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

**EXAMINATIONS — 2010**

MID-YEAR

# NWEN 303

# 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.

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

1. Critical Sections and Semaphores.
2. Monitors.
3. Fine-grained Synchronisation.
4. Message Passing.
5. Distributed Programs (RPC, Rendezvous and Linda).
6. Paradigms for Distributed Parallel Programming.

Each question is worth 30 marks.

The exam is worth 180 marks in total.

## Question 1. Critical Sections and Semaphores [30 marks]

**(a)** [5 marks] Define the following correctness conditions with respect to the critical section problem.

**(i)** Mutual exclusion.

*Only one process may be in its critical section at any time.*

**(ii)** Deadlock freedom.

*If any process is executing the preprotocol (attempting to enter the critical section), then eventually some process enters the critical section.*

**(iii)** Starvation freedom.

*If any process is executing the preprotocol (attempting to enter the critical section), then eventually that process enters its critical section.*

**(b)** [15 marks] Consider the following solution to the critical section problem:

| **Critical section** | |
|---|---|
| integer turn := 1 | |
| **p** | **q** |
| loop forever | loop forever |
| p1: non-critical section | q1: non-critical section |
| p2: await turn = 1 | q2: await turn = 2 |
| p3: critical section | q3: critical section |
| p4: turn := 2 | q4: turn := 1 |

**(i)** Prove inductively that the solution has the following invariant:

$$\neg(p3 \wedge q3)$$

*See Ben-Ari.*

**(ii)** Explain why this invariant implies the mutual exclusion property.

*The only way that mutual exclusion can be violated is if two processes are in their critical region, in which case $p3 \wedge q3$ holds; the invariant ensures that this condition is always false.*

**(iii)** Is the above solution deadlock free? Justify your answer.

*When turn=1 (resp. 2), p (resp. q) can enter the CS and q cannot. This guarantees deadlock freedom (and mutual exclusion).*

**(iv)** Is the above solution starvation free? Justify your answer.

*The solution is not starvation free. If (eg) p does not complete its NCS when turn=1, q can never enter its CS.*

**(c)** [10 marks]

**(i)** A semaphore s has two components: an integer `s.value` and a set of processes `s.waitset`. Describe the effect of the `wait` and `signal` operations for a blocking semaphore `s`. You should describe the effect of each operation on `s.value` and `s.waitset`, and on the invoking processes.

`wait(s)` *first checks whether* `s.value` *is greater than* `0`*. If so, it decrements* `s.value` *and returns. If not, it adds the invoking process into* `s.waitset` *and puts the invoked process into its* blocked *state.*

`signal(s)` *first checks whether* `s.waitset` *is empty. If so, it increments* `s.value` *and returns. If not, it chooses an arbitrary process p from* `s.waitset`*, removes p from* `s.waitset` *and puts p in its* ready *state.*

*Both* `wait` *and* `signal` *execute atomically.*

**(ii)** Give a solution to the critical section problem using semaphores, and explain why it satisfies mutual exclusion and deadlock freedom

| **Critical section** | |
|---|---|
| semaphore s:={1,∅} | |
| **p** | **q** |
| loop forever | loop forever |
| p1: non-critical section | q1: non-critical section |
| p2: wait(s) | q2: wait(s) |
| p3: critical section | q3: critical section |
| p4: signal(s) | q4: signal(s) |

*Mutual exclusion:*

*Deadlock freedom: If some process is attempting to enter the CS it will eventually execute wait(s). When this happens, either the other process is in the CS or not. In the first case, the process in the CS will eventually complete and execute signal(s), enabling the waiting process to enter the CS. In the second case, s.value=1, so the wait operation can complete immediately.*

**(iii)** Explain the difference between a *weak semaphore* and a *strong semaphore*.

*A strong semaphore guarantees that waiting processes are signalled in the order in which they executed their wait operations. A weak semaphore does not make this guarantee.*

*Saying that processes wait in a wait queue in a strong semaphore, and a wait set in a weak semaphore is also ok.*

**(iv)** How does this difference affect whether your solution is starvation free?

## Question 2. Monitors [30 marks]

A multi-threaded message service uses a shared table to associate user id numbers with email addresses. The table supports two basic operations:

- `put(id, address)`, which can add a new id number and email address to the table, or alter the email address associated with an existing id number; and
- `get(id)`, which returns the email address associated with a given id number (or a null string if the id number is not in the table).

In this application, `get` operations are far more common that `put` operations.

**(a)** [14 marks] Explain how a monitor can be used to control access to the table, ensuring that multiple `get` operations are able to execute concurrently whenever possible.

You do not need to discuss details of the underlying table data structure; just assume suitable operations that can be invoked by the monitor operations.

**(b)** [8 marks] Describe the *signal and wait* and *signal and continue* signalling disciplines for monitors. How does the signalling discipline used affect the way the monitor is used?

**(c)** [8 marks] In Java, the *nested monitor problem* arises when one thread calls `wait()` on an object from within a `synchronized` block in another object. Explain why this can lead to deadlock.

> *A thread which calls* `obj.wait()` *will block at least until* `obj.notify()` *is called on that object by second thread. In a particular application, it may be that a thread will only call* `obj.notify()` *from a* `synchronized` *block in the object on which the original thread is* `synchronized`*. If this is the case, the original thread will never complete its synchronized block, and so the notifying thread can never call* `obj.notify()`*.*

## Question 3. Fine-grained Synchronisation [30 marks]

Suppose you wish to implement a set, to be shared by multiple processes, using an ordered linked list representation.

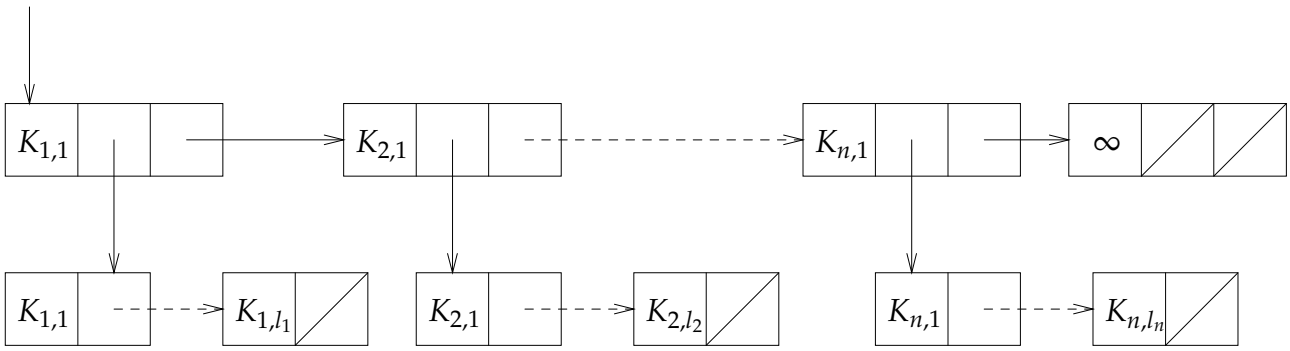
**(a)** [4 marks] The simplest way to implement the set is to use a single lock, which is acquired by any process wanting to perform an operation on the set and released when the operation is complete (or in Java, making each method on the set `synchronized`).

Explain the drawbacks of this approach.

**(b)** [10 marks] Explain how *hand-over-hand locking* can be used to improve the performance of a shared linked list, and give pseudo-code (or Java) to insert an item into an ordered linked list.

**(c)** [8 marks] While hand-over-hand locking is better than a single lock, it is still not ideal. Explain some ways in which hand-over-hand locking limits possible concurrency, and describe another approach to implementing a shared linked list which overcomes some of these problems.

**(d)** Another way to improve the performance of a shared linked list is to modify the data structure in a way that will allow more concurrency. For example, we might divide the list into a list of several smaller lists, as shown in the following diagram:




Each item in the top level list contains a key and a pointer to a lower level list; the top level list ends with a sentinel node containing a value larger than all possible keys. The concatenation of the keys in the lower level lists (i.e. $K_{1,1}, \ldots, K_{1,l_1}, K_{2,1}, \ldots, K_{2,l_2}, \ldots, K_{n,1}, \ldots, K_{n,l_n}$) forms a strictly ascending sequence containing all of the elements of the set.

To locate an item's position in the list (for insertion, deletion or look-up), we search the top level list to determine which of the smaller lists it should be in, then search within that list. When items are inserted, we need to determine whether to add a new element to the top level list; we will assume that this is a random choice. When the last item of a lower level list is deleted, the corresponding element of the top-level list is deleted.

**(i)** [6 marks] Discuss how fine grained locking could be used to implement a shared set using this representation and describe the performance gains you would expect.

**(ii)** [2 marks] What, if any, advantage is gained by repeating the first key of each lower level list in the top level list?

## Question 4. Message Passing [30 marks]

**(a)** [5 marks] Explain the operation of the following pseudo-code (note that we assume two channels `ch1` and `ch2` have already been defined as well as a variable `x`):

```
either
  ch1 ⇒ x
or
  ch2 ⇒ x
```

*Only one alternative can succeed.*
*If communication can take both on multiple channels then one is chosen non-deterministically and succeeds (either ch1 or ch2).*
*Otherwise it is the alternative branch with the communication statement that could succeed.*
*If neither can succeed the process blocks.*

**(b)** [15 marks] Write an algorithm for a pipeline of processes that can sort a stream of numbers. Assume that there are $N$ processes in the pipeline and the input stream of $N$ numbers is followed by a special end of stream marker (EOS).

For example, a pipeline composed of five processes should be able to transform an input stream of 5, 6, 1, 10, 12, EOS into an output stream of EOS, 1, 5, 6, 10, 12.

*Should have N processes.*

*There are two modes of operation for each process in the pipeline.*

*1. If EOS has not been seen then each process accepts an input number and checks to see if it is the smallest that it has seen so far. If it is the smallest so far it passes on the previous smallest to downstream processes, otherwise it retains its older smallest and passes on the newly received number. When all values from the input stream have been seen then the result should be that the pipeline itself holds the input stream in ascending order.*

*2. When a EOS is accepted it passes EOS on to the neighbour followed by the stored number. Any numbers subsequently received from upstream are simply passed along to the downstream neighbours. This means that eventually the numbers are read out in ascending order.*

*Need to describe how the processes are connected to each other.*

**(c)** [10 marks] Outline the implementation of a simple server that provides two distinct operations, for example `push` and `pop`. The `push` operation does not return anything and has an integer as its single parameter. The `pop` operation returns an integer and does not have any parameters. The integer values passed to the server should be stored on a stack.

*Key issues: the kind of request is passed. Body determines the kind of requests. Extra marks to be gained if include pseudocode for the types of data*

## Question 5. Distributed Programs (RPC, Rendezvous and Linda) [30 marks]

**(a)** [5 marks] Briefly identify some advantages that the remote procedure call paradigm has over channel-based message passing.

*Easier for programmers because maps from notion of local procedure call, removes need to write your own marshalling and unmarshalling code, removes need to explicitly manage server instances and removes the need for static channel declaration*

**(b)** [5 marks] Briefly identify the differences and similarities between remote procedure call paradigm and the rendezvous paradigm.

*Diffs: guarded communication, one instance of server for rendezvous, synchronisation. Sims: use of remote calls, blocking upon call to server*

**(c)** The following is an outline of a module implementing a time server that provides timing services to client processes in other modules:

```
module TimeServer
  op delay(int interval);
body
  int tod = 0;
  queue of (int waketime, int process_id) napQ;
  proc delay(interval) {
    int waketime = tod + interval;
    insert (waketime, myid) at appropriate place on napQ;
  }
  process Clock {
    while (true) {
      increment tod using hardware clock;
      while (tod >= smallest waketime on napQ) {
        remove (waketime, id) from napQ;
      }
    }
  }
end TimeServer
```

**(i)** [4 marks] What concurrency problems might occur in the above example? Justify your answer.

*delay and Clock both update the queue (a shared variable). Each invocation of delay is serviced by a new process and these may interfere with each other as well as with the Clock process.*

**(ii)** [6 marks] As well as concurrency problems, the module does not currently delay processes correctly. Rewrite the module so as to fix these problems.

*Need to add two variables: sem m = 1 for controlling access to the queue and sem d[n] (initialised to 0) for each process.*

*Modify delay so insert is surrounded by P(m) and V(m). Last statement should call P(d[myid])).*

*Modify Clock so access to queue is protected and when remove takes place that the process is awoken (V(d[id])).*

**(d)** [10 marks] Outline how you would implement an array using Linda. Assume that you must support multiple arrays, the ability to iterate through the array and synchronised access to the array. Use code examples to illustrate your discussion.

*Looking for use of code snippets to show how to insert, remove and iterate through the array. Expect them to show how each array can be named. Synchronisation can be achieved through Linda equivalent of a semaphore that is used before writing or reading the array.*

## Question 6. Paradigms for Distributed Parallel Programming [30 marks]

**(a)** [5 marks] Contrast the heartbeat interaction paradigm with the pipeline interaction paradigm.

*Heartbeat used where:*

*- can divide work evenly between workers*

*- computation can be structured in terms of relationships between immediate neighbours*

*- each worker responsible for own subset of data*

*- low cost of communication between work units or computation cost greatly outweighs the cost of communication.*

*Pipeline:*

*- can divide work evenly between workers*

*- where workers may be required to carry out the different computations on the same data AND/OR computation can be rewritten in terms of a recurrence relationship ie my computation depends upon or relates to the computation of my predecessor*

*- data has to circulate amongst all the workers*

*- low cost of communication between work units or computation cost greatly outweighs the cost of communication.*

**(b)** [10 marks] Explain briefly how you would implement matrix multiplication using the pipeline paradigm. Describe how the process interaction is structured, the details of the task performed by each process, and how concurrency is achieved. Do not write any code when answering this question.

*Each process in the pipeline multiplies a row and a column.*

*Each process reads input vectors from previous stage and writes results out to the next stage of the pipeline.*

*Must discuss pipeline filling and draining.*

*Concurrency achieved when pipeline is full.*

**(c)** [10 marks] Explain briefly how you would implement the Game of Life using the heartbeat interaction paradigm. Describe how the process interaction is structured, the details of the task performed by each process, and how concurrency is achieved. Do not write any code when answering this question.

Recall that the Game of Life is as follows. A two-dimensional board of cells is given. Each cell either contains an organism (it's alive), or is empty (it's dead). Each cell has eight neighbours (ignore the issue of cells at the edge of the board). The cells live or die according to the following rules:

- A cell will live if it has two or three live neighbours, otherwise it will die.

- A dead cell with exactly three live neighbours becomes alive.

*Each cell is modelled by a separate process. The cells are arranged in a nxn matrix. This provides good concurrency. Because of the blocking receive each cell cannot get more than an iteration ahead of each other.*

*Communication is via a set of exchange channels, one for each process. Each cell will announce its position and its state to the other cells.*

*There are three main phases to the algorithm : - Exchange state with 8 neighbours. Loop over each neighbour cell and announce this cells position and state to them. (send operation) - Receive state updates from all 8 neighbours. (receive operation) - Apply the Game of Life rules.*

*This algorithm repeats as many times as there are generations of cells.*

**(d)** [5 marks] Identify the two main factors that influence the overall performance of a distributed parallel program and discuss how "chunking" can be used to improve performance.

*Speed of local computation/processing and speed of communication. Ideally want to minimise cost of message passing and maximise local computation. Chunking can be used to do this although care has to be taken to ensure that the increase in data sent does not have a negative effect on overall performance because of the impact upon the speed of communication*

**********