Instructions:
Be concise and clear. Simplicity is an grading criterion. Use of books, notes, any other reference materials or digital utilities is not allowed. Ensure that every answer sheet contains your name, student number, the course code and the total number of answer sheets submitted for grading. Place the answer to each question on a separate page. Please place answers to questions 1 and 2 on one sheet of paper and the answers of 3, 4 and 5 on another sheet, this will help speed up the grading process.

1. Select three topics from the following list. Define the concepts and write a short essay on the topic. (6)
   (a) Differences between processes and threads
   (b) Amdahl's law and scalability of parallelism
   (c) Concurrency hazards: race conditions, deadlocks and livelocks
   (d) Differences between the general monitor and the corresponding Java construct
2. Using axiomatic reasoning (the invariant method), prove or disprove that the following algorithm for two threads satisfies: (6)
   (a) Mutual exclusion of the critical sections of P and Q
   (b) Liveness

| boolean wantp ← false, wantq ← false<br>integer turn ← 1 | |
|---|---|
| **P** | **Q** |
| loop forever<br>p1: non-critical section<br>p2: wantp ← true<br>p3: if wantq<br>p4: if turn = 2<br>p5: wantp ← false<br>p6: await turn = 1<br>p7: wantp ← true<br>p8: await wantq = false<br>p9: critical section<br>p10: wantp ← false<br>p11: turn ← 2 | loop forever<br>q1: non-critical section<br>q2: wantq ← true<br>q3: if wantp<br>q4: if turn = 1<br>q5: wantq ← false<br>q6: await turn = 2<br>q7: wantq ← true<br>q8: await wantp = false<br>q9: critical section<br>q10: wantq ← false<br>q11: turn ← 1 |

3. Implement an *exchanger* in Java according to the following interface. You may assume that the data objects are unique and not null. You may not use `java.util.concurrent`. You may use collections from `java.util`. (6)

```
1  /**
2   * A synchronization point at which threads can pair and swap data elements
3   * of type T within pairs. Each thread presents a data element on entry to the
4   * exchange method, matches with a partner thread, and receives its partner's
5   * data element in return.
6   * @param <T>
7   */
8  public interface Exchanger<T> {
9
10     /**
11      * Wait for another thread to arrive at this exchange point (unless the
12      * current thread is interrupted), and then transfers the given data element
13      * to the other thread, receiving it's data element in return.
14      * @param dataElement to give to another thread
15      * @return dataElement received from another thread
16      * @throws InterruptedException
17      */
18     public T exchange(T dataElement) throws InterruptedException;
19 }
```

4. Implement a *latch* using a tuple space and the operations `postnote`, `readnote` and `removenote`. A latch is a synchronization mechanism which allows multiple threads to wait for a single event, the release of the latch. When the latch is released, all threads that are currently waiting on the latch can proceed again. The latch can be used multiple times. Explain the content and purpose of all of the tuple types. (6)

```
operation initialize():
//add code

operation await():
//Block until release() is called
//add code

operation release():
//Release all threads currently blocked on await()
//add code
```

5. Analyse the following single producer, single consumer system. Define requirements that the system must meet for it to be correct. Define a set of invariants which capture the correctness of the system. You do not need to prove the invariants. Show that the system is totally correct. (6)

   You may use the following assumptions:

   1. The methods `produce()` and `consume()` will always eventually return
   2. The method `produce()` will eventually always return a null value
   3. The system is started by `(new Consumer<Datatype>()).start()`

   Hints: Start with invariants for Semaphore, BlockingQueue and then use these in constructing invariants for the Producer and the Consumer. Consider constructing the invariants for Semaphore and BlockingQueue based on their state before a method call and after that call returns.

Semaphore.java

```
1  public class Semaphore {
2      private int value;
3      public Semaphore(int v) { this.value = v; }
4
5      public synchronized void acquire() throws InterruptedException {
6          while(this.value < 1) {
7              this.wait();
8          }
9          this.value--;
10     }
11
12     public synchronized void release() {
13         this.value++;
14         this.notify();
15     }
16 }
```

BlockingQueue.java

```
1  import java.util.ArrayDeque;
2  import java.util.Queue;
3
4  public class BlockingQueue<T> {
5      private static final int CAPACITY = 3;
6
7      private final Queue<T> queue = new ArrayDeque<T>();
8      private final Semaphore isFull = new Semaphore(0);
9      private final Semaphore isEmpty = new Semaphore(CAPACITY);
10
11     public T get() throws InterruptedException {
12         T item;
13         isFull.acquire();
14         item = queue.remove();
15         isEmpty.release();
16         return item;
17     }
```

```
18
19      public void put(T item) throws InterruptedException {
20          isEmpty.acquire();
21          queue.add(item);
22          isFull.release();
23      }
24  }
```

Producer.java

```
1   public class Producer<T> extends Thread {
2       private final BlockingQueue<T> queue;
3       public Producer(BlockingQueue<T> q) { this.queue = q; }
4
5       public void run() {
6           T data;
7           while(true) {
8               try {
9                   data = produce();
10                  queue.put(data);
11              } catch (InterruptedException e) { break; }
12          }
13      }
14
15      public T produce() {
16          //Code omitted
17          return anInstanceOfT;
18      }
19  }
```

Consumer.java

```
1   public class Consumer<T> extends Thread {
2       private final BlockingQueue<T> queue = new BlockingQueue<T>();
3       private final Producer<T> producer = new Producer<T>(queue);
4
5       public void run() {
6           producer.start();
7           T data;
8           try {
9               do {
10                  data = queue.get();
11                  consume(data);
12              } while(data != null);
13          } catch (InterruptedException e) {
14          } finally {
15              producer.interrupt();
16          }
17      }
18
19      private void consume(T data) {
20          //Code omitted
21      }
22  }
```