Part 1 – Multi-instruction Atomics

Locks can be difficult to use correctly. An alternative approach is to use "transactional memory", where a set of memory reads and writes are only observable as an atomic transaction. There are a variety of implementations that are both software and hardware based.

Load Linked / Store Conditional (LL/SC):

Load Linked – Load a memory location into a register
All further memory access is disabled until SC is called

Store Conditional – Stores a value to a memory location
Can only store to the location accessed via LL
Only store to the location if no other processor has accessed it since LL, else abort

Q1) How does using LL/SC result in updating a memory location as an atomic transaction?

The location can only be updated if no other processor has accessed it. Therefore, without that interleaved access, it is as if the load and store occurred atomically.

Q2) Using LL/SC, provide an implementation of CMPXCHG.

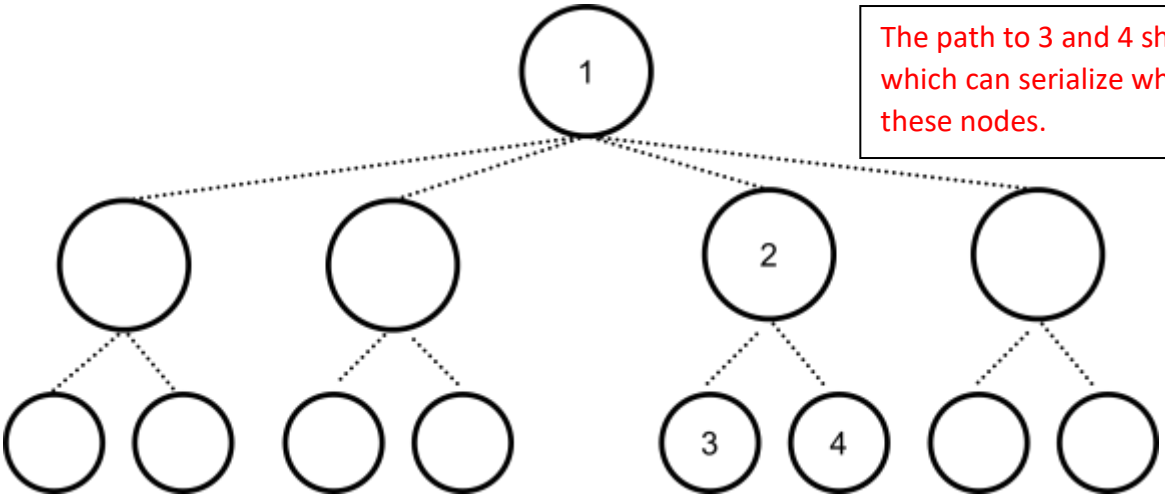
(see exercise 3 solution)

Part 2 – Transaction Problems

Q3) Based on Wednesday's lecture, how do we protect concurrent accesses to a binary tree?

Fine-grained aka "hand over hand" locking.

Q4) Given the following tree, how is concurrency impacted by Q3 when trying to update nodes 3 and 4?



The path to 3 and 4 share 1 and 2, which can serialize while traversing these nodes.

Q5) If threads 0 and 1 call transfer at the same time, what is one possible result?

```
void transfer(A, B, amount)
{
   synchronized(A) {
     synchronized(B) {
       withdraw(A, amount);
       deposit(B, amount);
     }
   }
}
```

Thread 0:
`transfer(x, y, 100);`

Thread 1:
`transfer(y, x, 100);`

Deadlock. Thread 0 has synch'd on x, while 1 is synch'd on y. Now neither can make progress.

Q6) What is one approach to address the problem in Q5?

Enforce a lock ordering. For example x < y.

Part 3 – Hardware Transactional Memory

Intel provides transactional memory support in the Haswell processor, called TSX. TSX provides the ability to track some quantity (proprietary) of cache lines and expose the updates made to all of the tracked lines atomically.

Q7) What coherence state(s) may likely represent a cache line that is being tracked for LL/SC or by TSX?

The M state most likely represents a tracked cache line.

Q8) If another processor accesses the tracked cache line, the transaction should be aborted. Based on Q7, when how does the existing cache coherence protocols support this detection?

All coherence requests for a tracked cache line will require that line to be downgraded from M to either S or I in MESI.

Q9) With TSX, it is permissible for another thread to read a line that has already been read as part of a transaction. How does this change your answers to Q7 and Q8? Will a processor executing a transaction see this read?

Since it is possible for multiple readers to access the same cache line in a transaction, lines that are part of a transaction's read set can instead be tracked in the S state.