The University of Nottingham

SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

A LEVEL 2 MODULE, SPRING SEMESTER 2002–2003

CONCEPTS OF CONCURRENCY
(Course G52CON)

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 may be used.

DO NOT turn your examination paper over until instructed to do so

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

(a). In a multiprocessing implementation of concurrency, atomic actions are those which:

(i) execute concurrently;
(ii) load and store registers;
(iii) can't be interrupted;
(iv) can't be interleaved;
(v) lock memory. (3)

(b). Critical sections in different classes:

(i) must always be executed with mutual exclusion;
(ii) are necessary to ensure condition synchronisation;
(iii) involve updating a shared resource;
(iv) can't be interleaved;
(v) may run concurrently. (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). The *signal and wait* signalling discipline:

(i) causes the signalling process to wait;
(ii) causes the signalled process to wait;
(iii) ensures the signalled condition is be true when the waiting process resumes;
(iv) indicates the signalled condition may be true when the waiting process resumes;
(v) gives highest priority to processes blocked by mutual exclusion; (3)

(e). The ***P*** operation on a semaphore:

(i) always decrements a counter;
(ii) may suspend the calling procedure;
(iii) always suspends some procedure;
(iv) can only be executed once by any procedure;
(v) sometimes has no effect. (3)

(f). Asynchronous message passing:

(i) may cause the sender to block;

(ii) may cause the receiver to block;

(iii) always causes either the sender or receiver to block;

(iv) always causes both the sender and the receiver to block;

(v) never blocks. (3)

(g). The following unsynchronized block is executed by two Java threads. What are the possible values for `x` and `y` when the threads exit:

```
int x = 0;
double y = 100;

x = x + 1;
y = y - 1;
```

(i) `x` 0 or 1, `y` 100 or 99;

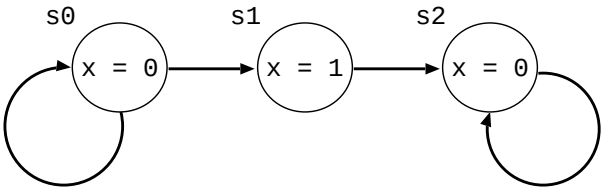
(ii) `x` 1 or 2, `y` 99 or 98;

(iii) `x` 1 or 2, `y` 100 or 99;

(iv) `x` 1 or 2, `y` undefined;

(v) the values of both `x` and `y` are undefined. (3)

(h). Which of the CTL formulas below does the following labelled transition system satisfy?



(i) $AF(x = 1)$

(ii) $AG(x = 0)$

(iii) $EF(x = 1)$

(iv) $EG(x = 1)$

(v) $\neg EG(x = 0)$

(4)

Question 2:

(a). What is meant by the terms *safety property* and *liveness property* when applied to concurrent programs? (4)

(b). Consider the mutual exclusion protocol for 2 processes shown below:

```
// Process 1
<init1>;
while(true) {
    turn = 2;
    c1 = true;
    while (c2 && turn == 2) { };
    <crit1>;
    c1 = false;
    <rem1>;
}
```

```
// Process 2
<init2>;
while(true) {
    turn = 1;
    c2 = true;
    while (c1 && turn == 1) { };
    <crit2>;
    c2 = false;
    <rem2>;
}
```

```
// Shared variables
boolean c1 = false, c2 = false;
integer turn = 1;
```

Does the protocol satisfy the properties of Mutual Exclusion, Absence of Deadlock, Absence of Unnecessary Delay and Eventual Entry? If not, give an example trace that results in the property being violated and fix the algorithm *using only standard instructions* so that it satisfies all the properties. Explain your solution. (15)

(c). Modify the mutual exclusion protocol given in part (b) above to implement a *busy-waiting* producer-consumer protocol. Assume that process 1 is the producer, process 2 is the consumer and that the processes communicate using a single element buffer. State the safety and liveness properties of your solution. (6)

Question 3:

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

(b). An n-player game is implemented as a concurrent program consisting of $n + 1$ processes, one for each player and a gameserver process which manages the game state. The processes communicate using shared buffer of length n. At each round in the game, each player process makes a move by writing into the ith slot of the buffer, where i is the number of the player process. When all the players have made their moves, the gameserver process reads the moves and updates the game state. *Using semaphores*, devise a solution (in pseudo-code) to the buffer synchronisation problem which ensures that:

- each player can move only once in any turn;
- no moves are read until the buffer is full;
- each move is read only once;
- no moves are read from an empty buffer; and
- all moves made in a round are eventually read.

Assume that the buffer is empty when the game starts. Explain your solution and say why it is correct. (15)

(c). Explain the advantages and disadvantages of semaphore-based and monitor-based solutions to concurrent programming problems. Illustrate your answer using your solution to part (b). (5)

Question 4:

Consider the mutual exclusion protocol for 2 processes shown below:

```
// Process 1
<init1>;
while(true) {
    while (TS(lock)) { };
    <crit1>;
    lock = false;
    <rem1>;
}
```

```
// Process 2
<init2>;
while(true) {
    while (TS(lock)) { };
    <crit2>;
    lock = false
    <rem2>;
}
```

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

where the special Test-and-Set atomic instruction, `TS`, is defined as:

```
boolean TS(boolean lock) {
    boolean v = lock;
    lock = true;
    return v;
}
```

Assuming the following notation:

Propositional variables: s_i, c_i and n_i where s_i means 'process i is spinning in its entry protocol', c_i means 'process i is in its critical section', and n_i means 'process i is not in its critical section or trying to enter'.

State transitions: a_i and b_i where a_i means 'process i evaluates the loop condition in its entry protocol' (if `lock` is false, this results in `lock` being true and process i in critical section; if `lock` is true, this results in process i spinning), and b_i means 'process i sets `lock` to false in its exit protocol' (which results in `lock` being false and process i no longer being in its critical section or trying to enter).

(a). Draw the state transition system representing the two processes. Assume the system starts in state0: ($\neg lock, n_1, n_2$) (5)

(b). Write a CTL formula expressing the property of mutual exclusion: processes are never in their critical sections at the same time. (5)

(c). Is the formula expressing mutual exclusion true in the transition system representing the two processes? Justify your answer using the truth definition for CTL formulas. (5)

(d). Write a CTL formula expressing the property of eventual entry: if a process is trying to enter its critical section it will eventually do so. (5)

(e). Is the formula expressing eventual entry true in the given transition system representing the two processes? Justify your answer using the truth definition for CTL formulas. (5)

Question 5:

(a). A simple fixed-size last in first out (LIFO) queue supports three operations: *push* adds an item to the end of the queue, *pop* removes the last item from the queue and returns it and *length* returns the number of items currently in the queue. The length of the queue can increase and decrease but can't exceed the size of the queue and should never be negative.

Develop a Java `LIFO` class which allows safe concurrent update of `LIFO` objects. The class should implement the following four public methods:

- `void LIFO(long n)`: (constructor) creates a new `LIFO` object with a queue of size `n`;
- `void push(Object o)`: if the length of the queue is less than the size of the queue, adds `o` to the end of the queue, otherwise waits until there is a free slot and then adds the object;
- `Object pop()`: if the length of the queue is greater than 0, removes the last object from the queue and returns it, otherwise waits until there is an object in the queue and returns it;
- `long length()`: returns the current number of objects in the queue.

Assume that the queue is held in an array of `Objects` of size `n`. Explain your answer. (15)

(b). What changes would be necessary to allow remote access to the `LIFO` class using Java's RMI? Assume there is a single queue and that clients make remote calls to manipulate the queue. Illustrate your answer using your solution to part (a) above. (10)

Question 6:

(a). Write a well structured and coherent essay comparing and contrasting the two main approaches to remote invocation in distributed processing implementations of concurrency. Illustrate your answer with reference to a client-server problem of your choice. (25)