The University of Nottingham

SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

A LEVEL 2 MODULE, SPRING 2003–2004

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) Atomic actions:
- (i) always lock memory;
- (ii) can't be interrupted;
- (iii) always cause a process switch;
- (iv) can't be interleaved;
- (v) are the same on all computers. [3]

(b) Critical sections:
- (i) never run concurrently;
- (ii) ensure that programs never deadlock;
- (iii) must not be interrupted;
- (iv) access shared data;
- (v) are not relevant in a distributed processing implementation of concurrency. [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 another process is in its critical section;
- (iii) processes wait only when another process is in its critical section or is trying to enter;
- (iv) no processes outside their critical section have to wait.
- (v) no processes have to wait; [3]

(d) Semaphores are preferable to Dekker's algorithm for synchronising communication between concurrent processes because they:
- (i) avoid suspending processes;
- (ii) ensure that programs never deadlock;
- (iii) ensure eventual entry;
- (iv) are a lower-level programming construct;
- (v) avoid busy-waiting loops. [3]

(e) Within a monitor, calling the `wait` monitor operation:
- (i) puts the calling process on the entry queue;
- (ii) puts the calling process on a delay queue;
- (iii) moves the calling process from the entry queue to a delay queue;
- (iv) moves the calling process from a delay queue to the entry queue;
- (v) depends on the signalling discipline. [3]

(f) In a distributed processing implementation of concurrency using asymmetrical direct naming:
- (i) a send operation sends messages to all processes;
- (ii) a receive operation receives messages from all processes;
- (iii) a send operation sends messages to all channels;
- (iv) a receive operation recieves messages from all channels;

(v) send operations send messages to processes and receive operations recieve messages from channels. [3]

(g) In Java, invoking a `synchronised` method `foo()` of object `o`:

(i) prevents execution of other synchronized methods of `o` by the same thread;

(ii) prevents execution of other methods of `o` by the same thread;

(iii) prevents execution of all synchronized methods of `o` by other threads;

(iv) prevents execution of all methods of `o` by other threads;

(v) prevents another thread invoking `foo` on other objects. [3]

(h) In the mutual exclusion protocol shown below, which properties are always *satisfied*:

```
// Shared variables
boolean lock = false;
```

```
// Process 1

init1;
while(true) {
    // entry protocol
    while (lock) {};
    lock = true;
    crit1;
    // exit protocol
    lock = false;
    rem1;
}
```

```
// Process 2

init2;
while(true) {
    // entry protocol
    while (lock) {};
    lock = true;
    crit2;
    // exit protocol
    lock = false;
    rem2;
}
```

(i) mutual exclusion and deadlock;

(ii) deadlock and absence of unnecessary delay;

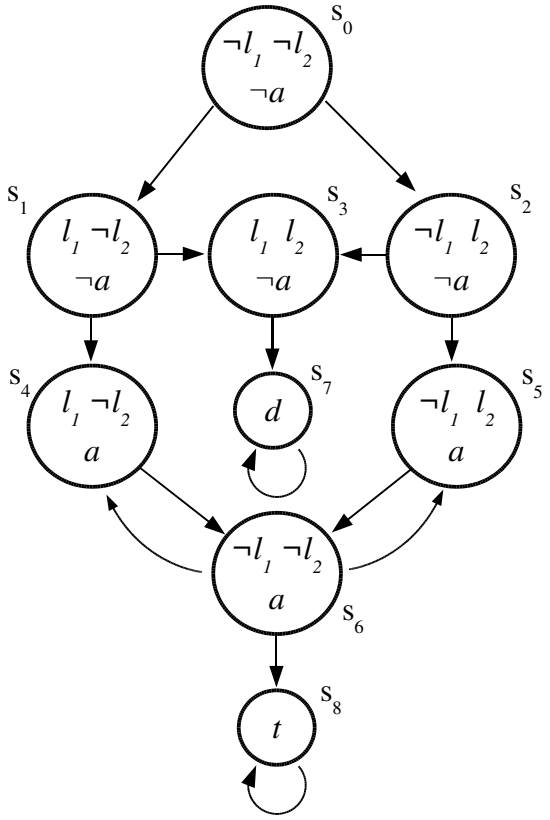
(iii) deadlock and eventual entry;

(iv) deadlock, absence of unnecessary delay and eventual entry;

(v) mutual exclusion, deadlock, absence of unnecessary delay and eventual entry. [4]

2 (a) What is meant by the terms *safety property* and *liveness property* when applied to concurrent programs? [5]

(b) Consider the state transition diagram below:



The diagram represents the state transitions of two processes in a concurrent program which assign values to a shared variable v. Unfortunately, the concurrent program is incorrect, and v is updated by critical sections in each process which are in different classes, each of which uses a different lock l1 and l2. l_1, l_2, a, d and t are propositional variables, where l_1 means 'process 1 has acquired the lock l1', l_2 means 'process 2 has acquired the lock l2', a means 'a process has assigned to the variable v', d means 'the processes are deadlocked', and t means 'the processes have terminated'. (To simplify the diagram, the propositions d and t are only shown in those states in which they are true. You may assume that a is false in s_7 and true in s_8.)

(i) Express in CTL: "if either process has performed an assignment, the system is guaranteed not to deadlock". Using CTL truth definitions, demonstrate that the formula is true at s_0. [10]

(ii) Express in CTL: "one of the processes is guaranteed to perform an assignment". Using CTL truth defintions, demonstrate that this formula is false at s_0. [10]

3 (a) Define the notion of a *semaphore* and explain the behaviour of the primitive operations provided on semaphores. [5]

(b) Define the *Readers and Writers* problem for concurrent systems. Illustrate your answer with an example. [5]

(c) *Using semaphores*, devise a solution to the Readers and Writers problem for a shared resource *file*. Explain your solution with reference to the example given in your answer to part (b), and say whether it is fair. [15]

4 (a) Define the notion of a *monitor* and explain how synchronisation within a monitor is achieved. [6]

(b) A concurrent program consists of multiple producer processes and a single consumer processes. Each producer process places a single item in a shared queue. The consumer process takes items from the queue in batches of size k. If there are no free slots in the queue, a producer has to wait. If there are less than k items in the queue, the consumer has to wait. The processes communicate via a `BoundedBuffer` monitor. The monitor has two public monitor procedures:

- `void addItem(Object o)`: which adds a data item to the next free slot in the buffer—if there are no free slots it waits until a slot is free; and
- `Object[] removeBatch()`: removes the next k unconsumed items from the buffer and returns them in an array—if there are not at least k items in the buffer, it waits until k slots are filled.

Develop an implementation (in pseudo code) of the `BoundedBuffer` monitor which minimises the number of context switches. Assume that the buffer is of length n, items are consumed in batches of size k ($k < n$) and that monitor uses a *signal and continue* signalling discipline. Explain how your solution works and state the synchronisation conditions it satisfies. [15]

(c) Explain the difference between the *signal and continue* and *signal and wait* signalling disciplines used in monitors. What changes would be necessary to your answer to part (b) above if the monitor used signal and wait? [4]

5 (a) A Java program simulates stock levels of different types of widgets in a warehouse. Deliveries and orders are represented by threads which increase and decrease the stock levels for a particular type of widget respectively. The number of widgets of each type should never be less than 0 or greater than the maximum stock level (which may be different for different kinds of widgets). A delivery must block until the maximum stock level for the widget type being delivered would not be exceeded. Similarly, an order must block if stock level for the type of widget being ordered would be non-negative after the order is processed.

Stock levels are represented by instances a `StockLevel` class which has four public methods:

- `void StockLevel(String name, long maxStock)`: (constructor) creates a new `StockLevel` object to represent the number of widgets of type `name` with maximum stock level `maxStock`;
- `void increase(long n)`: adds `n` to the stock level for this type of widget;
- `void decrease(long n)`: decreases the stock level for this type of widget by `n`; and
- `long inStock()`: returns the number of widgets of this type currently in stock.

Develop a Java `StockLevel` class which allows safe concurrent access to `StockLevel` objects. Your implementation should not throw any checked exceptions and should behave reasonably if threads are interrupted during execution of the program. Explain your answer. [15]

(b) What changes would be necessary to allow remote access to the `StockLevel` class using Java's RMI? Illustrate your answer using your solution to part (a) above. [10]

6 Write a well structured and coherent essay on the *mutual exclusion* problem for concurrent systems that communicate using shared memory. Your essay should define the problem and explain how coarse-grain atomic actions can be implemented using only standard instructions. Illustrate your answer with a solution to the mutual exclusion problem that uses only standard instructions, and state any limitations of your solution. [25]