UNIVERSITY OF OTAGO EXAMINATIONS 2016

COMPUTER SCIENCE

Paper COSC441

CONCURRENT PROGRAMMING

Semester 2

(TIME ALLOWED: THREE HOURS)

This examination comprises 4 pages.

Candidates should answer questions as follows:

Candidates must answer **5** questions.
All questions are worth 20 marks, and submarks are shown thus: (5)
The total number of marks available for this examination is 100.

The following material is provided:

Nil.

Use of calculators:

No calculators are permitted.

Candidates are permitted copies of:

Nil.

1. **Parallelism, concurrency, and distribution**

Explain the similarities and differences between *parallelism*, *concurrency*, and *distribution*. Your discussion should address at least the following points:

- What is the purpose of parallelism?
- What issues does parallelism add to sequential programming?
- What is the purpose of concurrency? Why was concurrency important even on single-core machines?
- What issues does concurrency add to parallelism?
- What is distribution?
- What are some reasons for using distribution?
- What issues does distribution add to concurrency?
- How do distributed computations find each other?
- How is dealing with failure different in these models? (20)

2. **Deadlock**

(a) What is deadlock? (2)

(b) You might try to deal with deadlock in a system by killing one or more of the deadlocked threads. If one of the threads you kill is holding a lock on a data structure, what might go wrong? (3)

(c) Sketch a small shared-memory program with two threads where a deadlock might occur. (4)

(d) Explain one way to avoid deadlocks. (4)

(e) Sketch a small message-passing program with two threads where a deadlock might occur. (4)

(f) Imagine two UNIX programs exchanging messages through UNIX pipes. Can they deadlock? If so, how? If not, why not? (3)

3. **Message passing and happens-before**

(a) In Newtonian physics we have the idea of time advancing linearly and regularly, the same for every place. What goes wrong with this notion of time in distributed programs? (4)

(b) Since we don't have a total temporal order, we use a partial order called *happens-before*, between *events*. What are some things that might count as events? Define happens-before. (6)

(c) Briefly describe the *message-passing* approach to concurrent programming. Explain how this provides both *data* and *time* information. Explain the difference between *synchronous* and *asynchronous* messages. (6)

(d) Leslie Lamport showed how to derive a total ordering on events that is consistent with happens-before. How? (4)

4. **POSIX threads**

(a) Explain POSIX or C11 *threads*. What is a thread? How is it like a POSIX process? How is it unlike a POSIX process? What is the *stack* and how do you know how big to make it? What does it mean for one thread to *join* another, and why do you need to? (5)

(b) What is a *mutex*? Write a small function to add an element to the beginning of a linked list, using a mutex to make it thread-safe. What might happen without the mutex? (5)

(c) What is a *condition*? How are conditions related to *monitors*? (5)

(d) What is a *bounded buffer*? What are bounded buffers used for? How is a bounded buffer like a POSIX *pipe*? (5)

5. **Memory and multicore**

(a) Briefly explain the *shared memory model*. What is good about it? What is bad about it? (3)

(b) What is a *data race*? (2)

(c) Briefly explain the *memory hierarchy*. Why does it matter that the memory stores done by one CPU might appear to other CPUs to occur in a different order? (5)

(d) C and Java have a `volatile` keyword. What problem does it solve and when should you use it? (2)

(e) What does it mean for loads, stores, or any other operation to be *atomic*? (1)

(f) What does a Compare-And-Swap instruction do? How could you use it to provide atomic updates on floating-point variables? (2)

(g) What is the A-B-A problem? (2)

(h) Suppose you have a `Vector<T>` class with *atomic* `get(index)` and `set(index, element)` methods, and you need to atomically increment an element. Would

```
vector.set(index, vector.get(index) + 1)
```

work? If so, how? If not, why not, and what could you do about it? (3)

TURN OVER

6. Actor model

(a) Describe the *actor model* of concurrent programming. Your answer should mention its relationship to object-oriented programming and the assumptions it makes about message delivery. (8)

(b) Discuss whether the actor model is better or worse for distributed programming than the shared memory model. Does the actor model handle failure well? (6)

(c) Briefly describe some similarities and differences between Erlang and the actor model. In particular, how do they handle failure? (3)

(d) What are *timeouts*? How are they relevant to failure in distributed systems? Is there a straightforward rule you can use to set timeouts? If so, what? If not, why not? (3)

7. Erlang

(a) In the client-server model, the clients have to find the server. How does Erlang's *registry* deal with this? (2)

(b) Erlang's registry is effectively a global shared mutable dictionary with atomic updates. What problem does having a single global registry create? What might you do about it? (3)

(c) Erlang ensures that no process can change another's variables by not having C-style mutable variables at all. Why does this also help in writing exception handlers? (3)

(d) What is *selective receive* and what is it good for? (3)

(e) What is a *supervision tree* and what is it good for? (3)

(f) In Erlang, there are no syntactic differences between sending a message to another thread on the same machine and sending a message to thread on a different machine. There are, however, some important *pragmatic* differences. What are some of them? (3)

(g) What is the *end to end principle* in system design, and what, if anything, does it have to do with the fact that Erlang message passing does not include flow control? (3)

END