Concurrent Systems and Applications

CST Part 1B, Michaelmas 2002

Examples class
solution notes

tim.harris@cl.cam.ac.uk

Concurrent Systems and Applicati

For each of these problems neither option (*i*) nor option (*ii*) is a universally 'right' answer and so solutions should identify a number of pros and cons of each.

(*a*) The first solution suggests adding a field, say `public int x`, to the class. The `public` modifier makes it accessible from other classes. The second solution suggests using methods for external access and using the `private` modifier to limit direct access to within the defining class:

```
private int x;

public int getX() { return x; }

public void setX(int x) { this.x = x; }
```

The first solution is evidently simpler to write. It may run faster too. The second solution, using encapsulation, can aid maintainability – e.g. the code in the `getX` and `setX` methods can be updated if the data is to be held in some other form. The second solution would have to be used if the data is to be accessed remotely through RMI.

(*b*) Suppose that the existing code is in a method `m` defined on `I1` and that its counterpart on `I2` is `n`. The first solution proposes:

```
class D extends C implements I2 {
  public void n() {
    <code before>
    m();
    <code after>
  }
}
```

The second proposes:

```
class Adapter implements I2 {
  I1 ref;

  public void n() {
    <code before>
    ref.m();
    <code after>
  }
}
```

As before the first solution may be simpler to write. The class `D` supports both

the methods of `I1` and of `I2` – a problem if they have any method signatures in common that should be implemented differently for each interface. The first solution only solves the problem for this one class `C` whereas the second can be used to adapt any class implementing `I1` to the interface `I2`.

(*c*) This is a problem that arose when the standard utility classes, such as `java.util.Hashtable`, were being designed. The first solution (the one chosen there) just adds the synchronized modifier to each method, for example:

```
public void synchronized insert (Object k, Object v)
```

This often performs poorly: acquiring and releasing these locks has some cost in the case of single threaded applications and simple mutual exclusion prevents even read operations from proceeding concurrently. In any case, if the data structure forms part of a larger system then that system may provide its own concurrency control (for example by only invoking operations on the hashtable after acquiring some other lock).

The second solution addresses those problems by allowing concurrency control to be managed on a per-application basis. However, the programmer using the data structure has to be aware of this for correct operation.

(*d*) Syntactically, the only difference between the two solutions is the name of the method. In each case it would be of the form

```
public void finalize () {
  <do close operations>
}
```

The body of the method may have to interact with the server, or simply invoke `close` on the TCP socket.

The two solutions differ in when this method will come to be called. In the first case it must be invoked explicitly by the application: the application must be aware of when it has finished using the connection and it is ready to be closed. This may require extra book-keeping, for example if it is being accessed by several threads. In the second case the method will be invoked automatically once the garbage collector has determined that the object is otherwise no longer accessible to the application. This automated scheme avoids the application having to track when the connection can be closed and removes any risk of calling the method too early (i.e. while the connection is still in use). However, while potentially simpler to the programmer, the second scheme could be overly pessimistic – there is no guarantee of exactly when the finalizer will be called and so it may be delayed some time from when the object ceases to be reachable (which may itself be delayed from when the application will no longer use it).

Concurrent Systems and Applicati

(*a*) The `suspend()` method causes the target thread to pause execution immediately. It may do when while it is holding a lock – either one acquired explicitly by the application or one used internally in the implementation of the JVM. Either case can result in deadlock. 'Lost wake up' problems can also arise if a thread invokes suspend on itself after determining that it should not proceed into a critical section at that time.

(*b*) See example file `Barrier.java`. Points to note:

(*i*) At all times the simple idiom of a while loop calling `wait()` has been used along with `notifyAll()` to wake waiting threads.

(*ii*) At the cost of some extra programming `notify()` could be used instead – note the correspondence between the calls to `notifyAll()` and the threads that those calls wake.

(*iii*) `InterruptedException` is propagated since there is no clear way to deal with errors here.

(*iv*) The two methods are entirely symmetric, as you would expect. Each has two sections: during the first the thread competes with those of the same kind, essentially picking which will be paired up next, and then during the second it waits for a partner.

(*v*) This structure means that there is no need for an explicit shared structure to hold the `id` values – they are passed between the two threads concerned in the `a` and `b` fields.

(*c*) See example files `Shop.java`, `Cutomer.java` and `Barber.java`. This is a 'classic' concurrency problem, usually attributed to Dijkstra. In a computing context the barbers represent devices performing operations on behalf of clients (the customers). During service there's a 1-1 association between clients and devices. After service the device must wait for the client to retrieve results (leave the chair) before moving to another client. Points to note:

(*i*) The `Barrier` class is used to pair up customers with barbers.

(*ii*) The shop, as defined here, serves only to identify the barrier to use – it does not need to identify the barbers or customers using it.

(*iii*) The `haircutFinished` and `customerLeft` fields are protected by the mutual-exclusion lock on the instance of barber – they essentially denote when the barber and the customer have respectively finished using the chair.

(*iv*) It is important to be clear on how the `notify` methods are used in conjunction with mutual-exclusion locks – that is, that the `synchronized` regions here in `Customer` acquire the lock on the associated `Barber` object

```
public class Barrier {
    boolean waitingA = false;
    boolean waitingB = false;
    Object a, b;

    public synchronized Object enterA (Object id)
        throws InterruptedException
    {
        while (waitingA) {
            wait();
        }
        waitingA = true;
        a = id;
        notifyAll();
        while (!waitingB) {
            wait();
        }
        waitingA = false;
        notifyAll();
        return b;
    }

    public synchronized Object enterB (Object id)
        throws InterruptedException
    {
        while (waitingB) {
            wait ();
        }
        waitingB = true;
        b = id;
        notifyAll();
        while (!waitingA) {
            wait ();
        }
        waitingB = false;
        notifyAll();
        return a;
    }
}
```

Paper 5 Question 4. 2002
Concurrent Systems & Applications
part (b)

```
public class Barber {
    boolean haircutFinished;
    boolean customerLeft;

    public Customer getCustomer (Shop s)
        throws InterruptedException
    {
        Customer c;
        haircutFinished = false;
        customerLeft = false;
        c = (Customer) (s.w.enterB (this));
        return c;
    }

    public synchronized void finishedCustomer (Customer c)
        throws InterruptedException
    {
        haircutFinished = true;
        notifyAll();
        while (!customerLeft) {
            wait();
        }
    }
}
```



```
public class Shop {
    Barrier w = new Barrier ();
}
```



```
public class Customer {
    public Barber getHaircut (Shop s)
        throws InterruptedException
    {
        Barber b = (Barber) (s.w.enterA (this));
        synchronized (b) {
            while (!b.haircutFinished) {
                b.wait();
            }
        }
        return b;
    }

    public void leaveChair (Barber b)
        throws InterruptedException
    {
        synchronized (b) {
            b.customerLeft = true;
            b.notifyAll ();
        }
    }
}
```

Paper 5 Question 4 2002

- - - - - Systems & Applications

Concurrent Systems and Applications

2002 Paper 6 Q4

(a) Strict isolation requires that transactions are isolated during their execution from the concurrent effects of others. Non-strict isolation relaxes this during execution, but requires that a transaction executed as-if isolated if it commits.

(b) The transaction proceeds by acquiring locks and performing operations as the locks permit. It releases all of its locks at the point at which it commits or aborts. This enforces strict isolation because retaining locks in this way prevents other transactions from seeing updates made by transactions that are still in progress.

(c) 2PL splits transactions into two phases; a first phase of acquiring locks and a second phase during which locks may be released. As before, operations may be performed at any time that the locks permit. This introduces cascading aborts because updates made by one transaction may be seen by others before they have been committed.

(i) This means that locks may be released during the second phase of execution, possibly increasing concurrency. (ii) When a transaction attempts to commit it must wait until any transactions it saw updates from have committed. (iii) When a transaction aborts it must cause any other transactions that saw updates from it to abort.

(d) (i) Deadlock may have occured. Neither 2PL nor S-2PL prevents deadlock. The usual solutions are possible: coalesce locks, enforce an ordering between lock acquisition, or detect and abort the deadlocked transactions.

(ii) All of the locks must be held until the transaction attempts to commit. 2PL may be better since only the lock protecting the final update would have to be held.

(iii) Use timestamp ordering or optimistic concurrency control. (For 4 marks include a description of one of these)