1 Concurrency

1. What does concurrency, distribution, parallelity stand for? What is the difference between a thread and a process?

2. The wait() statement in the bounded buffer and in the readers-writers solution was surrounded by a **while** loop. Explain, why you need to use a **while** around the wait and not an **if**.

3. A group of people wants to match up. There is one method findPartner(**int**). Each thread calls it with its own number to find a partner.

   If there are two calls findPartner(3) and findPartner(7), both of them should return 3 or both of them 7. If there is just one call it waits for another call. The following solution of course does not work!

```
class MatchUp {
    boolean empty;
    int who;

    public MatchUp() {
        empty = true;
    }

    int findPartner(int i) {
        if (empty) {
            who = i;
            empty = false;
            while (!empty)
                ;
            return i;
        } else {
            empty = true;
            return who;
        }
    }
}
```

   (a) How can the above solution fail in a race condition?

   (b) How can the above solution fail in a deadlock?

   (c) Correct the class MatchUp using two binary semaphores, one as mutex, the other one which also reflects the empty-condition.

   (d) Correct the class MatchUp using normal Java synchronisation.

   (e) Does your solution need notifyAll() or notify() ? Does it matter?

4. For the readers-writers solution with semaphores.

```
class database {
    Semaphore mutex;
    Semaphore db;
    int readerCount;

    public database () {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }

    void startRead() {
        mutex.P(); // 1
        readerCount ++;
        if (readerCount == 1)
            db.P(); // 2
        mutex.V(); // 3
    }

    void endRead() {
        mutex.P(); // 4
        readerCount --;
        if (readerCount == 0)
            db.V(); // 5
        mutex.V(); // 6
    }

    void startWrite() {
        db.P(); // 7
    }
    void endWrite() {
        db.V(); // 8
    }
}
```

- What would happen, if we forget 1 or 2 or 3?
- What would happen, if we accidentally pick the wrong one for 4 or 5 or 6?
- What would happen if we exchange 7 and 8?
- What would happen if we forget `readerCount --`?
- What would happen if we forget `readerCount ++`?

Do we get a race condition or a deadlock?

5. Write a program which starts a new thread, waits until this thread has done some task and then prints "Bye". Here, the task of the thread will be to write "Hello".

2 Solutions

1. -

2. After a notify() the thread which goes from the wait to the entry set is often not continued immediately. Many things might happen before it continues. Also the condition we were waiting for might not be true any more. (In the bounded buffer we may wait until the buffer is not empty, but when the thread continues after the wait, the buffer may be already empty again).

   In the case of the notifyAll(), often the condition we are waiting for is not even true at the time of the notifyAll()-call (In the readers-writers example a finished reader puts all waiting threads in the entry set, but a writer may not be able to continue, because there are still readers in the data base).

3. A group of people wants to match up. There is one method findPartner(**int**). Each thread calls it with its own number to find a partner.

   (a) Assume empty = false. T1 is waiting in the loop. who = 4 (this was T1's argument). Now T2 calls findPartner(3), goes into the else-part sets empty=true. Before the return T3 calls findPartner(7), goes into the then-part and sets who=7. Now T1 and T2 continue to the return. But they do return different values, 7 and 4.

   There are many more solutions to this question.

   (b) Assume empty = true. T1 and T2 calling findPartner at almost the same time. They both enter the then-part. Both set empty to false. and both wait in the loop. Instead they should have returned.

   (c) Correct the class MatchUp using two binary semaphores.

```
class MatchUp {
    int who;
    boolean empty;
    Semaphore sEmpty, mutex;

    public MatchUp() {
        sEmpty = new Semaphore(0);
        mutex = new Semaphore(1);
        empty = true;
    }

    int findPartner(int i) {
        mutex.P()
        if (empty) {
            who = i;
            empty = false;
            mutex.V();
            sEmpty.P();
            return i;
        } else {
            empty = true;
            int i = who;
            mutex.V();
            sEmpty.V()
            return i;
        }
    }
}
```

(d) Correct the class MatchUp using normal Java synchronisation.

```
class MatchUp {
    int who;
    boolean empty;

    public MatchUp() {
        empty = true;
    }

    synchronized int findPartner(int i) {
        if (empty) {
            who = i;
            empty = false;
            try {
                wait ();
            } catch (InterruptedException e) { }
            return i;
        } else {
            empty = true;
            notify();
            return who;
        }
    }
}
```

(e) Here at most one thread is in the wait set, so it doesn't matter!

4. For the readers-writers solution with semaphores.
   - Forget 1: Race Condition.
   - Forget 2: Race Condition.
   - Forget 3: Deadlock.
   - Wrong 4: Race Condition.
   - Wrong 5: Deadlock.
   - Wrong 6: Deadlock.
   - Exchange 7 and 8: Race Condition.
   - Forget readerCount —: Deadlock.
   - Forget readerCount ++: Race Condition.

5. The class HelloBye

```
public class HelloBye implements Runnable {

    public void run() {
        System.out.println("Hello");
        synchronized (this) {
            notify();
        }
    }
```

```
    public void main() {
        Thread thread = new Thread(this);
        thread.start();
        synchronized (this) {
            try {
                wait();
            } catch (InterruptedException exception) {
            }
        }
        System.out.println("Bye");
    }

    public static void main(String[ ] args) {
        new HelloBye().main();
    }
}
```