Concurrent Systems and Applications

Example Questions, Section 1

Michaelmas 2001

tim.harris@cl.cam.ac.uk

This question sheet is intended to accompany the first half of the Michaelmas 2001 Part 1b course in *Concurrent Systems and Applications*. It includes a number of new questions along with parts of examination questions that remain relevant from the *Concurrent Systems* and *Further Java* courses. They are intended to cover a range of topics, including practical programming exercises and questions for which a written answer is more appropriate.

A further question sheet will be distributed for the second half of the course, covering topics in inter-process communication and transactions.

1. Describe the facilities in Java for defining classes and for combining them through composition, inheritance and interfaces. Explain with a worked example how they support the principle of encapsulation in an object-oriented language.

2. What is meant by *reflection* or *introspection* in Java? Give an example of how and why these facilities may be useful.

3. Describe the facilities in Java for restricting concurrent access to critical regions. Explain how shared data can be protected through the use of objects.

4. Describe the model for handling graphical output and interactive input in the Abstract Windowing Toolkit (AWT) for Java. Your answer should cover the use of

   - hierarchies of classes,
   - overriding methods,
   - interfaces,
   - inner classes,
   - spatial hierarchy.

5. Describe the differences and similarities between abstract classes and interfaces in Java. How would you select which kind of definition to use?

6. An enthusiast for programming with *closures* proposes a new language, D$^\flat$, extending Java so that the following method definition would be valid:

```
Closure myCounter (int start) {
  int counter = start;
  return {
    System.out.println (counter ++);
  }
}
```

   The programmer intends that no output would be produced when this method is executed, but that it would return an object of a new built-in type, `Closure`:

```
interface Closure {
  void apply();
}
```

   Invoking `apply()` on the object returned by `myCounter` will cause successive values to be printed. By using an *inner class* definition, show how this example could be re-written as a valid Java program.

7. The *Memento* design pattern can be used when designing an 'undo' facility for part of a program – e.g. in a graphical drawing environment to allow parts of the drawing to be restored to an earlier state.

   In this pattern classes occupy one of three rôles: the *originator* is the class whose state must be saved and restored (e.g. a component in the graphical design), instances of the *memento* class represent snapshots of the saved state that are created and restored under the control of the *caretaker* class (typically the 'main' part of the application acting in response to user requests to undo operations). The originator is responsible for instantiating the memento class, for saving its state into such an instance and for restoring its state from an instance.

   Suggest how the classes in the Memento pattern may be organized:

   - As a UML class diagram.
   - In terms of example Java code. Justify your answer's use (or otherwise) of access modifiers, inheritance, interfaces and abstract or nested classes.

   Does the availability of reflection and serialization facilities in Java obviate the need for the Memento pattern?

8. A common programming mistake is to try to define a class to have more than one superclass. For example a naïve programmer may write

```
class AutoTree extends Thread, BinaryTree {
  ...
}
```

when defining a new kind of data structure which uses an additional thread to keep the tree balanced. Describe *three* ways in which this problem can be resolved to produce (one or more) valid class definitions. State, with a brief justification, which you would use here.

9. Describe how mutual-exclusion locks provided by the `synchronized` keyword can be used to control access to shared data structures. In particular you should be clear about the behaviour of concurrent invocations of different synchronized methods on the same object, or of the same synchronized method on different objects.

10. Describe what a *race condition* is, with the aid of example code.

    "*A Java class is safe for use by multiple threads if all of its methods are synchronized*". To what extent do you agree with this statement?

11. *Processor affinity*, *take scheduling* and *gang scheduling* are three techniques used within multiprocessor operating systems.

    - Briefly describe the operation of each.
    - Which problem does the processor affinity technique seek to overcome?
    - What problem does the processor affinity technique suffer from, and how could this problem be overcome?
    - In which circumstances is a gang scheduling approach most appropriate?

12. A software module controls a car park of known capacity. Calls to the module's procedures *enter*() and *exit*() are triggered when cars enter and leave via the barriers.

    Sketch the implementation of the *enter* and *exit* procedures in Java

    - using the ordinary concurrency-control operations of that language,
    - if the programming language in which the module is written provides only semaphores.

13. Consider the following class definition:

```
class Example implements Runnable {
  public static Object o = new Object();
  int count = 0;

  public void run() {
    while (true) {
      synchronized (o) {
        count ++;
      }
    }
  }
}
```

(a) Show how to start two threads, each executing this `run` method on a separate instance of `Example`.

(b) When this program is executed, only one of the `count` fields is found to increment, even though threads are scheduled preemptively. Why might this be?

(c) Define a new class `FairLock`. Each instance should support two methods, `lock` and `unlock`, which acquire and release a mutual exclusion lock such that calls to `unlock` never block the caller, but will allow the longest-waiting blocked thread to acquire the lock. The lock should be *recursive*, meaning that the thread holding the lock may make multiple calls to `lock` without blocking. The lock is released only when a matched number of `unlock` operations have been made.

(You may wish to make use of the fact the `Thread.currentThread()` returns the instance of `Thread` that is currently executing.)

14. Five philosophers spend their time alternately *thinking* and *eating*. The share a common, circular table, surrounded by five chairs, each belonging to one philosopher. In the centre of the table is a bowl of spaghetti and the table is laid with five forks, one between each pair of adjacent chairs. Thinking philosophers don't interact with their colleagues. From time to time philosophers may get hungry and try to pick up the two closest forks. A philosopher may only pick up one fork at a time. It is an axiom of philosophic thought that one is only allowed to eat with the aid of two forks and that, of course, both forks are put down while thinking.

Simulate this problem in Java using a separate thread for each philosopher.

Does your simulation illustrate either deadlock or livelock? If so then what changes could you make to reduce the risks?

15. Suppose that, instead of using mutual exclusion locks, a programmer attempts to avoid concurrent access to a data structure by manipulating the running thread's scheduling priority in a class extending `java.lang.Thread`:

```
void enterCriticalRegion {
  oldPriority = getPriority ();
  setPriority (Thread.MAX_PRIORITY);
}

void exitCriticalRegion {
  setPriority (oldPriority);
}
```

What assumptions are needed to guarantee that this is safe?

16. One method for avoiding deadlock is for a thread to simultaneously acquire all of the locks it needs for an operation. However, Java's `synchronized` keyword can only acquire or release a single lock at a time.

Sketch the design of a class `LockManager` that implements the `LockManagerIfc` interface (below) so that the `doWithLocks` operation appears to:

(a) acquire locks on all of the objects in the array `o`,

(b) invoke `op.do(arg)` keeping the result of that method as the eventual result of `doWithLocks`,

(c) release all of the locks initially acquired.

You should assume that your class is the only part of the application in which locks are acquired or released.

```
interface Operation {
  Object do(Object arg);
}

interface LockManagerIfc {
  Object doWithLocks (Object o[],
                      Operation op,
                      Object arg);
}
```

17. The built-in facilities for restricting concurrency in Java allow only one thread at a time to be executing within the critical region. A different approach is to distinguish *shared* and *exclusive* access to a critical region: any number of *readers* may share access at the same time, but only one *writer* may acquire exclusive access (excluding any readers while it does so).

    (a) Specify a `MultiSync` class in Java with methods to acquire and release both read and write access, and sketch its implementation.

    (b) Derive a `MultiBuffer` class that extends `MultiSync` with methods to store and read a data field, ensuring that any locks are released when a waiting thread is interrupted. Your example may use a simple data field such as an integer but you should explain why such an elaborate scheme of concurrency control is unnecessary in this case.

18. What are the key issues with scheduling for shared-memory multiprocessors?

19. Outline the implementation of

    - semaphores
    - monitors

    Make clear what facilities you are assuming of the processor and operating system.

20. The runtime system of a concurrent programming language includes semaphore management and user-thread management modules. The design assumption was that the operating system on which the concurrent programs are to run supports only one thread per address space. Outline the functionality and interaction of semaphore and user-thread management.

    The runtime system is to be ported to run on an operating system which supports multi-threaded processes and symmetric multiprocessing.

    - Describe how the two modules must be changed in order to support
        - one kernel-thread per user-thread
        - scheduler activations

      State any assumption you make about the functionality of the operating system.

    - Discuss the problem of concurrent execution of the semaphore management module and how this concurrency might be controlled.