TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2008

MID-YEAR

COMP 310

Concurrent Programming

Time Allowed: 3 Hours

Instructions: Paper foreign language dictionaries are permitted.

Non-programmable calculators without full alphabetic keys are permitted.

Electronic dictionaries and programmable calculators are not permitted.

An appendix at the end of the exam lists some useful algorithms.

Answer all of the following **six** questions:

1. Critical Sections.
2. Semaphores and Monitors.
3. Fine-grained Synchronisation and Relaxed Memory Models.
4. Channels and Data Spaces.
5. Mutual Exclusion in Distributed Systems.
6. Termination and Consensus.

Each question is worth 30 marks.

The exam is worth 180 marks in total.

Question 1. Critical Sections [30 marks]

Boolean a := false, Boolean b := false	
p	q
$p1$: **while** !b	$q1$: **if** !a
$p2$: a := true;	$q2$: **print**("A");
$p3$: **print**("$*$");	$q3$: b := true;

(a) [4 marks] Draw a state diagram or scenario showing how the above program can print the string "*A".

(b) [4 marks] Describe in words all the possible outputs of this program.

The program below is Dekker's solution to the critical section problem.

turn := 1, wantp := false, wantq := false	
p	q
loop :	loop :
$p1$: *noncriticalsection*;	$q1$: *noncriticalsection*;
$p2$: wantp := true;	$q2$: wantq := true;
$p3$: **while** wantq = true	$q3$: **while** wantp = true
$p4$: **if** turn = 2	$q4$: **if** turn = 1
$p5$: wantq := false;	$q5$: wantp := false;
$p6$: **while** wantq = true;	$q6$: **while** wantp = true;
$p7$: *criticalsection*;	$q7$: *criticalsection*;
$p8$: wantp := false;	$q8$: wantq := false;
$p9$: turn := 2;	$q9$: turn := 1;

(c) [2 marks] State the mutual exclusion property for Dekker's algorithm.

(d) [4 marks] Does Dekker's algorithm have starvation freedom? Explain why or why not.

(e) [6 marks] Prove inductively that Dekker's algorithm satisfies the following invariant:

$$p3..5 \vee p8 \Rightarrow wantp$$

The base case is clear because p8 is false in the initial state.

Execution of lines p1, p5, p6, and p8 to p10 all make the antecedent of the invariant false. So the invariant is preserved by execution of these lines.

When p executes line p2 it assigns true to wantp, so the consequent of the invariant is true in the poststate. The same is true when p executes line p7.

Lines p4 and p5 do not modify any variables mentioned in the invariant, so the invariant is preserved. Finally, when p executes line p3 (whether or not wantq is true), no variables mentioned in the invariant are modifed either. So the invariant is preserved.

(f) [3 marks] Prove that execution of line $p3$ preserves the mutual exclusion property. You may use the invariant from the previous question, or the symmetric invariant:

$$q3..5 \vee q8 \Rightarrow wantq$$

*If wantq is true in the prestate, then p*8 *is false in the poststate, so the invariant is preserved. If wantq is false, then by the invariant applied to process q, q*8 *must be false in the prestate. It is not modified by the statement, so q*8 *is false in the poststate also, and thus the invariant is preserved.*

(g) [2 marks] Briefly describe the difference between a *blocking* semaphore and a *busy-wait* semaphore.

During a `wait` *operation on a* blocking *semaphore, if the value of the semaphore is zero, the process becomes blocked, so that it is not executing instructions. During a* `wait` *operation on a* busy-wait *semaphore, if the value of the semaphore is zero, the process spins in a loop, checking whether it can complete the wait operation.*

(h) [5 marks] Define a Java class called `Lock`, having an `acquire` method and a `release` method. These methods should provide mutual exclusion in the following sense: after a thread completes the `acquire` method, no other thread should complete `acquire` until the first thread has executed `release`. The `release` method should throw a `RuntimeException` if the thread executing `release` is not the thread that most recently completed `acquire`.

Question 2. Semaphores and Monitors [30 marks]

(a) [3 marks] There are three processes p, q and r. Write pseudocode for each process so that:

- p prints the string `"p"`.
- q prints the string `"q"`.
- r prints the string `"r"`.
- After all processes have finished, `"p"`, `"q"` and `"r"` have been printed in that order.

You should use two semaphores `sem1` and `sem2`. Remember to specify the initial value of each semaphore. Assume that there is a procedure `print(String s)` that prints the string `s`.

The code below describes a monitor with two operations, `LeaveWork` and `GetWork`. When a process calls `LeaveWork`, it assigns the value `w`, which represents some work to be performed, to the variable `nextWork`. It is intended that once this has been achieved, the next `GetWork` operation will return the value `w`.

```
monitor WorkMonitor
  variable nextWork;
  condvar leavers, workers;

  operation LeaveWork(w):
     if (nextWork != null)
          waitC(leavers);
     nextWork = w;
     signalC(workers);

  operation GetWork():
     if (nextWork == null)
          waitC(workers);
     w = nextWork;
     nextWork = null;
     signalC(leavers);
     return w;
```

(b) [6 marks] Under which of the following two signalling disciples does the above monitor work correctly: E<S<W (signal-and-continue), E=W<S (Java's signalling discipline)? For each discipline, briefly explain why the monitor is correct or incorrect. Focus on whether or not the variable `nextWork` can be overwritten when it is not `null`.

(Question 2 continued on next page)

(Question 2 continued)

(c) [5 marks] Modify the `WorkMonitor` so that `LeaveWork(w)` does not return until the work `w` has been completed. You should create an operation `FinishedWork(w)` which a process calls after it has completed a unit of work, and you should define a new condition variable `waitForCompletion`. You are not required to modify the `GetWork` operation. Assume that the monitor has the signalling discipline E<W<S.

(d) [7 marks] There are several differences between the classic monitors that we discussed in class, and the monitor-like behaviour of objects in the Java language. Apart from Java's signalling discipline, describe some of the other differences. For each of the features of Java's monitors that you discuss, evaluate its impact on concurrent programming, relative to the classic monitors.

(e) [5 marks] Define a monitor that provides `Sleep` and `Wakeup` operations, with the following specification.

- When a process calls `Sleep`, it always blocks.
- When a process calls `Wakeup`, all processes that are currently blocked (because of an earlier call to `Sleep`) become unblocked.

Assume that your monitor has the signalling discipline E<W<S (signal-and-continue). Also, you may use the test `empty(cv)` that returns true if and only if the condition variable `cv` has no waiting processes. You need to declare any variables or condition variables of the monitor, write the `Sleep` operation and write the `WakeUp` operation. (This specification is like that of Assignment 2, but in the assignment you used semaphores in the implementation.)

(f) [4 marks] Analyse how your monitor behaves under the signalling discipline E=W<S. Does it perform correctly? What might go wrong? If it is still correct, explain why you think this is so.

Question 3. Fine-grained Synchronisation and Relaxed Memory Models
[30 marks]

(a) [4 marks] State important advantages and disadvantages of using fine-grained synchronisation in multithreaded or multiprocess code.

(b) [2 marks] Relaxed memory models make programming multiprocessors difficult. Explain why hardware designers might choose to provide only relaxed memory models.

(c) [4 marks] Explain why the following operation history is not sequentially consistent.

p :	$W(x)1$	$R(y)0$	
q :		$W(y)1$	$R(x)0$

(d) [4 marks] An operation history is *PRAM-consistent* iff every operation of each process p is observed by every other process in the order that p executed those operations. Show, by constructing a suitable local view for each process, that the following operation history is PRAM consistent.

p :	$W(x)1$	$R(y)0$		
q :		$W(y)2$	$W(y)3$	
r :		$R(y)2$	$R(x)1$	$R(y)3$

The following questions are concerned with implementations of concurrent container classes in Java. We use the following `Entry` class in these implementations. (The instance variables have been marked `public` so that they can be accessed more simply, without using getters and setters.)

```
public Entry {

  public Object value;
  public Entry next = null;

  public Entry(V val) {
    value = val;
  }
}
```

(Question 3 continued on next page)

(Question 3 continued)

The code below is part of a Java class definition for the two-lock queue that we studied in class, including instance variables, a constructor, and the `dequeue()` method. In the following two questions, you will implement two methods of this class: `enqueue()` and `isEmpty()`.

```
public class TwoLockQueue {
  Object enqLock;
  Object deqLock;
  Entry head;
  Entry tail;

  public TwoLockQueue() {
    head = new Entry(null);
    tail = head;
  }

  public Object dequeue() {
    synchronized(deqLock} {
      if (head.next == null)
        return null;
      Object value = head.next.value;
      head = head.next;
    }
    return value;
  }
...
}
```

(e) [4 marks] Write an `enqueue(Object v)` method, which adds the `Object v` onto the end of the queue.

(f) [2 marks] Write an `isEmpty()` method, which returns true if and only if the queue is empty at some point during the execution of the method.

(Question 3 continued on next page)

(Question 3 continued)

The code below is part of a Java class `TSTack` that implements the Treiber stack discussed in lectures.

```
public class TStack {
    AtomicReference<Entry> top;

  public Tstack(){
    top = new AtomicReference<Entry>(null);
  }

  void push(Object x) {

    if (x == null)
      throw new IllegalArgumentException();

    Entry e = new Entry(x);

    while (true) {
      Entry t = top.getValue();
      e->next = t;
      if (top.compareAndSet(t, e))
        return;
    }
  }
}
```

(g) [5 marks] Implement the `pop` operation.

(h) [5 marks] Consider the following relaxed memory model:

> All processes observe each `compareAndSet` operation in the same order. (I.e., all `compareAndSet` operations appear in the local view of each process in the same order.) Read and write operations of one process can be observed by other processes in any order.

Explain why the implementation of the `push` method given in the class `TStack` above is incorrect under this memory model.

Question 4. Channels and Data Spaces [30 marks]

(a) [5 marks] Define the semantics of Linda's `postnote` and `removenote` operations.

(b) [10 marks] Use channels to write a pipeline program that accepts a non-ending stream of characters. The pipeline program has two functions: (1) Count the number of words seen so far (a word is defined as one or more characters delimited by a single space); (2) Reverse the order of the characters in each word before sending the word onto a sink process. For example, a stream such as "The fox jumped over the lazy dog " would result in a word count of seven and "ehT xof depmuj revo eht yzal god " being output.

You should compose your pipeline as an environment process, two processes to implement the word count and reverse word functionality, and a sink process.

> *Expect an environment and sink for completeness. The wc just counts spaces. The reverse reads a character and buffers it, when a space is read it reads the characters out in reverse to the next process (the sink). Four processes. 2 marks for environment and sink. 3 marks for the wc. 5 marks for reverse.*

(c) [5 marks] Consider the matrix multiplication algorithm. Briefly explain how you would use selective input to prevent unecessary blocking.

> *The standard algorithm always reads from the North and then then the East channel. THis is inefficient if there is input available on the second channel but not on the first. The selective input construct allows the algorithm to either read first from North then East or read from East then North. This avoids the algorithm from blocking unecessarily.*

(d) [5 marks] What are the advantages of having clients retrieve tasks from a dataspace compared to having a master process send tasks to clients using channels?

> *Loosely coupled versus tightly coupled. Clients have to know only of the existence of the dataspace versus master having to know existence of all the clients. Clients can also judge how much work that they can do. No need to explicitly configure the channels.*

(e) [5 marks] Explain the motivation for the development of Rendezvous and RPC.

> *Higher-level constructs – two-way flow, don't need to set up pairs of channels between processes, introduction of tools to make it easier for the programer.*

Question 5. Distributed Algorithms [30 marks]

(a) [5 marks] Anton has written a distributed algorithm that involves a node sending ticket numbers to another node. The sending node chooses the ticket number by incrementing the previous ticket number by one. Anton assumes, therefore, that the receiving node will always receive monotonically increasing ticket numbers. What is wrong with this assumption and how would you work around it?

No! Messages may be reordered, this means he may receive a lower number first! A fix for this would be to take the maximum of the numbers received rather than just storing what is received.

(b) [5 marks] Consider the following algorithm and explain the result of its execution. Indicate what would be printed to the console and where the algorithm may block.

Distributed Algorithm		
Node 1	**Node 2**	**Node 3**
p1: integer x	q1: integer x	r1: integer a, b, c, d
p2: send(subtract, 3, 2, 30, 40)	q2: receive(result, x)	r2: receive(subtract, a, b, c)
p3: receive(result, x)	q3: send(result, 1, x*2)	r3: d = b - c
p4: print x		r4: send(result, a, d)

Node 1 sends subtract request to node 3 but passes node 2's address with the request. The result is then sent back to node 2 instead of node 1. Node 1 will continue blocking at q2.

Looking for proper assignment of parameters to variables and understanding of role of sending myID with a message.

Some marks for partial understanding.

(c) [10 marks] Consider the Ricart-Agrawala permission based algorithm (Algorithm 10.2 in the appendix).

(i) What is the total number of messages sent in a scenario in which m out of n nodes enter their critical sections once?

Each of the m nodes must send n-1 requests and wait for n-1 replies

(ii) Describe a scenario leading to unbounded ticket numbers.

Even with just two nodes, scenarios will have unbounded ticket numbers. The reason is that highestNum never decreates and myNum is always increased to be higher than highestNum. For example, Aaron chooses 1 and sends a request to Becky; Becky receives 1, replies, chooses 2 and sends a request to Aaron; Aaron has left his critical section, so he relies to Becky and then chooses 3; and so on.

(d) [10 marks] Consider the Ricart-Agrawala token passing algorithm (Algorithm 10.3 in the appendix).

(i) Describe a scenario where a node receives multiple requests for entry into a critical section without seeing a corresponding reply from another node. Explain why this cannot happen with the permission-based version of the algorithm.

Draw a scenario where node A receives the token and passes it to another node B. Node A makes another request. Node C may receive these two requests. The key point is under the token-based algorithm not all nodes must receive the request before mutual exclusion is granted whereas under the permission-based algorithm this is the case. In the example discussed, node C would have to receive the first request and reply before node A could receive the token.

(ii) Discuss whether, in some node j where $j \neq i$, $requested[i]$ can be less than $granted[i]$?

No, granted is set in statement p10 to be the value of the local variable myNum, which is also the value that has been sent to other nodes and received as requested. Therefore, requested for i at j will never be less than granted for i. It may be greater than granted for i during the interval between choosing and sending myNum and receiving the token.

Question 6. Termination and Consensus [30 marks]

(a) [15 marks] Consider the DS algorithm (Algorithms 11.2 and 11.3 in the appendix) applied to a scenario with four nodes that are fully connected. The nodes are numbered 1 to 4. Node 1 is the environment node.

(i) Complete the following table showing the internal state of the nodes. The second column shows the actions: $n \Rightarrow m$ means that node n sends a message to node m. The other columns show the local data structures at each node: $(parent, inDeficit[E], outDeficit)$. Note that in your answer, the values of the array $inDeficit[E]$ should be shown in increasing numerical order of the nodes.

Actions and Internal States of Nodes					
Step	**Action**	**Node 1**	**Node 2**	**Node 3**	**Node 4**
1	$1 \Rightarrow 2$	$(-1, [], 0)$	$(-1, [], 0)$	$(-1, [], 0)$	$(-1, [], 0)$
2	$2 \Rightarrow 3$				
3	$3 \Rightarrow 4$				
4	$4 \Rightarrow 2$				
5	$4 \Rightarrow 3$				
6					

(ii) Draw the spanning tree that exists at step 6.

2 marks A to B to C to D

(iii) Carol wants to send a signal. Can she do this? With reference to the state of internal variables explain why or why not.

3 marks, no – not chosen to terminate yet, outdeficit = 1 and indeficit = 1, has to wait for children to terminate so know that computation is over

(b) [5 marks] When considering the termination algorithms we assume that communication channels are synchronous. Discuss what effect this would have on the Dijkstra-Scholten algorithm and on the Credit-Recovery algorithm (Algorithms 11.4 and 11.5 in the appendix) if we assumed that communications channels were asynchronous.

We do not need to account for messages buffered in the channels. DS would require logic added to account for sending messages. Credit-recovery would not require a change because it already accounts for messages by reducing the weight of a node when a message is sent.

(c) [5 marks] Define the terms *crash failure* and *Byzantine failure*.

A crash failure: a traitor simply stops sending messages at any arbitrary point during the execution of the algorithm. A byzantine failure: a traitor can send arbitrary messages, not just the messages required by the algorithm

(d) [5 marks] Consider the one-round algorithm for Consensus (Algorithm 12.1 in the appendix). Explain under what circumstances a traitor in a group of three nodes can cause the loyal nodes to disagree about their final plans. Illustrate your answer with an example scenario.

Classic case, imagine three nodes. Basil sends A to Leo but crashes before sending to Zoe. Leo sends R to Basil and Zoe. Zoe sends A to Basil and Leo. We get a tie at Zoe leading to R, and majority of A at Leo

****