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

EXAMINATIONS — 2005

END-YEAR

COMP 310
System and Network Programming

Time Allowed: 3 Hours

Instructions: Answer all **six** questions.

Each question is worth 30 marks.

The exam is worth 180 marks in total.

Paper foreign language dictionaries are permitted.

Non-programmable calculators without full alphabetic keys are permitted.

Electronic dictionaries and programmable calculators are not permitted.

continued...

Question 1. General Concepts of Concurrency [30 marks]

(a) [10 marks]

(i) Explain what is meant by *data independence*, and how data independence simplifies the design of concurrent programs. Illustrate your answer by showing how data independence can be exploited in designing a program to sum an array using two processes.

(ii) Suppose a concurrent program to sum an array is run on a multiprocessor system where the two processes execute on separate processors of greatly differing speed. Would a data independent solution be suitable to use in this environment? Explain why or why not.

(b) [20 marks]

(i) Explain, with reference to the following program, how different assumptions about what actions can be performed atomically can affect the possible outcomes of a concurrent program.

```
int x = 0; y = 0;
co x = y+1; // y = x+1; oc;
```

(ii) Explain briefly how the *At-Most-Once Property* can be used to determine whether an assignment statement can be considered to be atomic when read and write are the only atomic operations on variable locations.

(iii) Explain why it is often advantageous when designing concurrent programs to make unrealistic assumptions about what actions can be performed atomically. Illustrate your answer using a suitable example.

Question 2. Synchronisation and Barriers [30 marks]

(a) [6 marks] Explain what is meant by *barrier synchronisation*, and describe the kind of situation where a concurrent application would need to use barrier synchronisation.

(b) The following is an outline of the code to implement a simple barrier using a shared counter:

```
int count = 0;
process W[i=1 to n] {
  while (true) {
    code for task i;
    <count = count+1;>
    <await (count == n);>
  }
}
```

(i) [2 marks] Explain briefly how this solution works.

(ii) [6 marks] Describe and explain **two** ways in which the action `<count = count+1;>` could be implemented.

(iii) [6 marks] Describe and explain **two** ways in which the action `<await (count == n);>` could be implemented.

(c) [10 marks] The above implementation has two important problems: firstly, it does not reset the counter; and secondly, all of the processes access the same counter.

Explain why each of these is a problem, and describe a way of implementing a barrier that avoids both problems.

Question 3. Monitors and Java [30 marks]

Consider the following monitor to be used in solving the Readers/Writer Problem:

```
monitor RW_Controller {
  int nr = 0, nw = 0;    ## (nr == 0 ∨ nw == 0) ∧ nw <= 1
  cond oktoread;    # signalled when nw == 0
  cond oktowrite;   # signalled when nw == 0 and nw == 0
  procedure request_read() {
    while (nw > 0) wait(oktoread);
    nr = nr + 1;
  }
  procedure release_read() {
    nr = nr - 1;
    if (nr == 0) signal(oktowrite);
  }
  procedure request_write() {
    while (nr > 0 || nw > 0) wait(oktowrite);
    nw = nw + 1;
  }
  procedure release_write() {
    nw = nw - 1;
    signal(oktowrite);
    signal_all(oftoread);
  }
}
```

(a) [2 marks] Explain how procedures `request_read`, `release_read`, `request_write` and `release_write` are used to control access to the database.

(b) [2 marks] Why would you use a monitor like this rather than simply encapsulating the database in a monitor?

(c) [6 marks] Explain the effects of the `signal` and `signal_all` statements.

(d) [12 marks] Explain the significance of the invariant `(nr == 0 ∨ nw == 0) ∧ nw <= 1`, and present an argument to show that this invariant always holds.

(e) [8 marks] Explain briefly how you would implement this monitor in Java.

Question 4. Asynchronous and Synchronous Communication [30 marks]

Consider the following monitor to be used in solving the Resource Allocation Problem.

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = initial values;
  cond free;      # signaled when a process wants a unit
  procedure acquire(int &id) {
    if (avail == 0)
      wait(free);
    else
      avail = avail - 1;
    remove(units, id);
  }
  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail + 1;
    else
      signal(free);
  }
}
```

(a) [8 marks] Rewrite the monitor as a server using Andrews' *channel notation* for asynchronous message passing.

See figure 7.7

(b) [4 marks] Explain how you have simulated the monitor by a server process. In particular, explain how you have simulated procedure calls, monitor entry, procedure return and conditional variables.

Looking for understanding of the code

(c) [4 marks] Explain why synchronous message passing places an upper bound on the size of communication channels and may lead to reduced concurrency.

Bound on size of communication channels and on buffer space. Until message is received the sender will block preventing the sending of additional messages. Each sender process can have a maximum of one message queued on the sender channel.

Reduced concurrency because at least one of two communication processes will block. Depends on who tried to communicate first.

(d) [2 marks] Briefly describe the changes required to your server to make it use synchronous message passing.

There is no change required to the server, the only change required is to the client who should use synch_send statements

(e) [8 marks] Rewrite the monitor as a server using CSP.

the answer should make use of the ability to use guarded communication

(f) [4 marks] What features of CSP lead to a more concise implementation than the one you wrote for part **(a)**.

Ability to more acquire and release messages on the same channel; just use different ports and do statement for each case. Ability to delay receing an acquire message until there are available units; removes need to store pending requests.

Question 5. Remote Invocation [30 marks]

(a) 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) [2 marks] Identify the background process and exported operation in the module.

BP is ... and exported operation is ... Clock is the background process and delay are the exported operations.

(ii) [2 marks] Briefly describe how two modules in different address spaces can communicate.

Process in one module will invoke exported operations of another module.

(iii) [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.

(iv) [6 marks] As well as concurrency problems, the module does not currently delay processes correctly. Rewrite the module so as to fix both 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 $_{a}nd_{w}henremovetakesplacethattheprocessisawoken(V(d[id]))$.

(b) [6 marks] Describe how you would use rendezvous to implement the Shortest-Job-Next allocator module shown below. What special features of rendezvous make it a more concise implementation than an equivalent implementation using monitors?

```
module SJN_Allocator
  op request(int time) # shortest requests honoured first
  op release();        # job has finished
body
  ...
end SJN_Allocator
```

Request should have a synch boolean preventing access unless free plus a scheduling expression that chooses smallest time first. No need to maintain internal queue. Merely delays accepting calls of request until the resource is free, and only accept the call with the smallest argument for time.

(c) [4 marks] In Andrews' *Multiple Primitives Notation* there are two ways to invoke an operation (`call` and `send`) and two ways to service an invocation (`proc` and `in`). What are the four possible effects of combining these?

call + proc = procedure call.
call + in = rendezvous
send + proc = dynamic process creation
send + in = asynchronous message passing

(d) [6 marks] Ada95 introduced protected types and the `requeue` statement. Briefly define the semantics of these two mechanisms and discuss why they were introduced.

Protected types support synchronized access to shared data – like monitors, only one thread of control can be active at one time in a body. This saves the programmer needing write their own synchronisation calls. Synchronisation is required because tasks can share variables.

Requeue resubmits a call. It allows scheduling and synchronization that depends upon the arguments of calls. Ada 83 didn't allow accept statements to include references to arguments of calls.

Question 6. Process Interaction Paradigms [30 marks]

(a) [4 marks] Contrast the manager/workers interaction paradigm with the pipeline process interaction paradigm.

Manager/worker is a distributed bag-of-tasks. Worker processes share a bag of tasks held by a manager. The tasks are received from the manager and results returned to the manager. It's a many-to-one interaction. In the pipeline paradigm information flows from one process to another using a receive then send interaction. It is a one-to-one interaction.

(b) [4 marks] Use the logical clock update rules to assign timestamps to the following send and receive events amongst a group of communicating processes. Does event `f` *happen before* event `d`? Justify your answer.

Note that the initial values of the local clock for each process are shown on the left of the diagram and time advances monotonically from left to right.

(c) Consider the following "solution" for the distributed dining philosophers problem.

```
module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while(true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter
```

```
process Philosopher[i = 0 to 4] {
  int first = i, second = (i+1) mod 4;
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    eat;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    think;
  }
}
```

(i) [2 marks] What is wrong with the above "solution"? Explain.

The current solution will deadlock. If every philosopher gets their left fork then each will be waiting for the other philosopher to put it down.

(ii) [6 marks] Rewrite the "solution" so it is correct.

```
module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while(true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  int first = i, second = i+1;
  if (i == 4) {
    first = 0; second = 4; }
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    eat;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    think;
  }
}
```

(iii) [4 marks] Is your corrected solution fair? Explain why or why not.

Yes because forks are requested one at a time and invocations of getfork are serviced in the order they are called. Each call of getforks is serviced eventually, asssuming philosophers eventuall release forks they have acquired.

(d) [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. (4 marks)

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. (2 marks)

There are three main phases to the algorithm (6 marks):

- *Exchange state with 8 neighbours. Loop over each neighbour cell and announce this cell's 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.

****