Concurrent Programming (Part II)
Lecture 6: Semaphores, Nested Monitors and Deadlock

Dr Kevin Bryson
K.Bryson@cs.ucl.ac.uk
Room 8.04

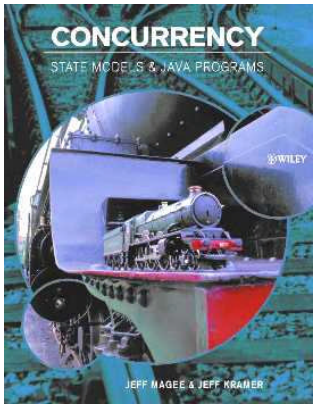
Course Web Site on Moodle
http://moodle.ucl.ac.uk/course/view.php?id=753
Enrolment Key: ATOMIC

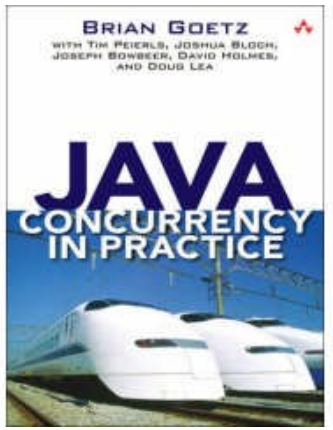
Overview of Lecture

- We will introduce the Semaphore – this is a key concurrency abstractions within a number of systems, for example:
  - Within Unix/POSIX (do "`man sem_init`").
  - Within Berkley SystemV Unix (do "`man semop`")
  - Within PERL (do "`man Thread::Semaphore`")
  - … and now within Java (java.util.concurrent.Semaphore)
- We will examine how we can create a Java Semaphore using conditional synchronization.
- We will then reimplement the BoundedBuffer using our Java Semaphore.
- … which will lead us into lots of problems !

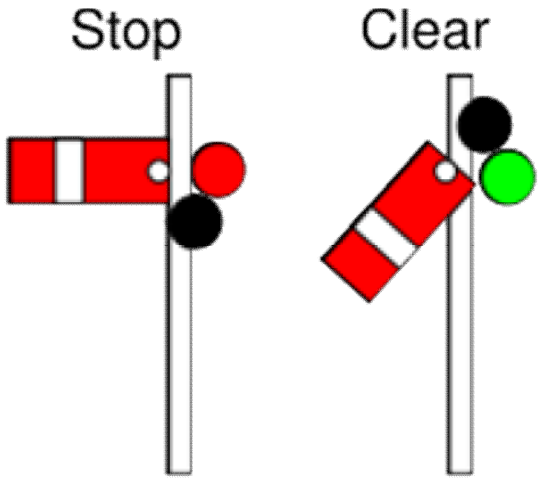
Semaphores – you might have noticed a train theme on concurrency books …

- Semaphores were introduced by Dijkstra in 1968 as one of the first mechanisms for inter-process synchronization (in the humbly named “THE multiprogramming system”)
- The term comes from the railroad – a railway semaphore is a signal flag stopping multiple trains being on the same track.





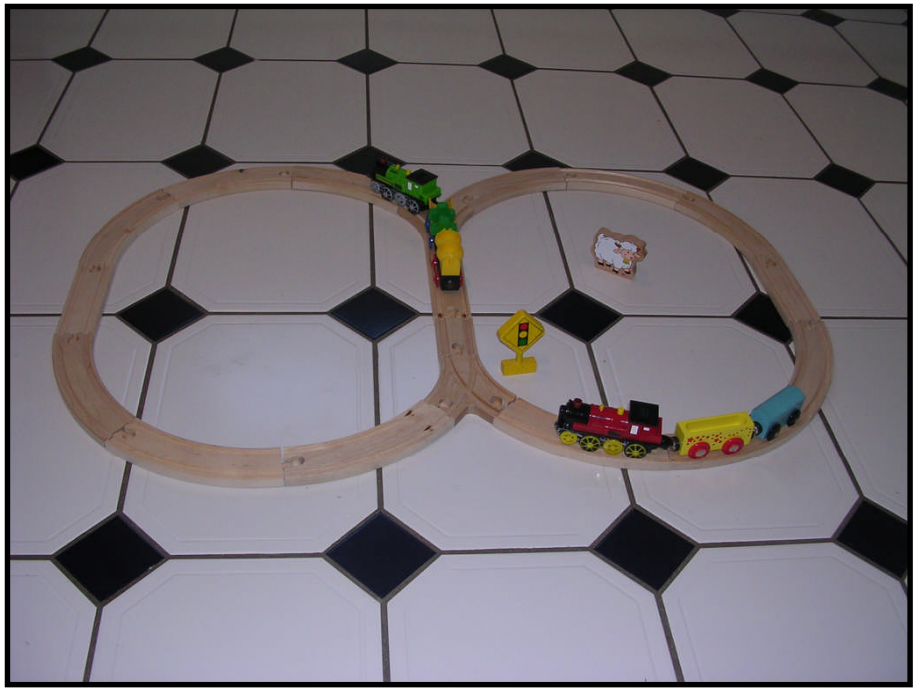
Railway Semaphores

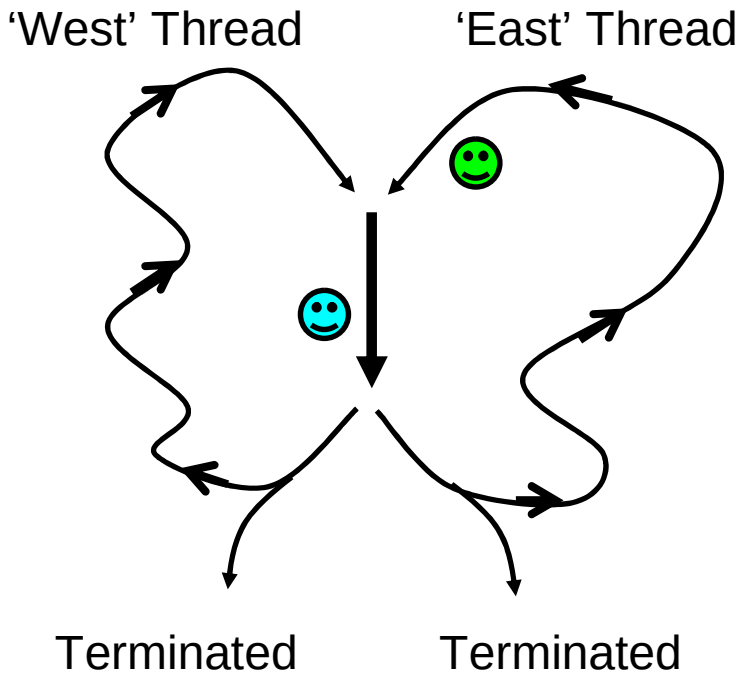


- Trains are real-world objects with concurrent behaviour (**true parallelism** to be precise …)
- Semaphores are used to control the concurrent behaviour of trains by getting them to wait at particular points (thus avoiding multiple trains being on ***critical sections of track***)

Transferring the railway concept to CS …

See the analogy … semaphores … critical sections of code …





Semaphores – Definition

Semaphores are widely used for dealing with inter-process synchronization in operating systems. Semaphore ***s*** is an integer variable that can take only non-negative values.

The only operations permitted on ***s*** are *up(s)* and *down(s)*.

***down(s)*: if** *s* >0 **then**
decrement *s*
else
block execution of the calling process

***up(s)*:** **if** processes blocked on *s* **then**
awaken one of them
else
increment *s*

Semaphores – Implementation

Semaphores are passive objects, therefore implemented as **monitors**.

(In practice, semaphores are a low-level mechanism often used in implementing the higher-level monitor construct.)

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notify();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```

Or we could just use the Java 5 implementation (`java.util.concurrent.Semaphore`)

```
public class Semaphore {
   public Semaphore(long permits);
   public Semaphore(long permits, boolean fair);
   public void acquire() throws InterruptedException;
   public void acquire(long permits) throws InterruptedException;
...
   public boolean tryAcquire();
   public boolean tryAcquire(long timeout, TimeUnit unit);
...
   public release();
   public release(long permits);
}
```

Using Semaphores

- A semaphore can be used to restrict access to a shared resource.
- The count is set to the number of threads that are permitted access simultaneously.

```
class ResourceHandler {
   ...

   Semaphore access = new Semaphore(10);

   public void accessResource() {

       try {
             access.down();
             // Use resource ...
       } finally {
             access.up();
       }
}
```

Binary Semaphores

- The previous Semaphores are also known as ‘Counting Semaphores’
- This distinguishes them from ‘Binary Semaphores’ which are initialized with a count of 1.
- Binary Semaphores behave like mutex (mutually exclusive) locks but **are not re-entrant (unlike Java locks which are).**

```
class CriticalCode {
  ...
  // Binary Semaphore with a value of 1.
  Semaphore access = new Semaphore(1);

  public void doSomethingWithSharedData() {

      try {
            access.down();
            // Critical section of code.
      } finally {
            access.up();
      }
}
```