Mutual Exclusion and Transactions

ICS432 - Fall 2008

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Future of Mutual Exclusion

- This is a brief lecture on the possible future of concurrent programming for mutual exclusion
- The content is basically digested from an article in ACM Queue, Vol 4. N. 10
  - *Unlocking Concurrency*, by Adl-Tabatabai, Kozyrakis, Saha
  - Link to the article on the course Web site
- The short story:
  - Concurrent programming has become part of everyday life due to multi-core architectures
  - Mutual exclusion is one of the fundamental requirements for concurrency
  - Mutual exclusion is not easy to program so that it's correct, low-cost, and high-concurrency
  - Ideally, the programmer should not have to worry about it and the system underneath should deal with it
  - Transactions are a way to achieve this goal, to some extent

Transaction Memory

- The programmer programs mutual exclusion with locks
  - Or something that boils down to locks, like monitors
- Locks have many disadvantages
  - Can be difficult to achieve correctness
    - No deadlocks, no race conditions, etc.
  - Locking/Unlocking can be time consuming
- Quote from the founder of Epic Games: "manual synchronization .. is hopelessly intractable" for dealing with concurrency in game-play simulation
  - but that's what we do, and what we'll do for the foreseeable future
  - The good news is that it's not always hopelessly intractable
- The "new" idea is that of Transaction Memory (TM) that alleviates the task of the programmer

What is a Transaction?

- The transaction concept comes from the database community
- A transaction is a sequence of (memory) operations that either executes completely (it's committed) or has no effect on the state of the system (it's aborted)
- If a transaction commits, it *appears* as if all its operations happened instantaneously, that is, atomically
  - The stores/writes are not visible until a transaction commits, also a transactions may have multiple such stores/writes
  - Therefore, there are no conflicts with other transactions
- It's all an illusion: Can we build a system with the *concept* of transaction with the above properties?
  - The whole point is for the programmer to reason on the illusion, which is convenient, and for the system to make the illusion happen
  - Just like many other things in a computer system (e.g., virtual memory)

Transactions in Languages

- If we had a system that support transactions, we could stop using locks and just declare sections of code as atomic

```
public class SomeClass {
  Object lock1, lock2;

  public SomeClass() {
    lock1 = new Object();
    lock2 = new Object();
  }

  public void f1() {
    synchronized(lock1) { . . . }
  }
  public void f2() {
    synchronized(lock2) { . . . }
  }
}
```

```
public class SomeClass {

  public SomeClass() {
  }

  public void f1() {
    atomic { . . . }
  }
  public void f2() {
    atomic { . . . }
  }
}
```

Why Transaction Languages?

- The code is shorter, simpler
  - Arguably not a big deal
- The programmer has to make a choice with locks
  - Use multiple locks like in the code before
    - Allows concurrent access to different parts of the object
    - But this ends up being very error-prone in large programs
      - Must be very careful what to lock/unlock and when
    - Must decide how many locks to use
  - Use complex locking to allow multiple readers / single writer
    - We've seen ways to do it
    - In a large program, this compounds the complexity
  - Furthermore, using many locks causes overhead just to call lock() and unlock() often, which ends up being non-negligible in some applications
  - Or on could use only a few locks
    - But this reduces concurrency, and thus performance
- By just declaring sections as "atomic", the system does the hard work, not the programmer

Array Example

- Assume you have an array of integers, and that multiple threads want to read-write elements
- Solution #1: one lock for the whole array
  - poor concurrency
- Solution #2: one lock for each element
  - memory consumption, complexity
- Solution #3: use transactions and put all array reads or writes in a atomic section

HashMap

- A good example / justification for the previous slide is the ConcurrentHashMap class in java.util.concurrent
- The reason that this class is in the package is that it's difficult to write a good thread-safe hash table that
  - Has many locks to allow for maximum concurrency
  - Doesn't have so many locks that overhead is large
  - Is correct in spite of the many locks
- So expert programmers have gotten together to implement the thread-safe ConcurrentHashMap class
- If we had something like transactions, anybody could easily write a thread-safe hash map (or any other data structure), just by annotating the sequential code with atomic sections
  - The benefits of fine-grain concurrency without the headaches

Concurrency and Composition

- Imagine you have two ConcurrentHashMap objects
- You want to move one object from one to the other
- You want for all threads to see your object in either hash table
  - Not in both
  - Not in neither
- This can be done by just locking both hash tables with a coarse-grain lock, doing the move, and then unlocking
  - In the mean time no other thread can access the hash tables, loss of concurrency, loss of performance
- This can be done by fiddling with the actual implementation of ConcurrentHashMap to preserve concurrency
  - Really difficult to do correctly
  - And you don't have access to that code!
- Solution: put the move in an "atomic" section, let the system deal with it
- With transactions, you can now get a bunch of thread-safe objects, do things on them in an atomic section, and still have maximum concurrency!
- Transactions can be implemented via various techniques
  - Hardware TM (HTM), Software TM (STM), Hybrid

Transactions are Great but...

- At this point, anybody would agree that transactions are good
- But we've been assuming that the system underneath can implement them... is this even possible?
- Turns out, the database community has been using transactions for a while
  - To main consistency to databases
  - Think of airline reservation services, etc.
- The way in which it works is (at a high level):
  - Versioning: keep multiple concurrent versions of the "state" of the system for multiple concurrent transactions
  - Conflict resolution: when a transaction tries to commit, check whether it can be done safely, otherwise make the transaction abort
  - Rollback: when a transaction cannot commit, one restores the old version of the state to negate the changes

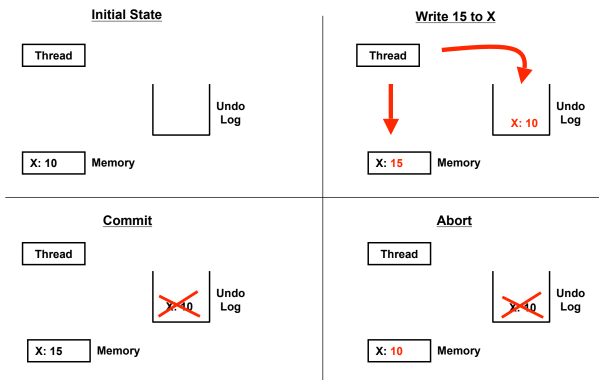
Conflict Resolution

- Conflict resolution is done by looking at the "read set" and "write set" of transactions
  - The set of "things" read
  - The set of "things" written
- When resolving conflicts, a TM system just looks at intersections
  - e.g., if two transactions have intersecting write sets, then one of them is going to be rolled back
- One question: what is the granularity?
  - Object level: similar to coarse-locking
    - If two transactions modify the same object, only one goes through
  - Word-level: great, but costly (many words)
  - Cache-block-level: probably a good compromise

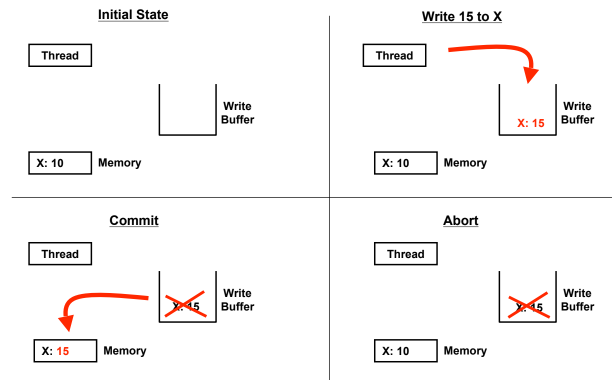
Data Versioning

- Goal: be able to remember old versions of data in case of a rollback
- Two options:
  - Eager (keep an "undo log")
    - Update memory location directly
    - Maintain undo info in a log
    - Good: Fast commit
    - Bad: Slow aborts
  - Lazy (keep a "write buffer")
    - Buffer writes until commit
    - Update memory location on commit
    - Good: Fast aborts
    - Bad: Slow commits

Eager Versioning



Lazy Versioning



Is it Coming, is it Good?

- Many groups in industry, including Intel, are looking at the hardware and software side of transaction memory
  - One of those new “hot” technological trends
  - Sun has announced that its upcoming Rock Processor will support the transaction memory idea
- Doesn’t solve all
  - Still need to find and exploit concurrency
  - Still need to understand what should be in a critical section
- Risks
  - Doesn’t scale well, with tons of “atomic” sprinkled all over the code that prevents concurrency
  - Many heated debates at the moment on whether it’s a good idea at all (touted as a “silver bullet” by some, and as a “disaster” by others)
  - Still too early to truly see the impact

Conclusion

- The article has a great discussion of
  - How transactions could be implemented
    - aggressive versioning / lazy versioning
    - optimistic writes / pessimistic writes
  - Why it’s difficult and not here yet
- You must read this article as an assignment and bring up questions in class if needed
- As programmers in the industry you may see the day when you rely on transactional memory systems routinely, so it’s good to know what’s coming
- But don’t get _too_ excited, you’ll still have to use locks / monitors like we’ve seen in class for the time being
  - Otherwise, I wouldn’t teach it as much :)