COMP2207
Managing Concurrency

Professor Timothy Norman

Department of Electronics and Computer Science, University of Southampton
T.J.Norman@soton.ac.uk

Academic Year 2017/18
Handling Concurrent Requests

- We may have many client requests to deal with at the same time.
- How does RMI handle this?
Handling Concurrent Requests

- We may have many client requests to deal with at the same time.
- How does RMI handle this?

- A new thread handles each request...
- ...so, our remote object may be handing multiple calls on multiple/the same methods at the same time.
Managing Concurrency

- We need to avoid race conditions
  - Two or more threads
  - Trying to manipulate a shared resource
  - At the same time

- Threads are racing against each other to win access to the resource

- The result of the computation depends on the schedule of these threads over time

- Different schedules produce different results

- Typical occurrence:
  - Check/read some state;
  - Depending on result of check/read, do a write
Contrived Example

```java
public class Incrementer {
    private int i = 0;
    private int j = 0;
    public Incrementer() {}
    public void increment( int ms ) {
        i++;
        pause( ms );
        j++;
        System.out.println( ((i==j) ? "Same" : "Different") );
    }
    private void pause( int p ) {
        try {
            Thread.currentThread().sleep( p );
        } catch( Exception e ) {}    
    }
}
```

What happens to the values of `i` and `j`?
Contrived Example Starter

```java
public class ThreadStarter extends Thread {
    // Incrementer is a shared resource (static)
    private static Incrementer inc = new Incrementer();
    public ThreadStarter() {}
    public void run() {
        inc.increment(1000);
    }
    public static void main(String[] args) {
        ThreadStarter t1 = new ThreadStarter();
        ThreadStarter t2 = new ThreadStarter();
        t1.start();
        t2.start();
    }
}
```

- Try this out
- Sometimes \(i\) and \(j\) are reported to be different, sometimes the same
Atomicity and Mutual Exclusion

- **Aim:** safeguard the performance of linked actions by a single thread
- **Atomicity**
  - Only one thread is permitted to execute a critical section of code in its entirety
- **Guaranteeing mutual exclusion**
  - Serialisation of access: threads wait for exclusive lock
- In Java, we have an implementation of C. A. R. Hoare’s monitor concept
- A monitor is a barrier securing a shared resource with a lock
- If the resource is not being used by another thread, a thread can acquire the lock
- Other threads wait in a queue until the lock is released
Monitor in Java

- In Java, monitors are generated on the basis of the `synchronized` keyword.
- This keyword declares critical sections of code:
  - A synchronized method
  - A synchronized block
  - A static synchronized block
- As a thread enters a critical section indicated by `synchronized` it needs to acquire a lock.
Monitor in Java

- In Java, monitors are generated on the basis of the `synchronized` keyword.
- This keyword declares critical sections of code:
  - A synchronized method
  - A synchronized block
  - A static synchronized block
- As a thread enters a critical section indicated by `synchronized` it needs to acquire a lock on the entire object.
- So, how can we declare a monitor for our contrived example?
Option 1: Synchronised Block

A thread executing the block of code calling `increment()` requires a lock on the `Incrementer` instance, `inc`

```java
public class ThreadStarter extends Thread {
    // Incrementer is a shared resource (static)
    private static Incrementer inc = new Incrementer();
    public ThreadStarter() {}
    public void run() {
        synchronized( inc ) {
            inc.increment( 1000 );
        }
    }
    public static void main( String[] args ) {
        ThreadStarter t1 = new ThreadStarter();
        ThreadStarter t2 = new ThreadStarter();
        t1.start();
        t2.start();
    }
}
```
Option 2: Synchronised Method

- The `increment()` method is declared as `synchronized`, and so threads calling this method require a lock on the instance.

```java
public class Incrementer {
    private int i = 0;
    private int j = 0;
    public Incrementer() {};
    public void synchronized increment( int ms ) {
        i++;
        pause( ms );
        j++;
        System.out.println( ((i==j) ? "Same" : "Different") );
    }
    private void pause( int p ) {
        try {
            Thread.currentThread().sleep( p );
        } catch( Exception e ) {};
    }
}
```
The Observer Pattern

- Java implements the Observer Pattern in `java.lang.Object`
  - `wait()`
  - `notify()`
  - `notifyAll()`
- These support efficient transfer of control among threads competing to enter a monitor
- Suppose we have a space-limited buffer with
  - Multiple producer threads
  - One or more consumer threads
- Suppose the buffer has a Boolean member variable `full` that is `true` if there’s something in the buffer
- We may have code like this in a `get()` method:

```java
while( !full ) {
  try {
    wait();
  } catch( InterruptedException e ) {} 
}
// Process the buffer
full = false;
notifyAll();
```
A Distributed Systems Example

- We want a remote object that provides a simple register/lookup service.
- In other words, for some (unknown) reason, we want to provide the same functionality provided by the rmiregistry!

```java
public interface RegInterface extends Remote {
    public void bind(String name, Remote obj)
        throws RemoteException, AlreadyRegisteredException;
    public void rebind(String name, Remote obj)
        throws RemoteException;
    public void unbind(String name)
        throws RemoteException;
    public Remote lookup(String name)
        throws RemoteException, NoSuchServiceException;
}
```

- These other exceptions have been defined by me; for example:

```java
public class NoSuchServiceException extends RemoteException {
    public NoSuchServiceException(String msg) {
        super(msg);
    }
}
```
An Unsafe Implementation

- Why is this unsafe?

```java
public class RegImpl implements RegInterface {
    private Map<String, Remote> _map = new HashMap<String, Remote>();

    public void bind(String name, Remote obj) {
        if (_map.containsKey(name)) {
            throw new AlreadyRegisteredException("No!");
        }
        _map.put(name, obj);
    }

    public void rebind(String name, Remote obj) {
        _map.put(name, obj);
    }

    public void unbind(String name) {
        _map.remove(name);
    }

    public Object lookup(String name) {
        if (_map.containsKey(name))
            return _map.get(name);
        throw new NoSuchServiceException("Not found");
    }
}
```
Protect the Critical Sections

- **HashMap is not synchronised** (check the API!)
- Protect the **check-then-act** sections in `bind()` and `lookup()`

```java
public class RegImpl implements RegInterface {
    private Map<String, Remote> _map = Collections.synchronizedMap(
        new HashMap<String, Remote>());
    public synchronized void bind(String name, Remote obj) {
        if (_map.containsKey(name)) {
            throw new AlreadyRegisteredException("No!");
        }
        _map.put(name, obj);
    }
    public void rebind(String name, Remote obj) {
        _map.put(name, obj);
    }
    public void unbind(String name) {
        _map.remove(name);
    }
    public synchronized Object lookup(String name) {
        if (_map.containsKey(name))
            return _map.get(name);
        throw new NoSuchServiceException("NotFound");
    }
}
```
Summary

- RMI generates a new thread for each client request
- We need to avoid race conditions
- Java provides monitors and supports the observer pattern
- Declare need for monitor via synchronized keyword
- Think about what you need to protect
  - check-then-act sections
  - Shared objects (e.g. check API for thread safe collections)