The University of Nottingham

SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

A LEVEL 2 MODULE, SPRING 2004–2005

CONCEPTS OF CONCURRENCY

Time allowed TWO hours

Candidates must NOT start writing their answers until told to do so.

Candidates should attempt QUESTION ONE and THREE other questions. *Marks available for sections of questions are shown in brackets in the right-hand margin*

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a dictionary to translate between that language and English provided that neither language is the subject of this examination.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

DO NOT turn examination paper over until instructed to do so

1 **(Compulsory)** Select ONE answer for each section:

(a) In a multiprocessing implementation of concurrency, atomic actions are those which
- (i) lock memory;
- (ii) can't be interrupted;
- (iii) can't be interrupted or lock memory;
- (iv) can't be interrupted and lock memory;
- (v) execute concurrently. [3]

(b) Mutual exclusion holds between:
- (i) shared variables;
- (ii) concurrent processes;
- (iii) critical sections in the same class;
- (iv) critical sections in different classes;
- (v) atomic actions. [3]

(c) Absence of Unnecessary Delay means that:
- (i) processes wait only when it is not their turn to enter their critical section;
- (ii) processes wait only when it is their turn to enter their critical section;
- (iii) processes wait only when another process is in its critical section;
- (iv) processes wait only when another process is in its critical section or is trying to enter;
- (v) no processes have to wait; [3]

(d) Synchronous message passing:
- (i) always causes the sending process to block;
- (ii) always causes the receiving process to block;
- (iii) always causes whichever process goes first to block;
- (iv) always causes both the sending and receiving process to block;
- (v) never blocks. [3]

(e) In RPC:
- (i) all calls are processed by the same server process and mutual exclusion is implicit;
- (ii) a new server process is created for each call and mutual exclusion is implicit;
- (iii) a new server process is created for each call and mutual exclusion must be explicitly programmed;
- (iv) a new server process may be created for each call and mutual exclusion must be explicitly programmed;
- (v) shared memory is not used and so there is no problem of mutual exclusion. [3]

(f) In Java, invoking `notify()` in a `synchronized` method of an object `o`:

(i) wakes up a thread in the wait set of `o`;

(ii) wakes up the first thread in the wait set of `o`;

(iii) wakes up the highest priority thread in the wait set of `o`;

(iv) wakes up all the threads in the wait set of `o`;

(v) blocks if the wait set of of `o` is empty. [3]

(g) Let c_1 and c_2 be propositional variables, where c_1 means that 'process 1 is in its critical section' and c_2 means that 'process 2 is in its critical section'. What does the CTL formula $AG(EFc_1 \wedge EFc_2)$ mean:

(i) it is always the case that one of the processes is in its critical section;

(ii) mutual exclusion is always violated;

(iii) each process will enter its critical section exactly once;

(iv) each process will enter its critical section infinitely often;

(v) at least one of the processes will enter its critical section infinitely often. [3]

(h) The propositional variable c means that a process is in its critical section and r means that it is in its remainder. How would you express in CTL that the process never terminates in its critical section:

(i) $AG(c \rightarrow AFr)$

(ii) $AG(c \rightarrow Xr)$

(iii) $AG(c \rightarrow EFr)$

(iv) $AG(c \rightarrow AGr)$

(v) $AG(c \rightarrow EGr)$ [4]

2 (a) Explain what is meant by the term *interference* when applied to concurrent programs and how problems of interference can be overcome using mutual exclusion protocols. [7]

(b) Consider the critical section protocol for two processes shown below:

```
// Shared variables
int c1 = c2 = 1;
```

```
// Process 1
init1;
while(true) {
    // entry protocol
    c1 = 0;
    while (c2 == 0) {
        c1 = 1;
        // sleep briefly ...
        c1 = 0;
    }
    crit1;
    // exit protocol
    c1 = 1;
    rem1;
}
```

```
// Process 2
init2;
while(true) {
    // entry protocol
    c2 = 0;
    while (c1 == 0) {
        c2 = 1;
        // sleep brielfy ...
        c2 = 0;
    }
    crit2;
    // exit protocol
    c2 = 1;
    rem2;
}
```

Does the protocol satisfy the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry? For each property, either explain clearly why the protocol satisfies the property or give an example trace that results in the property being violated. [12]

(c) Explain why it is important to prove the properties of concurrent programs and briefly describe two approaches to concurrent program verification. [6]

3 (a) Semaphores and Monitors are two higher-level synchronisation primitives. Explain the differences between semaphores and monitors and how mutual exclusion and condition synchronisation is achieved with each. [8]

(b) A programming language provides support for monitors but not for semaphores. Develop a monitor implementation (in pseudo code) of a general semaphore. Assume that the monitor uses a *signal and wait* signalling discipline. Explain how your solution works and why it is correct. [12]

(c) Explain the difference between the *signal and continue* and *signal and wait* signalling disciplines used in monitors. Modify your solution to part (b) for a monitor implementation that uses a *signal and continue* signalling discipline. Explain your answer. [5]

4 (a) What is a Thread in Java? [2]

(b) Briefly outline the major stages in the life cycle of a thread and explain how the transitions between stages occur. [6]

(c) A Java application maintains a queue of deferred tasks. Tasks are represented by instances of classes which implement the `Task` interface, which has a single method `execute()`. Develop a Java `DeferredTaskQueue` class with the following public methods:

- `void DeferredTaskQueue()`: (constructor) creates a new `DeferredTaskQueue` object;
- `void append(Task t, long mills)`: add the deferred task `t` to the queue. When `mills` milliseconds have elapsed from the time the task is added to the queue, the task is dequeued and executed; and
- `int length()`: returns the number of deferred tasks currently in the queue.

Your implementation should allow safe concurrent access to the deferred task queue by multiple threads, and should execute the deferred tasks at the specified time (insofar as this is possible given Java's scheduling guarantees). Explain your answer. [17]

5 (a) Explain how Remote Procedure Call works and its relationship to message passing in distributed processing implementations of concurrency. [5]

(b) A distributed Java application consists of a number of producers and consumers which communicate using a bounded buffer server. Clients make remote method calls to append and remove `Item` objects (which are assumed to be serializable) from the buffer. Using `java.rmi`, write a Java `RemoteBoundedBufferServer` class that implements a bounded buffer server. Your implementation should have a (public) constructor `RemoteBoundedBufferServer(int n)` which creates a new RemoteBoundedBufferServer with `n` slots and two public methods: `append(Item x)` which puts the `Item x` in the next free slot in the buffer and `Item remove()` which removes the next unconsumed item from the buffer. If there are no free slots, `append` waits until a slot is free and if there are no unconsumed items `remove` waits until a slot has been filled. Explain your answer. [15]

(c) Explain the steps in deploying the bounded buffer server. [5]

6 Write a well structured and coherent essay on the *Java Memory Model* and state its implications for the implementation of mutual exclusion protocols which use busy waiting. Illustrate your answer with a solution to the mutual exclusion problem that uses busy waiting. [25]