

SYSC 3303 Sample Final Exam “Core Programming” Questions

Question 1

A canal (such as the Rideau canal) is a manmade channel that bypasses sections of a river containing rapids (shallow rocky areas with fast flowing water). Digging a deep channel around the rapids isn't enough to make the waterway navigable, because canals with fast flowing water are difficult to navigate. The solution to this problem is to build locks in the canal. A canal lock is essentially a dam that keeps the water at the top and bottom of the lock from flowing too fast. (Don't confuse a canal lock with a Java object lock. Unless otherwise specified, the word "lock" in this question refers to a canal lock.)

A canal lock has two gates - an upstream gate on the high water side and a downstream gate on the low water side. To travel from the high water side to the low water side, a boat enters the lock through the upstream gate, and then water is let out of the lock until the boat is lowered to the level of the water on the low water side. To travel from the low water side to the high water side, a boat enters the lock through the downstream gate, and then water is let into the lock until the boat is raised to the level of the water on the high water side. The water level in a lock can be raised or lowered only when both gates are closed. Boats can enter and leave the lock only when a gate is open. The upstream gate can be opened and closed only when the water level in the lock is high. The downstream gate can be opened and closed only when the water level in the lock is low.

We are developing an application simulating boats traveling downstream (i.e. entering the upstream gate and exiting the downstream gate).

Our application consists of a controller thread (which contains the main method) and multiple boat threads. The canal lock is modeled by class `CanalLock`, and is shared among the boat threads. Newly created `CanalLock` objects have a high water level, the upstream gate open, and no boats inside them.

```
CanalLock canalLock = new CanalLock();
```

When the upstream gate is open, up to four boats can enter the lock (one at a time – each takes 30 seconds to enter). Once four boats have entered, the lock can be lowered:

```
canalLock.lower();
```

Lowering the lock consists of closing the upstream gate, lowering the water level, and opening the downstream gate. This whole operation takes 5 minutes.

Once the downstream gate is opened, the boats can exit (one at a time – each takes 30 seconds to exit) and proceed on their way.

Once the boats have left the lock, the lock can be raised:

```
canalLock.raise();
```

Raising the lock consists of closing the downstream gate, raising the water level, and opening the upstream gate. Again, this whole operation takes 5 minutes.

Note that the controller thread invokes `canal.lower()` and `canal.raise()` as necessary (the controller's code is not shown here).

Boats are modelled by class `Boat`. Here is a partial implementation of the class. Notice that this class implements the `Runnable` interface. As such, it is easy to use an instance of this class to create multiple boat threads.

```
public class Boat implements Runnable
{
    private CanalLock canalLock; // The lock the boat will go through.

    public Boat(CanalLock canalLock) {
        this.canalLock = canalLock;
    }

    public void run() {
        ...
        canalLock.pass(); // go through the lock
        ...
    }
}
```

To pass through the lock, a boat thread invokes the lock's `pass()` method. If the upstream gate is open and there are fewer than four boats in the lock, the boat enters the lock; otherwise, it must wait. Boats enter the lock one at a time and take 30 seconds to enter the lock. As soon as the downstream gate has opened, the boats exit one at a time and take 30 seconds each to exit the lock. Once a boat has exited the lock, the `pass()` method returns.

Here is an *incomplete* implementation of the `CanalLock` class:

```
public class CanalLock
{
    // instance variables go here
    CanalLock() {...}
    lower() {...}
    raise() {...}
    pass() {...}
}
```

Write the code for the complete `CanalLock` class in your answer booklet. Define all needed instance variables, as well as the complete signature for each operation (you may add on to what is provided here). You may also define constructors and methods other than the ones shown here. Your methods must implement whatever mutual exclusion and condition synchronization is required. You do not need to write any javadoc comments for this class, but informal comments will assist in understanding your code.

Question 2

In this question you are to write a `Barrier` class with a constructor and one other method, `checkpoint`:

```
public class Barrier
{
    public Barrier(int n) {...}
    public boolean checkpoint() {...}
}
```

After a `Barrier` object has been created, every time a thread invokes `checkpoint()` on that `Barrier` object it will pause until `n` threads have invoked `checkpoint()` at which time all `n` threads will continue. The `checkpoint()` method returns `true` if the thread successfully synchronized, and `false` otherwise (e.g. this thread was the `n+1` th thread).

- a) Write the `Barrier` class. Ensure each `Barrier` object is left in a state so that it can be reused.
- b) Draw a class diagram for the `Barrier` class.

Question 3

In this question you are going to write code relating to a **drawbridge**. Our drawbridges are road bridges over rivers. While small boats can pass under a bridge while the bridge is in use (by cars, pedestrians, etc.), tall boats cannot go past a bridge unless it is open. When a bridge is open, all vehicular (cars, etc.) and pedestrian traffic must stop. Our drawbridges are all very wide, so both cars and boats may proceed in both directions at once (two-way traffic). We will assume that all our drawbridges have the same `CLEARANCE`. Boats with height less than `CLEARANCE` may proceed at any time. Boats with height greater than or equal to `CLEARANCE` may proceed only when the drawbridge is open. All our drawbridges will start in the closed state. In case you are unfamiliar with a drawbridge, you will find pictures and explanations in the appendix.

Our real-time system that models a drawbridge has the following threads:

- **Controller:** The controller thread is in charge of opening and closing the bridge. There is just one controller thread for each drawbridge. The code in the `Controller`'s `run()` method will invoke `Drawbridge` methods `open()` and `close()` to open and close the bridge, respectively.
- **Car:** There is one thread for each car. Cars can cross the bridge from east to west, or from west to east. The code in the `Car` class `run()` method will invoke `Drawbridge` methods `goWest()` and `goEast()` so that the car can cross the bridge from east to west, or from west to east.
- **Boat:** There is one thread for each boat. Boats can pass the bridge from north to south, or from south to north. The code in the `Boat` class `run()` method will invoke `Drawbridge` methods `goSouth()` and `goNorth()` so that the boat can go past the bridge from north to south and from south to north, respectively.

You do **not** need to write the code for the `Controller`, `Car` or `Boat` classes or the main program that creates the threads and ensures that they are all sharing the same `Drawbridge` object.

You are going to write some of the code for the `Drawbridge` class. Note that your code must ensure that each instance of the `Drawbridge` class prevents cars from being on the bridge when it is opening, open or closing, and prevents tall boats from being under the bridge when it is closing, closed or opening. At the same time, ensure that your critical sections are as short as possible.

Hint: Think carefully about whether each “sleep” method needs to be in a critical section or not.

Here is an outline of the class:

```
public class Drawbridge {

    // all our drawbridges have a clearance of 10m. (Boats 10m or
    // over can proceed only when the drawbridge is open.)
    public final static int CLEARANCE = 10;
    // member variables go here

    /**
     * open: Opens the drawbridge. (If already open does nothing.)
     * This method has no arguments and returns void.
```

```

* To simulate the time it takes to (a) wait until all cars
* currently crossing the bridge have done so, and (b) then to
* open the bridge, this method sleeps for 1 minute before it
* returns. No cars or tall boats may proceed while the bridge
* is in the process of opening.
*/

// close: Closes the drawbridge.  (Similar to open().)

/**
* goWest: A car goes over the bridge from east to west.
* This method has no arguments and returns void.
* This method should not return until the car has safely gone
* over the bridge (while the bridge is closed!).
* To simulate the time it takes to cross the bridge, this
* method sleeps for 10 seconds before it returns.
* (Note that cars and small boats may all proceed in parallel.
* In other words, one car does not have to wait until another
* has finished crossing the bridge before it can start.)
*/

// goEast: A car goes over the bridge from west to east.
// (Similar to goWest().)

/**
* goSouth: A boat goes under the bridge from north to south.
* This method has one argument, the height of the boat (in
* meters) and returns void.
* This method should not return until the boat has safely gone
* past the bridge.  Small boats may go under the bridge at
* any time.  Tall boats may only proceed when the bridge is
* open. To simulate the time it takes to pass the
* bridge (for any size boat), this method sleeps for 30
* seconds before it returns.
* Note also that when boats are crossing under the bridge,
* they may proceed in parallel.  In other words one boat does
* not have to finish going under the bridge before another can
* start.  (And as mentioned in goWest above, small boats and
* cars can also proceed in parallel.)
*/

// goNorth: A boat goes under the bridge from south to north.
// (Similar to goSouth().)
}

```

- a) In your answer booklet, write all of class Drawbridge's member variables and full details for methods `open()`, `goWest()` and `goSouth()`, plus any other methods that you choose to add to the class to help you implement `open`, `goWest` and `goSouth`. Do **not**

write code for `close()`, `goEast()` and `goNorth()`. (13 marks)

- b) Reverse engineer the complete `Drawbridge` class into a UML class diagram. Ensure that you show the visibility, type and initial value of each attribute, along with the visibility, signature and synchronization property of each operation. **All** the operations, including the three that you did not have to write in part a), should be shown in your UML diagram. (5 marks)
- c) Of the five levels of thread safety discussed in the lectures, identify the appropriate level for class `Drawbridge` and briefly explain your answer. (2 marks)

Question 4

In this question you are to write the code for another class closely related to the `Box` class discussed in the lectures. (Note that you can find a copy of our original `Box` class in the appendix.)

`CoordBox1` is a version of `Box` where each `put` must wait for a `get`. In other words, the `put` and `get` methods return at (almost) the same time. In the original `Box` class, if a consumer invokes `get` while the box instance is empty, `get` will wait until a producer puts an item in the box, and then both return. We still want this behaviour in `CoordBox1`. With the original `Box` class, if a producer invokes `put` and no consumer is waiting, the producer would fill the box and return. This will be different in our new class. `CoordBox1` will be written so that the `put` method does not return until a consumer arrives to consume (`get`) the item.

After a producer/consumer pair has finished with the `CoordBox1` object, other producer/consumer pairs may use it. Thus, when the first pair finish, you must ensure that the `CoordBox1` object state is such that the next pair can successfully proceed if they invoke `put` and `get` (or `get` and `put`).

We want `put` and `get` to work properly in all circumstances.

- a) The simplest way to ensure that `put` and `get` work properly in all circumstances is as follows: If a second (or third, fourth, etc.) consumer invokes `get` before the first producer invokes `put`, then the second (third, fourth, etc.) consumer all immediately return `null`. This indicates failure, as a successful consumer returns a non-`null` `Object`. Similarly, if a second (third, fourth, etc.) producer invokes `put` before the first consumer invokes `get`, then the second (third, fourth, etc.) producer all immediately return `false` (note that in this part of the question the `put` method no longer returns `void`). Again, this indicates failure, as a successful producer returns `true`.

In your answer booklet, write the complete `CoordBox1` class, including member variables and both the `put` and `get` methods. (You may also include other methods, if desired.)
Hint: You will need to add to the list of member variables. (10 marks)

- b) Now we want to change our class so that `put` and `get` never return immediately, but instead any extra producers and consumers wait until they can use the box instance. In this version, `CoordBox2`, `put` will once again return `void`. In your answer booklet, write the new class, `CoordBox2`, or, if you prefer, just the portions that have changed from part a). (5 marks)

For both parts, ensure that you include adequate comments to explain your code.

Appendix: Reference Material

(You may remove the appendix from the rest of the exam paper, but all pages must be handed in at the end of the exam.)

The Box Class

Here is the `Box` class referred to in question 4. If you wish, you may refer to the line numbers below in your answers to avoid copying identical code into your answer booklet. For example, “include `Box` lines 3-6 here”.

```
1  public class Box
2  {
3      // The contents of the box.
4      private Object contents = null;

5      // If true, there is room for an object in the box.
6      private boolean empty = true;

7      // put: Puts an object (the argument obj) into the box.
8      // Returns when the object argument has been put into the box.
9      public synchronized void put(Object obj) {
10         while (!empty) {
11             try {
12                 wait();
13             } catch (InterruptedException e) { return; }
14         }
15         contents = obj;
16         empty = false;
17         notifyAll();
18     }

19     // get: Removes and returns the object currently in the box.
20     // Returns when there's an object in the box to remove.
21     public Object get() {
22         while (empty) {
23             try {
24                 wait();
25             } catch (InterruptedException e) { return null; }
26         }
27         Object obj = contents;
28         empty = true;
29         notifyAll();
30         return obj;
31     }
32 }
```

Java API Reference

Class Object

// `equals(Object)` returns true if this object is equal to `obj`.
`public boolean equals(Object obj);`

// `hashCode()` returns the hash value of this object.
`public int hashCode();`

// `clone()` returns a new object that is a copy (a duplicate) of this object.
`protected Object clone();`


```

// toString() returns a string representation of this object.
public String toString();

// wait() causes the current thread to wait until another thread invokes the
// notify() method or the notifyAll() method for this object. The interrupted
// status of the current thread is cleared if this method throws an
// InterruptedException.
public final void wait() throws InterruptedException

// notify() wakes up a single thread that is waiting on this object's lock.
public final void notify();

// notifyAll() wakes up all threads that are waiting on this object's lock.
public final void notifyAll();

```

Class Thread

```

// currentThread() returns a reference to the currently executing thread object.
public static Thread currentThread();

// Returns this thread's priority.
public final int getPriority();

// interrupt() interrupts this thread.
public void interrupt();

// interrupted() returns true if the current thread has been interrupted; false
// otherwise. The interrupted status of the thread is cleared by this method.
public static boolean interrupted();

// isInterrupted() returns true if this thread has been interrupted; false
// otherwise. The interrupted status of the thread is unaffected by this method.
public boolean isInterrupted();

// Causes this thread to begin execution; the Java Virtual Machine calls the run
// method of this thread.
public void start();

// Changes the priority of this thread to the specified newPriority.
public final void setPriority(int newPriority);

// Causes the currently executing thread to sleep (temporarily cease execution)
// for the specified number of milliseconds. The thread does not lose ownership
// of any monitors. Parameter millis is the length of time to sleep in
// milliseconds. The interrupted status of the current thread is cleared if this
// method throws an InterruptedException.
public static void sleep(long millis) throws InterruptedException;

// Causes the currently executing thread object to temporarily pause and allow
// other threads to execute.
public static void yield();

```

Drawbridge Pictures and Terminology

Here are two photos of a drawbridge to help you understand what you are modeling in question 3.



Above is a drawbridge in the “closed” state. Note that for a drawbridge, “closed” means that the bridge is in the lowered state (no span of the bridge is raised). “Closed” does **not** mean that it is closed to car traffic. When the drawbridge is closed, cars can proceed in both directions (two-way car traffic) across the bridge. When the drawbridge is closed, only small boats (those whose height is less than the bridge clearance) may proceed. Small boats may proceed in both directions (two-way boat traffic). Tall boats (with height greater than or equal to the bridge clearance) must wait until the bridge is open to proceed.



Above is the same drawbridge in the “open” state. For a drawbridge, “open” means that parts of the bridge have been raised into the air so that the road cannot be used. “Open” does **not** mean that it is open to car traffic. When the drawbridge is open, all car traffic in both directions is stopped. When the drawbridge is open, all boats may proceed and can go past the bridge in either direction (two-way boat traffic).

A “real-life” drawbridge has lots of safety features built in. Before the bridge can be opened, there are car traffic lights that change to red, bells that ring, barricades that come down to prevent vehicles from accessing the bridge, and, finally, a visual check by the person controlling the bridge that there are no stalled cars on the bridge. Before the bridge can be closed, there are boat traffic lights that are set to red, and a visual inspection for disabled boats under the bridge is performed. For purposes of question 1, we assume that no cars or boats will break down. You do not need to model any of the safety features (lights, barricades, etc.), as careful use the drawbridge “lock” (in your code) will prevent cars and boats from proceeding when they should not.