G52CON:
Concepts of Concurrency

Lecture 12 Synchronisation in Java III

Natasha Alechina

School of Computer Science

nza@cs.nott.ac.uk

Outline of this lecture

- Condition synchronisation in Java
- Multiple shared variables, deadlocks and lock ordering

Summary of the previous lecture

Using `synchronized` makes it easy to write thread-safe Java programs but has 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: e.g. cannot do hand-over-hand locking
- there is no way to alter the semantics of a lock, e.g., read vs write protection
- there is no access control for condition synchronisation

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

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();
}
```

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;
    }
}
```

Bounded buffer with semaphores

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

    public BoundedBufferWithSemaphores(int n) {
        buf = new BufferArray(n);
        empty = new Semaphore(n);
        full = new Semaphore(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

```
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 `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` in `java.util.concurrent` 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 in Java summary

- Before Java 1.5, each object had only one wait set
- Now `java.util.concurrent` has `Condition` interface which allows us to create multiple wait sets per object
- This enables us to write more efficient concurrent programs (minimising context switching)

Overlapping sets of shared variables

• Now about the problems where you have to access a set of variables/acquire a number of locks…

• A potential hazard here is a deadlock: when for example two processes want to acquire locks on two objects, get one lock each, and wait forever for each other to release the other lock.

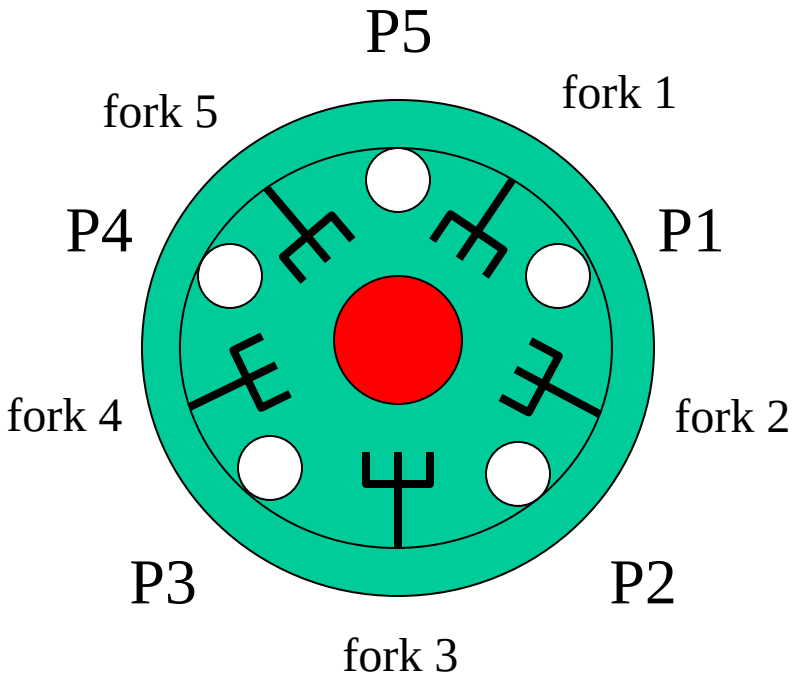
Dining Philosophers Problem

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 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: 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

Message Passing

Suggested reading:

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