CSE 613: Parallel Programming

Lecture 18
(A Brief Introduction to Transactional Memory)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2012

Problems with Locking

Priority Inversion: When a lower-priority thread is preempted while holding a lock needed by higher-priority threads.

Convoying: When a thread holding a lock is descheduled (e.g., by page fault) no other thread requiring that lock can make any progress until the inactive thread is rescheduled and able to release the lock.

Deadlock: When threads attempt to lock the same objects in different orders.

Not Easily Composable: Consider, e.g., what you need to do in order to dequeue an item from q_1 and enqueue it to q_2, and make sure that the transfer is atomic.

Hard to Manage: How do you manage a large number of locks in a large system?

Problems with compareAndSet( )

compareAndSet( ) works only on a single word. This restriction often forces a complex and unnatural structure on algorithms.

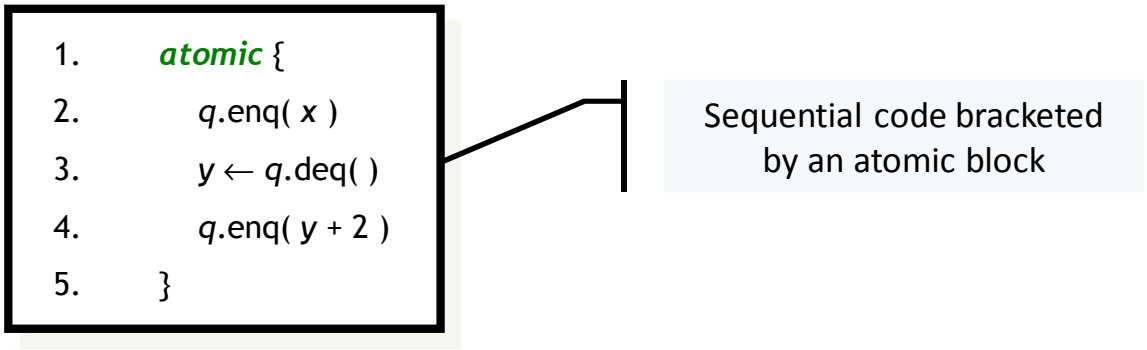
Recall the half-finished *enq*( ) operation from lecture 17:

```
1.    public void enq( T value ) {
2.        Node node = new Node( value );
3.        while ( true ) {
4.            Node last = tail.get( );
5.            Node next = last.next.get( );
6.            if ( last == tail.get( ) ) {
7.                if ( next == null ) {
8.                    if ( last.next.compareAndSet( next, node ) ) {
9.                        tail.compareAndSet( last, node );
10.                       return;
11.                   }
12.               } else { tail.compareAndSet( last, next ); }
13.           }
14.       }
15.   }
```

If the 2nd CAS fails we return leaving the *enq* operation half-finished.

We sometimes finish *enq* operations left half-finished by others

Memory Transactions & Transactional Memory



Memory Transactions: A sequence of memory accesses made by a single thread with the following properties.

- **atomicity:** either all memory updates appear to take effect at once (*commit*) in the order they appear in the transaction, or none of them takes effect (*abort*)
- **isolation:** updates made by a transaction are not visible to others until the transaction commits
- **serializability:** concurrent transactions on a data structure appear to take effect in some sequential one-at-a-time order

Transactional Memory: Allows transactions for threads running concurrently in a shared-memory environment

Composability, Serialization and Concurrency

```
1.    Transfer( q1, q2 )
2.       atomic {
3.          x ← q1.deq( )
4.          q2.enq( x )
5.       }
```

Composability: Transactions compose easily.

Serialization: *Transfer*(q_1, q_2) and *Transfer*(q_2, q_1) will be serialized.

Concurrency: *Transfer*(q_1, q_2) and *Transfer*(q_3, q_4) will execute concurrently.

Implementation: Optimistic Concurrency

atomic { … ⟨ code ⟩ … }

One possible implementation strategy

- execute ⟨ *code* ⟩ without taking any locks
- each read and write operation in ⟨ *code* ⟩ is recorded in a thread-local transaction log
- writes do not update memory, instead they go to the log only
- at the end, atomically commit the changes to the memory provided there are no conflicts (e.g., no other transaction is reading the memory locations the current transaction is going to update or updating the locations it read)
- if the commit fails, rerun the transaction

Some Limitations

- no I/O inside transactions
- no deadlocks, but livelocks are still possible
- long running transactions may be repeatedly aborted because of committing short transactions
- aborts waste resources (time, energy)
- shared-memory abstraction only

Transactional Memory Implementations

Software Transactional Memory (STM)

- high overhead (often slower than sequential code)
- flexible as no hardware requirements
- examples: DSTM (Java), STM Haskell, C# SXM

Hardware-Accelerated Software Transactional Memory (HASTM)

- hardware support is used to reduce some overheads (e.g., memory access tracking and conflict detection)
- Intel reported almost HTM speed

Hardware Transactional Memory (HTM)

- modern cache-coherence protocols already do most of what is needed to implement TM (e.g., detecting and resolving synchronization conflicts, buffering tentative updates), but still little TM support, if any
- transactions are often bounded by hardware limitations
- examples: SUN ROCK processor (now cancelled), Standford TCC, Intel VTM

Hybrid Transactional Memory (HyTM)

- do HTM, but fall back to STM when hardware limits are reached
- examples: gcc 4.7 TM library (experimental)