SOLUTION NOTES

Concurrent Systems and Applications 2002 Paper 5 Question 4 (TLH)

(*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`, `Customer.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

---

`Barrier.java`:

```
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;
    }
}
```

Shop.java

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

Customer.java

```
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 ();
}
    }
}
```

Barber.java

```
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();
}
    }
}
```