Concurrent & Distributed Systems
Supervision Exercises

Stephen Kell
Stephen.Kell@cl.cam.ac.uk

November 9, 2009

These exercises are intended to cover all the main points of understanding in the lecture course. There are roughly $1\frac{1}{2}$ questions per lecture, and each question is supposed to take roughly the same amount of time to complete. Don't expect to be able to answer everything. You are advised not to spend more than an hour on any one question – unless you really want to.

Unlike most Tripos questions, most of these can be answered quite briefly, so you should expect to spend more time reading and thinking than you spend writing answers. Be warned that exam questions may expect you to remember incidental facts from the lecture notes which are not covered here.

Criticisms or comments about any aspect of these questions would be very gratefully received, however large or small.

1. *Concurrency in centralised systems*

   Consider a single-core computer running an operating system, and containing the usual I/O devices (disk, network interface, etc.).

   (*a*) Suppose the operating system is a monolithic OS running a single monolithic application which is single-threaded. Write down a list of system components which might exhibit concurrent behaviour with respect to each other.

   (*b*) What is meant by "true concurrency"? What are the components exhibiting true concurrency in the system as described so far? What *other* components exhibit behaviour which, although not true concurrency, is usefully analysed as concurrent behaviour?

   (*c*) Now suppose the application consists of two components: a client and a library. The application remains single-threaded. What features of the interface between client and library might introduce concurrent behaviour between these components? [Put differently: what kind of interface between these would *definitely not* exhibit concurrent behaviour?]

   (*d*) Now suppose the monolithic operating system is replaced with a kernel-based OS, running *two* instances of the application. What additional concurrent behaviour is introduced?

   (*e*) Now suppose the application is rewritten to become multi-threaded, where data structures are shared between cooperating threads. What additional concurrent behaviour is introduced?

   (*f*) Now suppose the hardware is upgraded to a multi-core CPU (or multiple single-core CPUs). What additional concurrent behaviour is introduced? State any assumptions you make in your answer.

2. *Nondeterminism, atomicity and mutual exclusion*

   (*a*) What is *nondeterminism*? How does nondeterminism arise in concurrent systems?

   (*b*) What is a *race condition*? Give an example. Are race conditions always a problem?

   (*c*) What is *atomicity*? Give an example of an operation performing memory access which most CPUs can perform atomically, and another which most CPUs cannot perform atomically.

   (*d*) What is *mutual exclusion*? Give one example of mutual exclusion in a computer system, and one example from some other kind of system.

   (*e*) What class of problem can *mutual exclusion* avoid in a computer system? Illustrate your an example with two source code snippets (in any language, or pseudocode): one suffering a problem, and the other solving that problem with mutual exclusion.

3. *Condition synchronization*

   (*a*) What is *condition synchronization*? Give an example of a concurrent system which cannot be correctly implemented with only simple mutual exclusion, but can be implemented using condition synchronization.

   (*b*) A student claims that condition synchronization is always an optimization, because the same effect can always be achieved with *busy-waiting*. What is *busy-waiting*? Is the student correct? Justify your answer.

   (*c*) Sketch a pseudocode implementation of the `wait()` and `signal()` condition synchronization primitives, and some code which uses your implementations. You may assume the availability of a mutual exclusion (lock, mutex) primitive and a *non-threadsafe* queue implementation.

4. *Deadlock and livelock*

   (*a*) What is *deadlock*? Give one example of deadlock in a computer system, and one example from some other kind of system.

   (*b*) What is *livelock*? Give one example of livelock in a computer system, and one example from some other kind of system.

   (*c*) Explain the meanings of *safety* and *liveness*.

   (*d*) Describe a race condition which can lead to deadlock.

5. *Threading models in operating systems*

   (*a*) Threading in user processes on kernel-based operating systems is achieved by either *user-level* or *kernel-level* schemes. Briefly state some advantages and disadvantages of the former with respect to the latter.

   (*b*) Consider a *non-preemptive* user-level threading implementation. Why might such a system suffer from very poor performance?

   (*c*) Can user-level threading be *preemptive*? Justify your answer.

   (*d*) At any given time in a kernel-based threading system, many threads may exist which are not part of any user-level process. Suggest *two* functions which such threads might be performing.

   (*e*) Most CPUs allow *disabling of interrupts* when running in supervisor mode. Why is this useful? What happens when the CPU has disabled interrupts but a device wishes to raise an interrupt?

6. *Programming with semaphores*

   (*a*) Describe an interface syntax and semantics for a *semaphore*.

   (*b*) What CPU support is required to implement semaphores efficiently?

   (*c*) Sketch a machine-level implementation of a semaphore, using the support your described in the previous part.

   (*d*) Describe *two* features provided by mutexes as implemented in the Java programming language which are not provided by a simple semaphore used as a mutex.

   (*e*) A lecturer claims that a semaphore may be used to implement *n-resource allocation*. Give an example of n-resource allocation. The lecturer also states that a mutex is not sufficient. Is the lecturer correct? Justify your answer.

7. *Semaphores and other programming language features*

   (*a*) Consider an n-slot buffer with exactly one producer and one consumer. One student suggests an implementation using two semaphores. Another student suggests an implementation using two monitors [where each monitor contains one condition variable and provides *enter* and *exit* operations executing under mutual exclusion]. Sketch pseudocode for both producer and consumer, in each version side-by-side.

   (*b*) What changes are required in the case where there may be multiple producers or multiple consumers?

   (*c*) What code would be greatly reduced in complexity by using an *active object* to manage the buffer? Where does the complexity go? What additional code must be written, and what extra flexibility might be gained by refining this code?

   (*d*) What conditions would cause *priority inversion* to occur in the above examples? Describe a scheduling policy which will avoid priority inversion.

8. *Lock-free programming*

   (*a*) Explain, using an example, how a lock-free programmer would atomically perform an *in-place update* of a single word of memory. You must assume that the word's new value depends on its previous value, and that the word is shared between multiple threads performing similar updates. [Hint: consider the example of a shared counter being incremented by multiple threads.]

   (*b*) How can this technique be extended to perform atomic operations over multiple words?

   (*c*) Write pseudocode for deleting a node in Tim Harris's lock-free linked list implementation described in the notes.

   (*d*) A programmer is implementing lock-free updates to a shared tree structure. Tree nodes are allocated and deallocated manually on the heap, and are immutable. He finds that his code occasionally suffers from lost updates, but these go away when modifying his code to run on a garbage-collected heap. What might be the cause of these lost updates, and how could the problem be fixed? [Hint: consider replacing a tree node and freeing its memory. This is a hard question.]

9. *Inter-process communication*

   Communication channels and shared storage cells are both examples of resources which may be shared by concurrently executing processes.

   (*a*) Contrast *message-passing IPC* with *shared-memory IPC*. [It may be helpful to consider the following criteria: control flow behaviours, data flow behaviours, message content, message context.]

   (*b*) A graduate student claims that message-passing IPC is a special case of shared-memory IPC. Is he correct?

   (*c*) Another graduate student claims that shared-memory IPC is a special case of message-passing. Is he correct?

   (*d*) What is the difference between *synchronous* and *asynchronous* message passing? In what situations can asynchronous degenerate into synchronous? Name an implementation technique for asynchronous *receipt* of messages.

   (*e*) What does it mean to send or receive a message in a *non-blocking* fashion? How does it relate to sending or receiving a message *asynchronously*?

10. *Erlang, Kilim, Linda and more*

(*a*) Erlang and Kilim are both message-passing systems. Briefly describe their differences.

(*b*) Linda's tuple spaces are shared storage areas. How do they differ from conventional shared memory?

(*c*) Some programming languages provide a feature called a *future*. Describe futures briefly. [You'll have to read beyond the course notes!] How are they similar to an Erlang thread? How are they similar to a Linda tuple?

11. *More on deadlock*

(*a*) What are the four requirements for deadlock?

(*b*) Describe a deadlock where the CPU is one of the contended resources. What scheduling policies are necessary for this to occur?

(*c*) Draw an object allocation graph for a deadlocked set of *four* dining philosophers.

(*d*) Draw the object allocation and object request matrices for the set of dining philosophers you drew.

(*e*) Step through the deadlock detection algorithm (as described in the notes) on your matrices. [You should not have to redraw the matrices for each iteration, only the working vector and row marks.]

(*f*) Why can this deadlock detection algorithm not, in general, be used to *statically* show that a program is deadlock-free?

12. *Transactions defined*

(*a*) What, abstractly, is a transaction? For what sorts of systems is a "transaction" a useful abstraction?

(*b*) Define the ACID properties of a transaction. Which property relates to persistent storage?

(*c*) What is *serializability* with regard to transactions? How is it different from *serial execution*?

(*d*) What is *commutativity* with regard to operations contained within transactions? [Hint: there is a direct relation to like-named the concept in mathematics, but with *operands* and *operator* reversed.]

(*e*) What are the two possible outcomes of a transaction's execution?

(*f*) Draw *four* possible execution history graphs for transactions T1 and T2 on slide 29. [An execution graph is a graph like those on slide 30, where nodes represent atomic primitive operations and edges represent "happens before".]

(*g*) For each of your four execution history graphs, draw a transaction serialization graph. Which graph or graphs represent serializable executions?

13. *Transactions implemented: atomicity and isolation using locks*

(*a*) What is the simplest possible *locking*-based implementation of transaction isolation? Why is it often not good enough?

(*b*) Describe an example where a naive *per-object locking* implementation would permit non-serializable executions.

(*c*) What is *two-phase locking*? How does it address the problem with naive locking?

(*d*) What is *strict isolation*? What additional constraints on two-phase locking can make it *strict*?

(*e*) What problems can be observed using non-strict isolation but which are prevented under strict isolation? Why might a system nevertheless be better off using non-strict isolation?

(*f*) Suggest one technique for avoiding deadlock in two-phase locking, noting any difficulties you foresee in implementing this.

(*g*) Consider the problem with two-phase locking described on slide 36 (in the hand-out "Transactions: composite operations on persistent objects"). Is it really a special case? Describe a constraint on concurrent executions which would solve the problem while still allowing process P's execution to be interleaved with that of process Q.

14. *Transactions implemented: atomicity and durability*

(*a*) What is a *fail–stop* model of crashes? Suggest an alternative model.

(*b*) What is *idempotency*? Why is it a useful property for recovery from failure? Why can't all operations be made idempotent?

(*c*) What is a *write-ahead log*? Using an example which details the step-by-step execution of a transaction, describe how you would use a write-ahead log *both* during normal execution *and* during recovery from a crash.

(*d*) At fail time, a transaction may be in two states: still executing, or finished. What must happen to transactions which were still executing? How can we separate out these transactions from completed transactions when processing the log at recovery time?

(*e*) A simple implementation of write-ahead logging might synchronously write a log record to disk before performing each primitive update. Why is this preferable to synchronously updating objects on disk?

(*f*) Is it safe to buffer log records in memory while a transaction is executing? What about after the transaction has committed? [Put another way: what is the weakest possible constraint on when log records must be written to disk?]

(*g*) Most databases include a *checkpoint* mechanism. When a checkpoint operation is completed, cached log records are flushed to disk and a special "checkpoint record" is appended to the log. What else occurs during a checkpoint? Briefly state *two* potential benefits of checkpointing.

(*h*) Consider a window of time immediately preceding a crash *and containing a checkpoint*.

(*i*) Transaction A committed after the checkpoint, and subsequently all its object updates were written to disk before failure. Would the recovery process need to consult the log records for transaction A?

(*ii*) Consider a transaction which completed all its updates before the crash, but did not commit. Can we use the log to re-do this transaction and treat it as committed?

15. *Transactions implemented: atomicity and isolation without locking*

(*a*) Describe the basic operation of *timestamp ordering*.

(*b*) Consider a transaction which visits several objects and increments an integer field in each. The transaction is processed using timestamp ordering, supporting read and write operations only. Describe a second transaction which, if it arrived during the execution of the first transaction, would cause the first to abort, despite the existence of a serializable execution schedule.

(*c*) What are *two* ways in which the approach of *optimistic concurrency control* differs from approaches discussed so far?

(*d*) A student suggests that OCC offers poor performance because shadowing imposes large overheads. Suggest an implementation technique which might mitigate these overheads.

(*e*) An engineer is not sure whether to use timestamp ordering or optimistic concurrency control in a new persistent object management system. Suggest, with explanations, a few criteria which would help the engineer decide.

16. *Optimistic concurrency control: detail*

Consider a single-threaded validator in a centralised OCC commit manager.

(*a*) What is the distinction between "validated" and "committed" transactions?

(*b*) How do *object* timestamps relate to *transaction* timestamps?

(*c*) How does the validator determine whether it can commit an executed transaction? [Your answer should mention both *shadow copies* and *conflicting operations*. It may help to draw a table.]