Concurrent and Distributed Systems — 2024/25

Examples Sheet CS1

Q1 LL/SC FSM Diagram

How might you show compare-and-swap or LL/SC interactions using an abstract FSM view? Draw diagram for one thread. Two threads attempt to acquire a lock using LL/SC: using a few words, describe features you would observe in the product.

Q2 Philosophers FSM

Draw out the state space for 2 or 3 dining philosophers. Do you see any dead states?

Q3 Semaphore Unblocking Order

Sketch a small piece of code that will report on the queuing discipline for threads blocked on a mutex or semaphore. The 'discipline' is a constraint on the relative ordering of blocking and unblocking of threads. Possible disciplines include FIFO, LIFO and random. The code might print 'in order', 'reverse order' or 'random' or something like that at the end of its run. What would we expect if the threads had differing priorities?

Q4 Recursive Locks

Part of a recent Tripos question discussed what happens if one thread tries to acquire the same lock twice without releasing it. What might happen in a simple system without special support for this?

Harder: what support might be added and why might this be useful?

Q5 Producer/Consumer Relaxation

Can we modify the generalized P-C solution, as lectured (page 18/77), to allow concurrent access by 1 producer and 1 consumer by adding one further semaphore? A friend thinks you need a single lock, as originally presented, and comments 'Surely, you should never have a producer and a consumer active at the same time?' Are they making a valid point?

Q6 Monitors

(a) What is the primary invariant ensured in a monitor?

(b) How can a thread block itself inside a monitor while still allowing other threads to come in? Why might it want to do this?

(c) Does a condition variable hold a value? If not, why do we have them and how many do we need? Why shouldn't a thread just repeatedly test a normal shared variable to poll for a condition?

(d) A scheduller resumes a thread when it is ready to run. Would it be helpful if a user could give the sheduller an arbitrary predicate that says whether a thread is ready? What could go wrong? What is commonly made available?

Q7 MRSW Monitor

Sketch an MRSW monitor implementation (L04, slide 24). This monitor is used to provide a lock around a shared resource that enables either one writer or multiple readers to have access. Hint: because monitors cannot have multiple readers concurrently running inside them, your monitor is likely to need separate `pre_read()` and `post_read()` methods that the user code will use to bracket its operations on the shared resource.

(Beginners will find this a very hard exercise. But it is well worthwhile, so I have not put an asterisk.)

Where multiple writers contend, does your solution offer preference to (is fair to) readers and if so does this tend to mean that readers are more often handed out-of-date information? Is this good or bad?

Optional Questions. The remainder of this sheet contains starred questions.

* Star denotes optional/advanced exercise. You are recommended to **not** attempt the starred exercises unless you are sure you have time. Just quickly read through them and discuss with your supervisor any points that pique you.

Q8 Co-Routines (*)

(a) * Concurrency can be provided without a sophisticated scheduller using a co-routine package. A set of co-routines time-share on a voluntary basis by calling yield() which blocks the current thread that will resume again (by returning from yield()) after all other threads have been run. There is no pre-emption. The other essential API call provided by a co-routine package is a create_thread(void (*f)(void)) which is passed the address of a C routine (that normally contains an infinite loop whose body contains calls to yield()). [The syntax for passing a function to a function in C is a little off-putting, but should eventually be familiar to those reading the C course.] Sketch or describe the code for a co-routine package, being clear about the central datastructure needed and how the stack pointer should be handled. If you want to do a full C implementation, you should first be familiar with longjump. *This exercise should be easier after reading Ib Compiler Construction.*

(b) * As mentioned on the previous sheet, early versions of Windows provided co-routines as the only concurrency mechanism. What problems did this lead to? For what application scenario might it be common to use co-routines today?

Q9 LL/SC Behaviour on Context Switch (*)

* Consider why an on operating system based over the LL/SC mechanism should issue an instruction during a context switch to clear any pending LL that has not been cleared by an SC.

Q10 Threading primitives (*)

(a) * Can several threads be running the same piece of code on a given computer if they are written in a language that has no intrinsic support for parallel programming? Describe various ways how.

(b) * Alice says the C language does not have any threading primitives, but Bob says the volatile keyword demonstrates threading support. Prof Mycroft says that thread-local statics can be found in recent versions of C++. Who is correct? [Thread-local statics were not lectured in Concurrent Systems but perhaps in C++.]

(c) * For discussion with supervisor: Can language-level support for concurrent programming make code more adaptable to running on platforms with varying numbers of cores and highly non-uniform memory access time?