No hand-written solutions will be accepted. Your solutions must be produced using some word processing or typesetting software such as Microsoft Word or LaTeX, and exported to the PDF format. Exception: Neatly hand-drawn figures will be accepted. Submit only one single PDF.

1 Theory [100 points]

1.1 Java memory model [7 points]

Consider the class shown in listing 1. According to what you have been told about the Java memory model, will the *reader* method ever divide by zero?

```
1  class VolatileExample {
2      int x = 0;
3      volatile boolean v = false;
4      public void writer() {
5          x = 42;
6          v = true;
7      }
8      public void reader() {
9          if (v == true) {
10             int y = 100/x;
11         }
12     }
13 }
```

Listing 1: Volatile field example from section 1.1

1.2 Sequential Consistency (1) [7 points]

For the following executions, please indicate if they are sequentially consistent. All variables are intially set to 0.

1. P1: W(x,1);
   P2: R(x,0); R(x,1);

2. P1: W(x,1);
   P2: R(x,1); R(x,0);

3.
```
P1:  W(x,1);
P2:  W(x,2);
P3:  R(x,1); R(x,2);
```

4.
```
P1:  W(x,1);
P2:  W(x,2);
P3:  R(x,2); R(x,1);
```

5.
```
P1:  W(x,1);
P2:  W(x,2);
P3:  R(x,2); R(x,1);
P4:  R(x,1); R(x,2);
```

6.
```
P1:  W(x,1); R(x,1); R(y,0);
P2:  W(y,1); R(y,1); R(x,1);
P3:  R(x,1); R(y,0);
P4:  R(y,0); R(x,0);
```

7.
```
P1:  W(x,1); R(x,1); R(y,0);
P2:  W(y,1); R(y,1); R(x,1);
P3:  R(y,1); R(x,0);
```

1.3 Sequential Consistency (2) [6 points]

P1	**P2**	**P3**
x = 1;	y = 1;	z = 1;
print(y,z);	print(x,z);	print(x,y);

All variables are stored in a memory system which offers sequential consistency. All operations, even the print statements, are atomic[1]. Are the following sequences legal outputs?

1. `001011`
2. `001111`
3. `001110`

Explain your answer by showing one possible interleaving of the instructions that might lead to the legal output. In case of an illegal output, explain why no possible interleaving exists.

[1] no operation can overlap with other operations

1.4 Consistency (1) [8 points]

Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory object. Does the converse hold? If the memory object is quiescently consistent, are the individual registers quiescently consistent? Outline a proof or give a counterexample.

1.5 Consistency (2) [7 points]

Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

1.6 Linearizability (1) [7 points]

For the following history of a shared register with the operations write(x)/void and read()/x, answer the questions below.

```
B: r.write(1)
A: r.read()
C: r.write(2)
A: r:1
B: r:void
C: r:void
B: r.read()
B: r:1
A: q.write(3)
C: r.read()
A: q:void
```

1. What is $H|B$?
2. What is $H|r$?
3. Turn H into a complete subhistory H'.
4. Is H' sequential?
5. Is H' well-formed?

6. Is H' linearizable? If yes, prove it!

7. If the first two events are swapped, is the resulting history equivalent to H?

1.7 Linearizability (2) [7 points]

Is the following history of a FIFO queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it! Is it sequentially consistent?

```
A: r.enq(x)
A: r:void
B: r.enq(y)
A: r.deq()
B: r:void
A: r:y
```

1.8 Linearizability (3) [7 points]

Is the following history of a fo queue with the operations enq(x)/void deq()/x linearizable? If yes, prove it!

```
A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y
```

1.9 Compositional Linearizability [7 points]

Proove the "only if" part of Theorem 3.6.1, reprinted below.

> H is linearizable if, and only if, for each object x, $H|x$ is linearizable.

1.10 More Histories (1) [7 points]

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

1.11 More Histories (2) [7 points]

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	Invoke `q.enq(y)`
2	Work on `q.enq(x)`	Return from `q.enq(y)`
3	Return from `q.enq(x)`	
4		Invoke `q.deq()`
5		Return `x` from `q.deq()`

1.12 More Histories (3) [7 points]

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Return from `q.enq(x)`	
2		Invoke `q.enq(y)`
3	Invoke `q.deq()`	Work on `q.enq(y)`
4	Work on `q.deq()`	Return from `q.enq(y)`
5	Return `y` from `q.deq()`	

1.13 AtomicInteger [7 points]

The `AtomicInteger` class (in the `java.util.concurrent.atomic` package) is a container for an integer value. One of its methods is `boolean compareAndSet (int expect, int update)`.

This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns `true`. Otherwise, it leaves the object's value unchanged and returns `false`. This class also provides `int get()`, which returns the object's actual value.

Consider the FIFO queue implementation shown in listing 2. It stores its items in an array `items`, which, for simplicity, we will assume has unbounded size. It has two `AtomicInteger` fields: `head` is the index of the next slot from which to remove an item, and `tail` is the index of the next slot in which to place an item. Give an example showing that this implementation is not linearizable.

1.14 Herlihy/Wing queue [9 points]

This exercise examines a queue implementation (listing 3) whose `enq()` method does not have a linearization point.

The queue stores its items in an `items` array, which for simplicity we will assume never hits `CAPACITY`. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail` and then storing the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.

The `deq()` method reads the value of `tail`, then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps `null` with the current contents, returning the first non-`null` item it finds. If all slots are `null`, the procedure is restarted.

Give an example execution showing that the linearization point for `enq()` cannot occur at line 14[2]. Give another example execution showing that the linearization point for `enq()` cannot occur at line 15. Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

[2]Hint: give an execution where two `enq()` calls are not linearized in the order they execute line 14.

```
1  class IQueue<T> {
2      AtomicInteger head = new AtomicInteger(0);
3      AtomicInteger tail = new AtomicInteger(0);
4      T[] items = (T[]) new Object[Integer.MAX_VALUE];
5      public void enq(T x) {
6          int slot;
7          do {
8              slot = tail.get();
9          } while (!tail.compareAndSet(slot, slot+1));
10         items[slot] = x;
11     }
12     public T deq() throws EmptyException {
13         T value;
14         int slot;
15         do {
16             slot = head.get();
17             value = items[slot];
18             if (value == null)
19                 throw new EmptyException();
20         } while (!head.compareAndSet(slot, slot+1));
21         return value;
22     }
23 }
```

Listing 2: `IQueue` implementation

```
1  public class HWQueue<T> {
2      AtomicReference<T>[] items;
3      AtomicInteger tail;
4      static final int CAPACITY = 1024;
5
6      public HWQueue() {
7          items = (AtomicReference<T>[])Array.
               newInstance(AtomicReference.class,
               CAPACITY);
8          for (int i = 0; i < items.length; i++) {
9              items[i] = new AtomicReference<T>(null);
10         }
11         tail = new AtomicInteger(0);
12     }
13     public void enq(T x) {
14         int i = tail.getAndIncrement();
15         items[i].set(x);
16     }
17     public T deq() {
18         while (true) {
19             int range = tail.get();
20             for (int i = 0; i < range; i++) {
21                 T value = items[i].getAndSet(null);
22                 if (value != null) {
23                     return value;
24                 }
25             }
26         }
27     }
28 }
```

Listing 3: Herlihy/Wing queue

2 Practice [Extra Credit 15 points]

Using a Linearizability Checker

A linearizability checker, as described by Jepsen Knossos, works as follows: Given a history of operations by a set of clients and some single-threaded model, attempts to show whether the history is linearizable with respect to that model.

Use the Knossos linearizability checker to evaluate the histories provided in Problems 1.6–1.8 above. To receive full-credit, you should run the checker as described in Knossos#as-a-library, and provide screenshots showing your inputs and the output of the tool. A README file has been attached to the assignment page that describes how to setup the library. Compare your answers to problems 1.6–1.8 and the checker's output; is the outcome of the checker correct? Remember, the author of the checker doesn't claim the checker to be correct and thus, you must verify the outcome by hand (which you should have already done in the Theory section).

Don't submit any code as part of this assignment. Include only screenshots in your submission PDF document.