CptS 483/580
Concurrent Programming
Final Exam (Take-home)

April 30, 2010

Instructions: This take-home exam is intended to give you a chance to show off your knowledge by applying it to some problems. The problems don't necessarily have right or wrong answers – rather what I want to see is the way you approach the problem. It is important to be accurate in what you say and also to avoid just giving a memory dump of what you know or can find out about a particular mechanism. What you write should be relevant to the particular exam problem. I have given an example problem and an answer that exhibits the kind and degree of detail that I'm looking for in your answers.

Answers must be typed – no handwritten answers.

Email the solution to me (hauser@eecs.wsu.edu) before 11:59:59 PM on Tuesday, May 4th as a PDF, .odt, or .doc file. If you need to do drawings you may sketch them neatly by hand, and scan (or digitally photograph) them and send them as an attachment to the email – they can be a separate attachment as long as I can easily identify to which answer they belong. I suggest setting the "return receipt requested" option on this email. If you do not hear from me that I have received your exam by Wednesday 5PM let me know so it can be tracked down.

If you have questions or need clarification: email me.

Example problem

In the synchronous communication assignment part 2 you were encouraged to use a single lock per channel rather than a lock for each of the two queues associated with the channel. Perhaps more concurrency could be attained if a separate lock were used for each of the queues. Analyze the feasibility of the per-queue lock approach and discuss under what circumstances such a solution would be beneficial.

Example answer:

Feasibility: the synch operation at a high level performs:

```
if (there is a waiting event of the opposite kind) {
    transfer the message object between the two events
    wake up the waiting other thread # see Problem 2
} else {
    enqueue this event on its own queue
    wait to be woken # see Problem 2
}
```

The if-then part is a check-then-act sequence that needs to be protected by a lock on the "opposite-kind" queue. The if-else is a check-then-act requiring a lock on the opposite queue, but the enqueue operation on the "own-kind queue" requires protection against concurrent updates by readEvent and writeEvent synch operations. If the "own-kind queue" is protected by a separate lock then both locks must be held during the if-else part. Since both locks must sometimes be held simultaneously they must always be acquired in the same order to avoid deadlocks. If the code simply acquires both locks every time we haven't gained (and have actually added overhead) relative to using a single lock for the channel. We might try something like this, however:

```
acquire opposite-kind lock
if there is a waiting event of the opposite kind {
    transfer the message object between the two events
    wake up the other thread
    release opposite-kind lock
    return
} else {
    release opposite-kind lock
    acquire send-kind lock
    ## note that with more careful coding the above two operations
    ## could be omitted in the case that the
    ## send-kind lock is the one already held
    acquire recv-kind lock
    if there is a waiting event of the opposite kind {
       transfer the message object between the two events
       wake up the other thread
    } else {
       enqueue this event on its own queue
    }
    release both locks
}
```

Benefits: This solution, if it works, trades off contention for a single channel-lock for additional work in every synch operation. Completing both sides of a successful communication requires one time through the if-then part which acquires and releases a single lock *and* one time through the if-else part which does 3 lock acquire and releases (2 sometimes if the optimization alluded to in the ## comments is done). So where with the first, simple solution we had a total of two lock acquisitions we now require at least 3 and sometimes 4. What do we gain for this extra code complexity and run-time cost? We were trying to reduce contention, but the things that now acquire only one lock (send with recvr waiting or receive with sender waiting) can't really happen very much so the acquire-three-locks path has to occur in equal numbers with the acquire-one-lock path through the code. The acquire-three-locks path suffers at least as much contention as in the original simple solution. Therefore, I don't see any particular benefit to the proposed approach.

Problem 1 (moderate)

We have previously noted that various primitive operations in the Erlang language pose low-level implementation problems of synchronization and visibility that they hide from the Erlang programmer. Consider for example the implementation of the registration-related functions (pp. 146-147) that allow atoms to serve in place of process ids for sending messages, *etc*.

Part 1: suggest data structures to use for implementing these functions and tell how accesses need to be synchronized using locks. Use pseudo-code like in the example answer above – the answer here is *not* Erlang code;

Part 2: Analyze the feasibility of using an RCU strategy for implementing these operations and discuss under what circumstances such a solution would be beneficial. As in the example problem solution above, "analyze the feasibility" includes suggesting algorithms at the level of pseudo-code for critical parts of the operations.

Problem 2 (easy)

In the synchronous communication project, Java's intrinsic synchronization mechanism was fairly unsatisfactory because there is no way to wake up a specific waiting thread (see the high-level description of the synch operation in the solution to the example problem above). Instead, where the pseudo-code says "wake up the waiting other thread" we find we need to use notifyAll() waking up all the waiting threads all but one of which wake up and discover that they just need to go back to sleep. Using notify() is inadequate because we have no control over which of the waiting threads wakes up. Section 14.1 in the Java textbook discusses the general problem of waiting for a particular state to be achieved and the use of intrinsic synchronization to achieve it (using wait(), notify() and notifyAll()). Section 14.2 then talks about ReentrantLock objects and their associated Condition objects as a more-flexible generalization of Java intrinsic synchronization. Using ReentrantLock and Condition objects show how to avoid the "wake up all the threads" problem in the single-channel synchronous communication outlined in the example problem solution above. (Base your answer on the single-lock-per-channel solution.)

Problem 3 (easy)

Database transactions were invented to provide four properties – the so-called ACID properties. Which of these properties are NOT present in the transactional memory solutions that we discussed? Explain the rationale for omitting it (or them).

Problem 4 (hard)

For the full implementation of the synchronous communication project, using select, I recommended that you use a single, global lock to avoid serious problems with lock acquisition order and achieving the required atomicity across multiple poll and enqueue operations. Using a single lock here is quite unfortunate in that it causes communication operations that have nothing to do with one another to contend for the same lock. For example if threads A and B communicate with each other while C and D also communicate, nevertheless they all contend for the same lock. Analyze the feasibility (by suggesting an implementation strategy as above) of using either the DSTM approach that I discussed or the timed STM that Cewei presented to create a synchronous communication implementation that does not require a global lock. (This is a hard problem – what I want to see in your answers is the story of how you think through the capabilities of the STM mechanism, how it might apply, how far you can get, and what is hard to address.)