

# Mandatory Exercise 2

ITU, IMDD F2005

March 2, 2005

This mandatory exercise is to be handed in March 16, 2004, in the pigeonhole of Jens Chr. Godskesen located at 4C.

The questions in this mandatory exercise are about

- Monitors
- Deadlocks and Safety Properties
- Liveness Properties

## Question 1

This question is about *deadlock*, *liveness* and *safety* properties.

Below is a model of a jobshop where Alic and Bob are supposed to be working under mutual exclusion since they both need the hammer and the mallet of the job shop.

```
Alice = 'getMallet.'getHammer.'workAlice.'putHammer.'putMallet.Alice
Bob = 'getHammer.'getMallet.'workBob.'putHammer.'putMallet.Bob
```

```
Hammer = getHammer.putHammer.Hammer
Mallet = getMallet.putMallet.Mallet
```

```
UnresJobShop = Alice | Bob | Hammer | Mallet
JobShop = (^ getHammer, putHammer, getMallet, putMallet)UnresJobShop
```

Figure 1: CCS model of a simple JobShop with two asymmetric jobbers.

**Question 1.1** Show and explain a minimal trace to a deadlock for the model JobShop in Figure 1. Argue why there is a deadlock.

**Question 1.2** Explain how a deadlock free model can be obtained from the model in Figure 1.

**Question 1.3** Give a safety property in HML that is satisfied by your revised deadlock free model. Argue why it is satisfied. Is the property satisfied by the model in Figure 1

**Question 1.4** Give a liveness property in HML that is satisfied by your revised deadlocked free model. Give a strong liveness property in HML that is satisfied by your revised deadlocked free model. Argue why the properties are satisfied.

## Question 2

This problem is about *monitors*, *deadlock*, *starvation*, and *fairness*.

```
public class BarberShop {  
  
    public static void main(String[] args) {  
        Shop shop = new Shop();  
        new Barber(shop).start();  
        for (int i = 0; i < 5; i++)  
            new Customer(shop).start();  
    }  
}
```

Figure 2: class BarberShop.

A small town contains a small *barbershop*. It's so small that only the *barber* and one out of five *customers* can be in the shop at a time. The implementation of the barbershop (the class BarberShop) is described in Figure 2. The classes Barber, Customer, and Shop will be described below.

```
class Barber extends Thread {  
    Shop shop;  
  
    Barber(Shop shop) {this.shop = shop;}  
  
    public void run() {  
        try {  
            while(true) {  
                shop.get_customer();  
                Thread.sleep(600000);  
                shop.finished_cut();  
            }  
        } catch (InterruptedException e){}  
    }  
}
```

Figure 3: class Barber.

The barber spends his life serving customers. His life cycle is to wait for the next customer to arrive and to give him a haircut. A haircut takes 10 minutes. The imple-

mentation of the barber (the thread Barber) is presented in Figure 3. Notice, that a haircut session is ended by a call to `shop.finished_cut()`.

```
class Customer extends Thread {
    Shop shop;

    Customer(Shop shop) {this.shop = shop;}

    public void run() {
        try {
            while(true) {
                Thread.sleep(900000);
                if (Math.random() < 0.5) shop.get_haircut();
                Thread.sleep(900000);
            }
        } catch (InterruptedException e){}
    }
}
```

Figure 4: class Customer.

A customer spend his life getting his hair cut. Occasionally, but at most once every half hour, he goes to the barbershop and wants a haircut (hair grows fast in this town). If the barbers chair is occupied he must wait for the chair to become available. The barbershop contains only a single chair. When a haircut is finished the customer leaves the shop and the chair becomes available. The implementation of a customer (the thread Customer) can be found in Figure 4.

```
public class Shop {
    private boolean occupied = false;

    public synchronized void get_haircut() throws InterruptedException {
        while (occupied) wait();
        occupied = true;
        notifyAll();
    }

    public synchronized void get_customer() throws InterruptedException {
        while (!occupied) wait();
    }

    public synchronized void finished_cut() throws InterruptedException {
        occupied = false;
        notify();
    }
}
```

Figure 5: class Shop.

The shop is implemented as a monitor in Figure 5. The boolean variable `occupied` tells whether the chair is occupied.

**Question 2.1** Consider the notification principle in Figure 5. Should it be altered or not in order to make the barbershop deadlock free? In either case give an explanation.

```
public class Shop {  
  
    private boolean occupied = false;  
    private boolean open = false;  
  
    public synchronized void get_haircut() throws InterruptedException {  
        ...  
    }  
  
    public synchronized void get_customer() throws InterruptedException {  
        while (!occupied) wait();  
    }  
  
    public synchronized void finished_cut() throws InterruptedException {  
        open = true;  
        notifyAll();  
        while (open) wait();  
        occupied = false;  
        notifyAll();  
    }  
}
```

Figure 6: class Shop extended with an exit door.

To control the entering and leaving in and out of the barbershop an *exit door* is introduced, see Figure 6. Customers must enter the shop as before but they are required to leave through the exit door. The boolean variable `open` represents as to whether the door is open or not. Now, in order to finish a haircut session the customer must leave through the exit door waiting for the door to be opened by the barber, and the barber must wait for the customer to close the door.

**Question 2.2** Complete the partial class definition in Figure 6. Explain your solution. There is no guarantee that a customer ever will obtain a haircut although he is waiting for one, so a customer may be *starved*.

**Question 2.3** Why may customers be starved? Give a short explanation.

To partly remedy the starvation problem we extend the barbershop with three *waiting chairs* where customers may sit before being allowed to enter the barbers chair. In order to ensure *fairness* among the customers sitting on the waiting chairs we want to impose a *first come first serve* mechanism for these.

**Question 2.4** Add the three waiting chairs and the requested first come first serve mechanism to your solution of Question 2.2. Only the method `get_haircut()` may be altered. Hint: Apply the *ticketing mechanism*

**Question 2.5** Why is the scenario suggested above only a partly solution to the starvation problem?