delegating operations with different `wait()` conditions to different *helper objects*
- `Condition` interface makes it clear that an object is being used as a condition variable, and allows interruptible and timed waits

Multiple locks

- utility classes implementing `Lock` and `Condition` allow finer-grained locking
- used correctly, this can increase concurrency/reduce latency
- however problems can arise when threads must acquire *multiple locks*
- a particular problem is *deadlock*
- e.g.: two threads must acquire locks on two objects, get one lock each, and wait forever for each other to release the other lock.

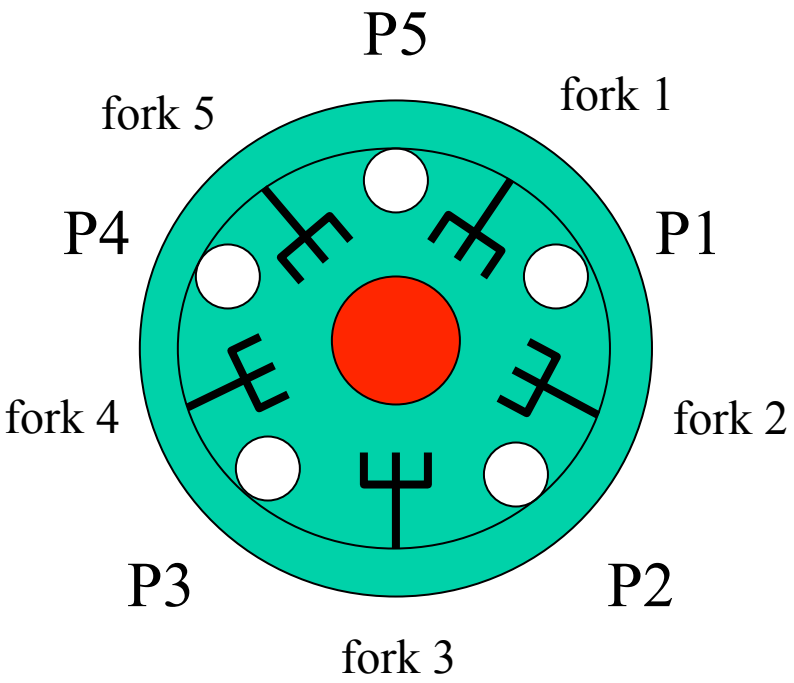
Dining Philosophers Problem

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

The problem is to avoid *starvation*–e.g., each philosopher acquires one fork and refuses to give it up.

Dining Philosophers Problem



Deadlock in the Dining Philosophers

The key to the solution is to avoid *deadlock* caused by circular waiting:

- process 1 is waiting for a resource (fork) held by process 2
- process 2 is waiting for a resource held by process 3
- process 3 is waiting for a resource held by process 4
- process 4 is waiting for a resource held by process 5
- process 5 is waiting for a resource held by process 1.

No process can make progress and all processes remain deadlocked.

Semaphore Solution

```
// Philosopher i, i == 1-4
while(true) {
    //get right fork then left
    P(fork[i]);
    P(fork[i+1]);
    // eat ...
    V(fork[i]);
    V(fork[i+1]);
    // think ...
}
```

```
// Philosopher 5
while(true) {
    //get left fork then right
    P(fork[1]);
    P(fork[5]);
    // eat ...
    V(fork[1]);
    V(fork[5]);
    // think ...
}
```

```
// Shared variables
binary semaphore fork[5] = {1, 1, 1, 1, 1};
```

Deadlock

Although fully synchronised objects are always safe, threads using them are not always live

- some `synchronized` actions are multiparty – they acquire locks on multiple objects

- *deadlock* is possible when two or more objects are mutually accessible from two or more threads, and each thread holds one lock while trying to obtain another lock held by another thread

Example: `Cell`

```
class Cell { // Broken, do not use ...
    private long value;

    synchronized long getValue() { return value; }

    synchronized void setValue(long v) { value = v; }

    synchronized void swapValue(Cell other) {
        long t = getValue();
        long v = other.getValue();
        setValue(v);
        other.setValue(t);
    }
}
```

– see Lea (2000), p 87.

Example deadlock trace

Consider two threads, one of which invokes `a.swapValue(b)` while the other invokes `b.swapValue(a)`

Thread 1	Thread 2
acquire lock for `a` on invoking `a.swapValue(b)`	
pass the lock for `a` (since already held) on invoking `t = getValue()`	acquire lock for `b` on invoking `b.swapValue(a)`
block waiting for lock on `b` on invoking `v = other.getValue()`	pass lock for `b` (since already held) on invoking `t = getValue()`
	block waiting for lock on `a` on invoking `v = other.getValue()`

Resource ordering

One way to avoid this kind of deadlock is to use *resource ordering*:

- associate a numerical (or any other strictly orderable data type) *tag* with each object that can be an argument to a `synchronized` multiparty action

- if synchronization is always performed in *tag order*, then a situation can never arise in which a thread which has a lock on object `x` and is waiting for a lock on `y` while another thread has a lock on `y` and is waiting for a lock on `x`

- whichever thread locks the resource with the lowest tag first will acquire both locks while the other waits, and then the second thread will acquire both locks

Example: resource ordering

- in the case of `Cells`, we can either extend the `Cell` class to add a tag field
- or we can use some existing information about `Cell` objects, e.g., their hash codes
- e.g., we can use `System.identityHashCode` which returns the hash value computed by `Object.hashCode` (even if a class overrides the `hashCode` method)
- while the `identityHashCode` value is not guaranteed to be unique, it is very likely to be unique, which is often good enough

Example: swapValue()

```
public void swapValue(Cell other) {
    if (System.identityHashCode(this) <
        System.identityHashCode(other))
        this.doSwapValue(other);
    else
        other.doSwapValue(this);
}
protected synchronized void doSwapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
}
```

The next lecture

Remote invocation

Suggested reading:

- Andrews (2000), chapter 8;
- Ben-Ari (1982), chapter 6;
- Burns & Davies (1993), chapter 5;
- Andrews (1991), chapters 9.