TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2004

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) [20 marks] Explain briefly what is meant by each of the following terms. Discuss, using appropriate examples, their relevance to describing the behaviour of concurrent systems.

(i) Safety properties

(ii) Liveness properties

(iii) Fairness

(iv) Data independence

(b) [10 marks] The following is a proof outline, showing that the concurrent statement `co x = x+1; // x = x+2; oc` will establish the condition `x == 3`, provided `x == 0` holds initially:

```
{ x == 0 }
co
   { x == 0 ∨ x == 2 }
   x = x+1;
   { x == 1 ∨ x == 3 }
//
   { x == 0 ∨ x == 1 }
   x = x+2;
   { x == 2 ∨ x == 3 }
oc
{ x == 3 }
```

List the *interference freedom* conditions that must hold in order for this proof outline to be valid, and briefly explain why each of them does or does not hold.

Question 2. Critical Sections and Locks [30 marks]

A course grained solution to the *critical section* problem for two processes has the form:

```
bool in1 = false, in2 = false;

process CS1 {
  while (true) {
    ⟨ await (!in2) in1 = true; ⟩   /* entry */
    critical section;
    in1 = false;                   /* exit  */
    noncritical section;
  }
}

process CS2 {
  while (true) {
    ⟨ await (!in1) in2 = true; ⟩   /* entry */
    critical section;
    in2 = false;                   /* exit  */
    noncritical section;
  }
}
```

(a) [10 marks] Explain how this solution guarantees mutual exclusion, by showing that the property $\neg(\texttt{in1} \wedge \texttt{in2})$ is a global invariant.

(b) [10 marks] Explain how the `await` statements in this solution can be implemented as a *spin lock* using a *Test-And-Set* instruction.

Explain how this solution can be improved by using a *Test-And-Test-And-Set protocol*.

(c) [5 marks] Explain why the spin-lock solution if not *fair* (assuming a weakly fair scheduler).

(d) [5 marks] *EITHER:*

Briefly describe one of the following fair solutions to the critical section problem: the *Tie-Breaker Algorithm*, the *Ticket Algorithm*, or the *Bakery Algorithm*. Explain how this solution guarantees *mutual exclusion*, and how it guarantees *fairness*. (Make sure you state clearly which algorithm you are describing.)

OR:

Explain how the critical section problem can be solved using *semaphores*. Explain how this solution guarantees *mutual exclusion*, and how a solution based on semaphores can guarantee *fairness*.

Question 3. Monitors and Java [30 marks]

(a) [10 marks] Explain briefly what is meant by a *monitor*, and show how a shared variable can be implemented using a monitor, assuming that only one operation may be performed on the shared variable at a time.

(b) [10 marks] Show how a monitor can be used to control access to a shared variable, assuming that several processes may read the variable concurrently, but that a write cannot be performed concurrently with any other operation.

(c) [10 marks] Briefly describe the facilities in Java that support the implementation of monitors. Explain why nested monitors cannot be used in Java.

Question 4. Asynchronous and Synchronous Communication [30 marks]

(a) [7 marks] Andrews introduces the channel with send and receive primitives as a notation for asynchronous message passing:

```
chan ch(type₁ id₁, ..., typeₙ idₙ);
send ch(expr₁, ..., exprₙ);
receive ch(var₁, ..., varₙ);
```

Define the semantics of the channel (`chan`) and the `send` and `receive` operations.

The channel is a queue that stores messages. Order and integrity are preserved.
Send evaluates the expressions and places the values as a message into the channel. It does not block.
Receive takes the next message out of the channel and places the values in the vars. Receive blocks if no message is available.

(b) [9 marks] Hoare developed CSP as a notation for synchronous message passing and guarded communication.

(i) Define the semantics of CSP's output and input statements:

```
destination!port(e₁, ..., eₙ);
source?port(x₁, ..., xₙ);
```

Destination and source are names of processes. If destination is attempting to send a message on the named port and source is attempting to receive a message on a port of the same name and the types of the evaluated expressions match the types of the vars the values are assigned to the vars. Both statements block until both are available. There is no buffering. (5 marks)

(ii) Explain the operation of the following guarded communication statement:

```
do B1; C1 → S1;
[] B2; C2 → S2;
od
```

Evaluate both/all B_i. If both fail (are false) the loop exits. If at least one guard succeeds (B_i is true and C_i would not cause a delay) then choose one nondeterministically and execute S_i. If no guard succeeds, but B_i is true then block until C_i is ready. (4 marks)

(c) [8 marks] Consider a simple server that provides two distinct operations, for example `get` and `put`.

(i) Outline the body of this server if channels are used for communication.

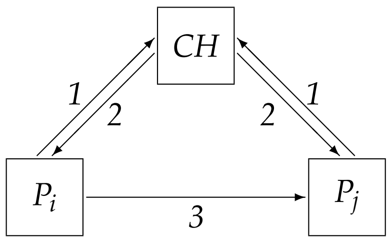
```
while ( true ) {
   receive ch ( kind, ... );
   if ( kind == get ) ...
   else ...; # kind == put
}
```

(ii) Outline the body of this server if CSP style guarded communication is used.

```
do source?get → ...;
[] source?put → ...; ]] od
```

(d) [6 marks] Explain the implementation of guarded synchronous communication, assuming that asynchronous communication using channels is available.

The answer should be based on the use of a clearinghouse and based on figure 10.6 of the text.



Each process sends a template to the clearinghouse (1). When the clearinghouse has identified a match it send an "ok" message to both processes. P_i then sends the data to P_j.

Question 5. Remote Invocation [30 marks]

The syntax of a module that supports remote invocation of operations $op_1, \ldots, op_n$ is:

```
module modName
   op op₁(...);
   ...;
   op opₙ(...);
body
   ...  # implementation
end modName
```

(a) [7 marks] The body a module which uses *rendezvous* would use the following basic implementation:

```
process P {
   # initialisation
   while (true)
      in op₁(args) and B₁ → S₁;
      [] ...;
      [] opₙ(args) and Bₙ → Sₙ;
      ni
}
```

Use this example to explain how rendezvous works. Include all key aspects of its semantics.

The in statement provides guarded communication. All service is provided by one process. On each iteration one of the true guards is selected nondeterministically and its corresponding statements executed. If no guard is enabled execution is blocked. The boolean component of the guard can reference the parameters of the op. Because there is only one process there is no requirement to implement concurrency control.

(b) [6 marks] Ada83 provides a form of guarded communication using rendezvous:

```
select when B₁ ⇒ accept E₁(args) do S₁ end;
or ...;
or when Bₙ ⇒ accept Eₙ(args) do Sₙ end;
end select;
```

Explain how Ada83 differs from the guarded communication using rendezvous described in part **(a)**. Describe one situation in which the difference makes an implementation more complicated, and another situation in which it does not.

The boolean cannot examine the args to the entry. This would not impact the bounded buffer solution. However the TimeServer is made more difficult as we saw in homework assignment 4.

(c) [12 marks] Using Andrew's *Multiple Primitives Notation* for remote invocation provide an implementation of the Readers and Writers problem that gives priority to waiting Writers over waiting Readers. In other words provide the implementation of the following module:

```
module ReadersWriters
  op read(result result types);
  op write(value types);
body
  ...  # implementation
end ReadersWriters
```

From figure 8.13.

```
module ReadersWriters
  op read(result result types);
  op write(value types);
body
  op startread(), endread();
  storage for data;

  proc read(results) {
    call startread();
    read the data;
    send endread();
  }

  process Writer {
    int nr = 0;
    while (true) {
      in startread() and ?write  0 → nr = nr + 1;
      [] endread() → nr = nr - 1;
      [] write(values) and nr == 0 →
        write the database;
      ni
    }
  }
end ReadersWriters
```

(d) [5 marks] Explain your choice of remote invocation primitives used to answer question **(c)**.

We want to allow concurrent readers so a rendezvous would be inappropriate – it would restrict us to a single reader. On the other hand we do need mutual exclusion when updating nr and writing. Thus we use rendezvous for these operations while using rpc for the read op.

Question 6. Process Interaction Paradigms [30 marks]

(a) [5 marks] Define both the heartbeat and probe/echo process interaction paradigms in a way that clearly shows the similarities and differences between the two.

Both paradigms involve the exchange of information amongst processes. In the heartbeat paradigm the information exchange is restricted to a process and its immediate "neighbours". In the probe echo paradigm exchanges may be forwarded (probing) from one process to another, results collected and returned (echo).

(b) [12 marks] Explain how the pipeline algorithm can be used for square matrix multiplication. You are not required to give a full implementation including all of the detail in the text or covered in lecture. Provide sufficient detail to indicate how the process interaction is structured, the task performed by each process and how concurrency is achieved.

There is a manager process that starts the computation and collects the result and n worker processes for an n x n matrix. The workers are arranged in a pipeline that starts and finishes with the manager.

Assume we are computing $C = A \times B$. The manager starts by sending A through the pipeline a row at a time. Each worker keeps the first row it receives and sends the remaining rows down the pipeline. As a consequence each worker has a unique row.

The manager then sends B through the pipeline a column at a time. Each worker multiplies the column by the row it holds to compute one element of C and passes the column of B to the next in the pipeline. Once the worker has processed all columns of B it will have computed a row of C.

Finally the worker receives preceeding rows of C from the pipeline, passes them on and then sends its row. The manager receives C row by row.

The concurrency occurs because each worker can be multiplying its row of A by one column of C. Each worker has a different row and column.

(c) [5 marks] Define the *happened before* relation and state any important properties.

We say event a "happened before" event b if either a and b are events in the same process with a occurring before b or a is the sending of a message and b is the receipt of the same message. Happened before is transitive.

(d) [8 marks] A set of interacting processes, P_1 to P_n, need to implement logical clocks to enable them to order events. State what is meant by a *logical clock* and give the rules that each process needs to follow to successfully implement a logical clock.

A logical clock is simply an integer that monotonically increases. Each process increments the clock when an event occurs within the process. When a process sends a message it timestamps the message with its logical clock and then increments the clock. When a process receives a message it compares the timestamp with its logical clock and selects the maximum plus one for the time of arrival. It then sets its clock to this value plus one.

****