# yz709-FJ-sup2

## Question 1

Describe why the definition of the Vector Clock algorithm, as found in this course, is different from that found in the Part IB Concurrent and Distributed Systems.

- Not represented as a fixed-length vector:
    - Since the number of clients is not known initially in this course, we cannot represent the vector clock as a fixed-length vector, as defined in the CDS course.
    - Hence, we use a `Map<String, Integer>` where each key in the map represents a client id and the value represents the clock counter for that client.

💡 Comments:

## Question 2

What are the rules for updating a Vector Clock?

- Each chat client should maintain its local vector clock for as long as the client is alive.
- On sending a message from client $i$, client $i$ needs to increment its clock elements by one before attaching a copy of its vector clock to the message

because sending a message is an event as well, after all. `client[i] += 1`

- On receiving a message, client $i$ compares the receiving vector clock and its local vector clock entry by entry, pick the largest value to be the updated value for each entry, then increment its own entry in the map by one. `client[j] = max(client[j], msgclient[j]); client[i] += 1` where $j$ includes all alive client ids.

💡 Comments:

## Question 3

Complete Table ~~1~~ 2 in Workbook 3 and explain the result.

- The expected results should be `a.bal = 110, b.bal=90`, this execution trace gives the correct results.
- The local variable `tmp1` is set by Thread `S` at the end of CPU cycle 2. When utilising it at CPU cycle 5, since the balance of `a` has not been modified by Thread `T` yet, `tmp1` still preserve the correct value.
- The local variable `tmp2` in Thread `S` fetches the latest balance of `b`, and from CPU cycle 6 to CPU cycle 7, Thread `T` has not changed the balance of `b` in the meantime. Therefore, `tmp2` still preserve the correct value when being read.
- Similarly, the local variable `tmp2` in Thread `T` fetches the latest balance of `a` and uses `tmp2` directly in the following CPU cycle, indicating that `tmp2` preserves the correct value when read.
- This execution order can be written as $R_S(a), R_T(b), W_T(b), R_S(b), W_S(b), R_T(a), W_T(a)$, this is a serialisable execution order because we can turn it into serial execution order $R_T(b), W_T(b), R_T(a), W_T(a) R_S(a), R_S(b), W_S(b)$ by exchanging non-conflict transactions.

|  | Thread S |  |  | Shared |  | Thread T |  |  |
|---|---|---|---|---|---|---|---|---|
|  | a.transferTo(b,10) | tmp1 | tmp2 | a.bal | b.bal | b.transferTo(a,20) | tmp1 | tmp2 |
| 1 |  |  |  | 100 | 100 |  |  |  |
| 2 | tmp1 = a.bal-10 | 90 |  | 100 | 100 |  |  |  |
| 3 |  |  |  | 100 | 100 | tmp1 = b.bal-20 | 80 |  |
| 4 |  |  |  | 100 | 80 | b.bal = tmp1 | 80 |  |
| 5 | a.bal = tmp1 | 90 |  | 90 | 80 |  |  |  |
| 6 | tmp2 = b.bal+10 | 90 | 90 | 90 | 80 |  |  |  |
| 7 | b.bal = tmp2 | 90 | 90 | 90 | 90 |  |  |  |
| 8 |  |  |  | 90 | 90 | tmp2 = a.bal+20 | 80 | 110 |
| 9 |  |  |  | 110 | 90 | a.bal = tmp2 | 80 | 110 |
| 10 |  |  |  |  |  |  |  |  |

Table 2

Comments: Since table 1 has already been filled in the workbook3, I assume this question refers to table 2 instead.

## Question 4

A student attempts to implement fine-grained locking to provide atomicity for the transferTo method of the BankAccount class as described in Workbook 3. Their solution is as follows:

```
class BankAccount {
  private int balance;
  private int account;
  public void transferTo(BankAccount b, int amount) {
    synchronized(this) {
      balance -= amount;
    }
    synchronized(b) {
      b.balance += amount;
    }
  }
}
```

Describe why this solution is incorrect and why testing for correctness is hard. Describe in words how to write a correct implementation. Provide your supervisor with a copy of your work for Ticklet 3.

- Two separate `synchronised` cannot make sure the total capital held by the back keeps constant when other unrelated transactions are interleaving.

```
A.transferTo(B, 20)
             A.balance = 100, B.balance = 100, total balance = 200
CPU cycle 1: synchronise(this) {balance -= amount;}
             A.balance = 80, B.balance = 100, total balance = 180
CPU cycle 2: other transactions (even unrelated ones)
             A.balance = 80, B.balance = 100, total balance = 180
CPU cycle 3: synchronise (b) {b.balance += amount;}
             A.balance = 80, B.balance = 120, total balance = 200
```

- It cannot prevent deadlock because the order of acquiring these locks is important.
  - If we nest the synchronised blocks as follows, then there might be two concurrent transactions `A.transferTo(B, 20); B.transferTo(A, 10);`
  - The first one acquires lock `A`, the second one acquires lock `B`, but after executing `balance -= amount`, both will be in a deadlock state waiting for the other lock whilst not releasing their lock.

```
public void transferTo(BankAccount b, int amount) {
  synchronised (this) {
      balance -= amount;
      synchronized (b) {
        b.balance += amount;
      }
    }
  }
```

- Why testing is hard:
  - Simulating transactions such that they are concurrent is not easy because of the fast CPU speed. We need large enough transactions happening around the same time to reach a concurrent scenario. However, it is easy to get concurrent transactions in a distributed system in real life, especially with many clients.
  - Tracking the total balance is not easy as well, because the period where the total balance is not constant is short, we need to have an interleaving transaction which takes a long time in order to observe the inconsistent total

    balance, but making a perfect interleaving transaction that executes between two synchronised blocks require a lot of effort.

- A correct implementation - Ticklet 3 on chime:
    - Nested synchronised blocks to ensure the total capital is always conserved even when other transactions unrelated are present.
    - Prevent deadlock by always acquiring the lock on account with the, e.g., smallest account number first.

```
private class BankAccount {
    private int balance;
    private int acc;

    public void transferTo(BankAccount b, int amount) {
      if (b.acc > acc) {
        synchronized (this) {
          balance -= amount;
          synchronized (b) {
            b.balance += amount;
          }
        }
      } else {
        synchronized (b) {
          b.balance += amount;
          synchronized (this) {
            balance -= amount;
          }
        }
      }
    }
  }
```

💡 Comments:

## Question 5

Complete the following exam questions:

First, write your solutions on paper. Then type up your answers and test them on a computer.
What errors did you make when writing solutions on paper, and how could you avoid making
them in the exam hall?

## y2010p3q9

Fellows at Norisbon College dine at a circular table on which there is a single fork between each Fellow. Fellows either eat or think, and always start dinner thinking. To eat, a Fellow first picks up the fork immediately to his left and, once successful, picks up the fork immediately to his right. When a required fork is not on the table, the Fellow waits, neither eating nor thinking, until the fork is returned to the table. After eating, a Fellow returns both forks to the table. No cutlery is required to think.

Your task is to model the above scenario in Java.

(*a*) Write a class called **`Fork`** with two public methods, **`pickUp`** and **`putDown`**. The methods should take no arguments and return no result. An instance of **`Fork`** should act as a lock to prevent concurrent access. In other words, once **`pickUp`** has been called, all further calls to **`pickUp`** should block until **`putDown`** is called; when **`putDown`** is called, one caller (if any) who is blocked should proceed. [7 marks]

```
public class Fork {
    private boolean inUse = false;

    public synchronized void pickUp() {
        if (inUse) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        inUse = true;
        System.out.println("Pick up a fork.");
    }

    public void putDown() {
        inUse = false;
        System.out.println("Pick down a fork.");
        this.notify();
    }
}
```

(*b*) Write a class called **`Fellow`** which inherits from the **`Thread`** class and implements the abstract method **`run`**. The **`Fellow`** class should have a single constructor which takes two **`Fork`** objects, one representing the fork to the Fellow's left, and one to the right. When run, an instance of **`Fellow`** should think for ten seconds, eat for ten seconds and think for ten seconds before terminating. [7 marks]

```
public class Fellow extends Thread {
    private Fork left;
    private Fork right;

    Fellow(Fork left, Fork right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public void run() {
        try {
            // think for 10 seconds
            Thread.sleep(10000);
            // eat for 10 seconds
            left.pickUp();
            right.pickUp();
            Thread.sleep(10000);
            left.putDown();
            right.putDown();
            // think for 10 seconds
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted");
        }
    }
}
```

$(c)$ Describe when and why your implementation may suffer deadlock. [2 marks]

- If multiple fellows sit at a round table and all of them pick up their left forks concurrently before picking up their right forks, then no one can pick up their right forks; hence they are all waiting for the right forks without processing further.
- This is a deadlock scenario because all threads are spinning and waiting for some other threads to give up resources.

$(d)$ By altering the order in which the forks are picked up, describe how you would modify your implementation so that it does not suffer deadlock. [4 marks]

- We need to give fellows a unique id number; for even fellows, try to pick the left fork up first before picking up the right fork, and for odd fellows, pick the right fork before picking up the left fork.

- Hence fellows adjacent to each other would either pick up both forks or none of the forks and need to spin to wait for resources to be released.

💡 Comments:

## y2011p3q7

In this question you will need to fill in missing parts of a Java program. You may ignore any exception handling and will not be penalised for minor syntactic errors.

You are provided with a class `Eval`:

```
public class Eval {
  public static int f(Record r) { ... }
}
```

(*a*) Add another method `Integer maxf(Iterator<Record> it)` to the class `Eval`. Your method should return the maximum value computed by `f` for every `Record` returned by the iterator or `null` if there are no records available. The relevant portion of the `Iterator` interface is as follows:

```
interface Iterator<T> {
  // return true if there are more values available
  public boolean hasNext();

  // return the available value and advance to the next one
  public T next();
}
```

[5 marks]

```
public static Integer maxf(Iterator<Record> it) {
        // return the maximum value computed by f
        // for every Record returned by the iterator or null
        // if there are no records available
        Integer maxV = null;
        while (it.hasNext()) {
            int value = f(it.next());
            maxV = maxV == null || maxV < value ? value : maxV;
        }
        return maxV;
    }
```

(*b*) Complete the methods **run()** and **join()** in the following abstract class. You may add additional fields or methods if you wish.

```
abstract class Joinable implements Runnable {
  abstract void exec();
  final public void run() {
    // ... call the exec() method ...
  }
  void join() throws InterruptedException {
    // block the calling thread until exec() completes in run()
  }
}
```

[7 marks]

```
public abstract class Joinable implements Runnable {
    abstract void exec();

    private boolean executing = false;

    final public void run() {
        // call the exec() method
        synchronized (this) {
            executing = true;
            this.exec();
            executing = false;
        }
        this.notify();
    }

    void join() {
        // block the calling thread until exec() completes in run()
        if (executing) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        run();
    }

}
```

(*c*) Provide a method **Integer parmaxf(Iterator<Record> it, int n)** which is functionally equivalent to **Eval.maxf**, except that it should create **n** parallel threads of execution to speed up the calculation of the result. You may assume that **Iterator<Record>** is thread-safe. You may find it helpful to subclass the **Joinable** class. [8 marks]

```
class Unit extends Joinable {
    private Iterator<Record> it;

    Unit(Iterator<Record> it) {
        this.it = it;
    }

    Integer maxV = null;

    @Override
    void exec() {
        maxV = Eval.maxf(it);
    }

}

class Eval {
  //...
  public static Integer parmaxf(Iterator<Record> it, int n) {
        // same functionality as maxf
        // but with n parallel threads of execution for computation
        List<Unit> units = new ArrayList<Unit>();
        for (int i = 0; i < n; i++) {
            units.add(new Unit(it));
            new Thread(units.get(i)).start();
        }

        for (Unit u : units) {
            u.join();
        }
        Integer maxV = null;
        for (Unit u : units) {
            maxV = maxV == null || maxV < u.maxV ? u.maxV : maxV;
        }
        return maxV;
    }
}
```

💡 Comments:

## y2013p3q7

A Java developer implements a class loader as follows:

```
public class NetworkClassLoader extends ClassLoader {

 private String server;
 private int port;

 NetworkClassLoader(String server, int port) { ... }

 @Override
 public Class<?> findClass(String name)
            throws ClassNotFoundException {
  //TODO: download class file and place contents in byte array b.
  return defineClass(name, b, 0, b.length);
 }
}
```

(*a*) Describe what a Java class loader is, when it is used, and why a developer might need to implement their own class loader. [3 marks]

- JVM controls when and how new pieces of program code (i.e., classes) are loaded and executed. The class loader is the part of JVM responsible for loading class definitions.
- It looks for `.class` files at various locations on disk and inside Jar files when an instance of a class or a static field in a class is first referenced.
- Possible to extend the class loader to load definitions of classes at runtime from other locations (e.g., website) or over a `Socket` object.

(*b*) Write a Java program which accepts two arguments on the command line: a network port number and the full path to a Java class file in the file system. When executed, your program should wait indefinitely on the specified port for connections from clients. Whenever a client connects, your program should send the contents of the Java class file to the client. [8 marks]

```
public static void main(String[] args) throws IOException {
        if (args.length != 2) {
            throw new IllegalArgumentException("Program usage:<port:int> <file path:st
ring>");
        }
        int port;
        String filepath;
        ServerSocket socket = null;
        try {
            port = Integer.parseInt(args[0]);
            filepath = args[1];
            // get the file contents and copy it into byte array
            File f = new File(filepath);
            FileInputStream in = new FileInputStream(filepath);
            int fileSize = (int) f.length();
            byte[] b = new byte[fileSize];
            int val = -1;
            int total = 0;
            while ((val = in.read(b)) != -1 && total < fileSize) {
                total += val;
            }
            in.close();
            socket = new ServerSocket(port);
            while (true) {
                Socket s = socket.accept();
                s.getOutputStream().write(b);
                s.close();
            }
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Program usage:<port:int> <file path:st
ring>");
        } catch (IOException e) {
            System.out.println("IOException from the socket.");
        } finally {
            if (socket != null) {
                socket.close();
            }
        }
    }
```

(*c*) Complete the method **findClass** by downloading the class file from the program you wrote in part (*b*). Any error states should generate a **ClassNotFoundException**. [8 marks]

```
@Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        // download class file and place contents in byte array b
        byte[] tmp = new byte[16]; // stores a chunk of contents of the file
        byte[] b = new byte[16]; // stores all contents of the file
        Socket socket = null;
        try {
            socket = new Socket(this.server, this.port);
            InputStream in = socket.getInputStream();
            int flag = -1;
            int readLength = 0;
            while ((flag = in.read(tmp, readLength, tmp.length - readLength)) != -1) {
                readLength += flag;
                byte[] b2 = Arrays.copyOf(b, readLength); // b2 = b + [0,0,...,0], wit
h len(b2) = readLength
                System.arraycopy(tmp, 0, b2, b.length, flag); // b2 = b + tmp(len = fl
ag), with len(b2) = readLength
                b = Arrays.copyOf(b2, readLength); // b = b2
            }
            // after read the whole file
            return defineClass(name, b, 0, b.length);
        } catch (IOException e) {
            throw new ClassNotFoundException(e.getMessage());
        } finally {
            try {
                if (socket != null) {
                    socket.close();
                }
            } catch (IOException e) {
                throw new ClassNotFoundException(e.getMessage());
            }
        }
    }
```

$(d)$ Outline a security vulnerability which might arise when using your implementation of **NetworkClassLoader** from part $(c)$ together with the server you wrote for part $(b)$. [1 mark]

- We would read the file contents without verifying the content type; hence, there might be a possibility of a denial of service attack if the file contents consist of some executable codes.

💡 Comments:

## y2016p3q6

$(a)$ Describe the operation of **wait()** and **notifyAll()**. Ensure that your answer explains when locks are acquired and released. [5 marks]

- When thread $A$ waits for thread $B$ to do something, thread $A$ simply calls the method `wait()` on a Java object for which it already holds the lock, then thread $A$ releases lock and pause execution.
- Thread $B$ acquire the lock that thread $A$ released, complete any necessary work, and call `notify()` to wake up thread $A$ when any shared data structures are in a consistent state. `notifyAll()` to wake up all threads waiting on an object. Then thread $A$ reawakes, reacquire the lock, and continues execution.
- Calling `wait()` or `notify()` on a Java object you don't hold the lock, you will receive a `java.lang.IllegalMonitorStateException` at runtime.

(*b*) A *future* is a mechanism to store the eventual result of a computation done in another thread. The idea is that the computation is run asynchronously and the calling thread only blocks if it tries to use a result that hasn't been computed yet. An example program using a future is shown below.

```
Future<String> f = new Future<String>() {
    @Override
    public String execute() {
       // ...long running computation...
       return data;
    };

// ...

String result = f.get(); // blocks if execute() unfinished
```

Use `wait()` and `notifyAll()` to provide an implementation of the `Future` class that would work with the example program above. [10 marks]

```
public abstract class Future<T> {
    private T data;
    private boolean executed = false;

    public Future() {
        Thread f = new Thread() {
            @Override
            public void run() {
                data = execute();
                executed = true;
                Future.this.notifyAll();
            }
        };
        f.start();
    }

    public abstract T execute();

    public synchronized T get() {
        if (!executed) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return data;
    }
}
```

(*c*) Give one potential advantage and one potential disadvantage of using **`notify()`** instead of **`notifyAll()`**. [2 marks]

- Advantage:
  - The function `notify()` takes less computation power than the function `notifyAll()` because it only processes with one awaiting thread rather than all threads in the waiting queue.
- Disadvantage:
  - With `notify()`, we can only wake up one thread; however, it is possible that this thread is waiting for multiple resources and cannot run immediately after being granted one of the requested resources. In contrast, `notifyAll()` wakes all awaiting threads to not miss out on runnable threads.

(*d*) Would it have been beneficial to use **`notify()`** instead of **`notifyAll()`** in your implementation? Justify your answer. [3 marks]

- It is better to use `notifyAll()` because, in the program, we assume the threads would be blocked if the requested computation results do not exist. If we have executed the computation, it makes sense to awake all of the awaiting threads because of the assumption.
- If we use `notify()`, then only one of the awaiting threads would get the results, and all other threads are blocking forever; we need another loop to wake them up one by one.

💡 Comments:

# Question 6(optional)

Java serialisation allows malicious clients to subject a server to a denial-of-service attack (a serialisation bomb; see Question 1.8). One alternative is to use a different encoding format (e.g. JSON, Protobuf) for messages sent between the client and the server. Rewrite your implementation of the Chat Client and Chat Server to use an alternative encoding format for messages. We strongly recommend you make use of an existing library to support encoding and decoding messages.

💡 Comments:

## Question 7(optional)

The Java NIO libraries provide support for non-blocking sockets. This means you can test whether data is available on a socket without blocking. This will allow you to write a variation of the Chat Server, which only requires a single thread (rather than 2n+1 where n is the number of clients). Your task is to research and implement a new Chat Client and Chat Server that does this.
Hint: If you decide to use serialisation to support sending and receiving of messages, then you will need to encode and send the payload length separately before the object so you can determine when you have received enough data that you can successfully read an object without blocking. Alternatively, you may choose to use a different encoding format (e.g. JSON, Protobuf) for the messages sent between the client and the server.

💡 Comments: