CS4532 Concurrent Programming
Homework - Answers
P.R.L. Fernando (140161F)

Question 1 - Answer

Thread1

```
sync(account1);
sync(account2);
    account1.withdraw(amount);
    account2.deposit(amount);
release(account2);
release(account1);
```

Thread 2

```
sync(account2);
sync(account1);
    account2.withdraw(amount);
    account1.deposit(amount);
release(account1);
release(account2);
```

Assume a situation where one thread try to transfer from account1 to account2 and another thread try to transfer from account2 to account1 and lines indicated by green colour are executed. Then each thread can not go beyond the lines indicated by red. **Hence deadlock is occurred and this code not satisfy the liveness property.**

Thread1

```
void withdraw(double amount){
   // read balance
   // reduce amount
   // save balance
}
```

Thread 2

```
void withdraw(double amount){
   // read balance
   // reduce amount
   // save balance
}
```

Assume a situation where two threads access same account. Balance -= amount is not a atomic operation. (read balance, reduce amount, safe balance). Assume a situation where each thread completed lines indicated with green colour. Then both threads has the same balance. When executing lines indicated with red colour changes by the thread executes last will remain other changes are lost. **Hence withdraw and deposit functions not satisfy the safety property.**

Question 2 - Answer (Implementation using Java)

```java
public class Satisfactions {
  // use to notify when there is a pizza available
  public final static Semaphore pizza = new Semaphore(0);
  // use to notify when there is a order available
  public final static Semaphore order = new Semaphore(0);
  // mutex used when reading and updating availableSlices variable
  public final static Semaphore mutex = new Semaphore(1);
  // variable holds the available number of pizza slices
  public static int availableSlices = 0;
}

public class PizzaDelivery extends Thread {
   // run method implementation not provided. execute() method will call inside run()
  private int slicesForPizza = 8;

  private void execute() throws InterruptedException {
     while (true) {
        // delivery thread will wait for the signal
        // when there are no pizza only one student thread will notify while others sleep
        // assume a situation where student thread signal before delivery thread wait
        // semaphore will store previous wake calls hence this does not affect the execution
        Satisfactions.order.acquire();
        // at this point all other thread are blocked and only pizza delivery thread is accessing
        Satisfactions.availableSlices = slicesForPizza;
        // notify student thread
        Satisfactions.pizza.release();
     }
  }
}
```

```java
public class Student extends Thread {
  // run method implementation not provided. execute() method will call inside run()

  private void execute() throws InterruptedException {
     while (true) {
        // only one thread can access the critical section
        Satisfactions.mutex.acquire();
        //============================= critical section =============================//
        // when there are no pizza slices available
        if (Satisfactions.availableSlices == 0) {
           // notify pizza delivery thread
           Satisfactions.order.release();
           // student thread will wait for the signal
           // assume a situation where pizza delivery notify before student thread wait
           // semaphore will store previous wake calls hence this does not affect the execution
           Satisfactions.pizza.acquire();
        }
        // in this point there should be pizza available to grab
        // if there are enough slices then availableSlices > 0
        // if there are not enough slices then delivery should happen hence availableSlices > 0
        Satisfactions.availableSlices--;
        //========================== end of critical section ==========================//
        // this will release mutex, other threads can enter the critical section
        Satisfactions.mutex.release();
        study();
     }
  }

  private void study() {
     // student will study while eating pizza
  }
}
```

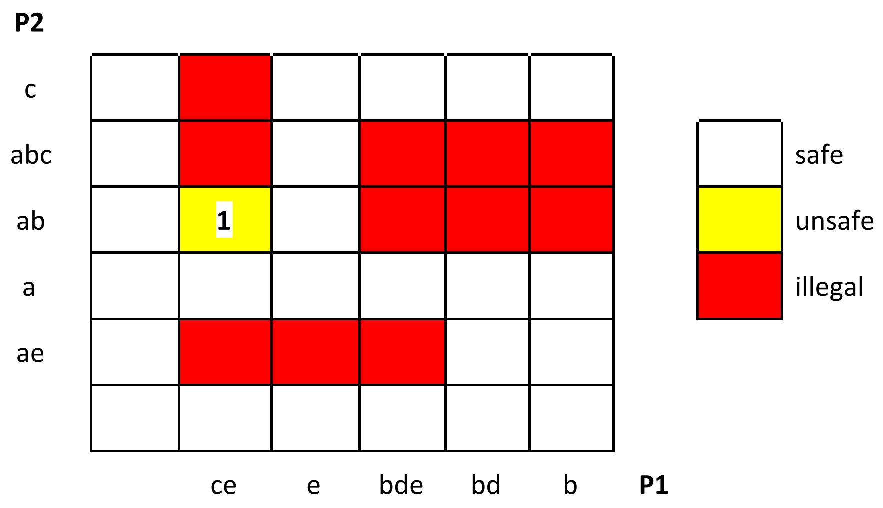
Question 3 (a) - Answer (Implementation using Java)

```java
public class Satisfactions {
  // keep the ticket price to calculate income this is a read only variable hence no mutual exclusion
  public final static double ticketPrice = 1000.00;
  // keep track of available tickets
  public static int availableTickets = 10;
  // keep track of income received
  public static double income = 0.00;
  // guard available tickets variable
  public final static Semaphore mutex1 = new Semaphore(1);
  // guard income received variable
  public final static Semaphore mutex2 = new Semaphore(1);
}

public class TicketIssuer extends Thread {
   // run method implementation not provided. bookTicket() method will call inside run()
  private int ticketsNeeded;

  private void bookTicket() throws InterruptedException {
     Satisfactions.mutex1.acquire();
     // ============================== critical section 1 ===============================
     boolean isAvailable = Satisfactions.availableTickets >= ticketsNeeded;
     if (isAvailable) Satisfactions.availableTickets -= ticketsNeeded;
     // =========================== end of critical section 1 ============================
     Satisfactions.mutex1.release();
     if (isAvailable) this.pay();
  }

  private void pay() throws InterruptedException {
     // some thread can grab tickets white another thread is at the payment section
     Satisfactions.mutex2.acquire();
     // ============================== critical section 2 ===============================
     Satisfactions.income += Satisfactions.ticketPrice * ticketsNeeded;
     // =========================== end of critical section 2 ============================
     Satisfactions.mutex2.release();
  }
}
```

Question 3 (b) - Answer (Implementation using Java)

```java
public class CurryMaker extends Thread {
   // run method implementation not provided. create() method will call inside run()

  private void create() throws InterruptedException {
     // since both serve at the same time curry maker need to wait until food maker make 2 foods
  }
}

public class FoodMaker extends Thread {
  // run method implementation not provided. create() method will call inside run()

  private void create() throws InterruptedException {
     // since both serve at the same time food maker need to wait until curry maker make 2 curries
  }
}

public class Shop extends Thread {
   // run method implementation not provided. execute() method will call inside run()

  private void execute() throws InterruptedException {
     // assumption1: start creating curries when there are two customers available
     // assumption2: both of them create 2 foods and 2 curries
     // assumption3: after serving those they will wait for another 2 customers
     while (true) {
        FoodMaker foodMaker = new FoodMaker();
        CurryMaker curryMaker = new CurryMaker();
        // fork and join technique used
        // fork child threads to execute parallel tasks
        foodMaker.start();
        curryMaker.start();
        // join main thread to the end of child threads
        foodMaker.join();
        curryMaker.join();
        // child threads are completed
        serve();
     }
  }

  private void serve() {
     // serve foods to customer
  }
}
```

Question 4 - Answer



“1” block

Resources assigned

	a	b	c	d	e
P1	1	1	0	0	0
P2	0	0	1	0	1

Resources still needed

	a	b	c	d	e
P1	1	1	1	0	1
P2	0	1	1	1	1

- At “1” block P1 needs “c” and “e” to complete P2 needs “b” and “d” to complete and only resource that is free is “d”. Since P1 holds “b” and P2 holds “c” according to Bankers algorithms these two processes cannot complete hence the state is unsafe.

All states in the bottom row and in the left column are safe. If either row 1 or column 1 states are followed, the processes execute in sequence, not interleaved, so no deadlock can occur.

Question 5 - Answer

Since philosophers flip coin to select a fork and if other fork is not available he/she release the remaining fork **the solution is deadlock-free**, however **starvation may occur**. The philosophers are never stuck waiting and unable to do “useful work”. However, a philosopher might not ever get a chance to eat because at least one of his/her forks is busy.