Transaction Memory

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Future of Mutual Exclusion

- The content of these lecture notes is inspired by
  - *Unlocking Concurrency*, by Adl-Tabatabai, Kozyrakis, Saha
  - “The Art of multiprocessor Programming”, Maurice Herlihy and Nir Shavit
- 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
    - You should be pretty convinced by now in this course
  - 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

Mutual Exclusion Hell

- The basic approach is to do mutual exclusion with locks, and it's difficult to make programs correct (or easy to debug) and fast
  - Lockfree programming solves performance issues, but if anything requires even more sophisticated/difficult thinking
- Quote from the founder of Epic Games: *"manual synchronization .. is hopelessly intractable"* (for dealing with concurrency in game-play simulation)
- Quote from Herb Sutter, chair of the ISO C++ standards committee: *"Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all."*
- Let's revisit locking a little bit…

Locking in Real Life

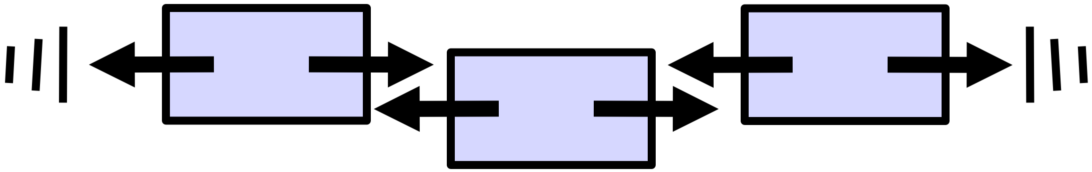
- One "easy" approach is to use coarse-grained locking: just protect your entire code using one lock
  - e.g., you have a tree structure that is traversed and updated by multiple threads
  - Lock the whole "traverse and update" operation
  - While a thread traverses the tree, no other thread can
- This is the easy solution, but it has poor performance
  - One long critical section
- We say that it "doesn't scale"
  - Adding threads/cores won't lead to performance improvements

Locking in Real Life

- The alternative is fine-grained locking: use multiple locks to create multiple shorter critical sections
  - More difficult to develop, debug, validate
  - Real-world Linux Kernel code comment
  - ```
    /*
    * When a locked buffer is visible to the I/O layer
    * BH_Launder is set. This means before unlocking
    * we must clear BH_Launder,mb() on alpha and then
    * clear BH_Lock, so no reader can see BH_Launder set
    * on an unlocked buffer and then risk to deadlock.
    */
    ```
  - When understanding comments becomes more difficult than understanding the code?
    ```

Locking in Real Life

- Consider a doubly-linked, two-ended queue



- Is efficient fine-grain locking feasible?
- Yes, but it is a publishable research result [Michael & Scott, PODC96]
- Question: are we happy with a technology with which writing a concurrent double-ended queue is actually a research problem????
- Waiting for java.util.concurrent to provide these cool solutions is not always possible

Locking in Real Life

- Locks are not "composable"
- Remember Homework Assignment #3: Two thread-safe hash tables, T1 and T2, each protected by its own lock
- We want to move an element, e, from T1 to T2, so that e must always be seen as either in T1 or T2
  - Therefore, T1.remove(e) followed by T2.add(e) doesn't work because any thread could access T1 or T2 in between the two calls and not see e anywhere!
- Solution: acquire T2's lock before calling T1.remove()
  - But T2's lock is supposed to be hidden to developers!
  - This is "breaking the abstraction" and users need either to use their own locks or "see" inside the abstract data type
- There is really no great solution here
- Again, shouldn't this be easy using a "good" technology?

So what?

- Perhaps we’re just doing the wrong thing?
- Could there be a solution that doesn’t require the programmer to spend countless hours solving concurrency problem
  - Intellectually challenging and rewarding
  - But not very productive
- One option is: just do not share any memory state ever (sort of the Erlang philosophy)
  - Share nothing, communicate via messages, and get over it
  - But reasoning about messages can be difficult too
- Another option: Transactions

What is a Transaction?

- The transaction concept comes from databases
- 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
- Can we build a transaction abstraction with these properties?
  - The programmer reasons assuming transactions, and the system makes it happen
  - Just like many other things in a computer system

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 programmer has to make a choices with locks:
  - Coarse-grain or fine-grain?
  - How fine is fine-grain?

- By just declaring sections as “atomic”, the system does the **hard** work, not the programmer
  - A transaction may fail, in which case the user can simply attempt it again

- And the code is simpler to write!

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 atomic sections

HashMap

- A good example / justification for the previous slide is the ConcurrentHashMap class in java.util.concurrent
- The reason for this class 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 (no deadlock)
- Several expert programmers have gotten together to implement the thread-safe ConcurrentHashMap class
  - Which uses CAS for lockfree programming under the hood!
- 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

Composability

- Let’s go back to the “move one element from one hash table to another” example from Homework #3
- This can actually 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 typically!
- Solution: put the move in an “atomic” section, let the system deal with it
- With transactions, you can now get a bunch of objects, do things on them in an atomic section, and still have maximum concurrency!

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?
- Database people has been using transactions for a while
  - To maintain consistency to databases (e.g., airline reservations)
- 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 abort the transaction
  - Rollback: when a transaction cannot commit, restore the old version of the state to negate the changes

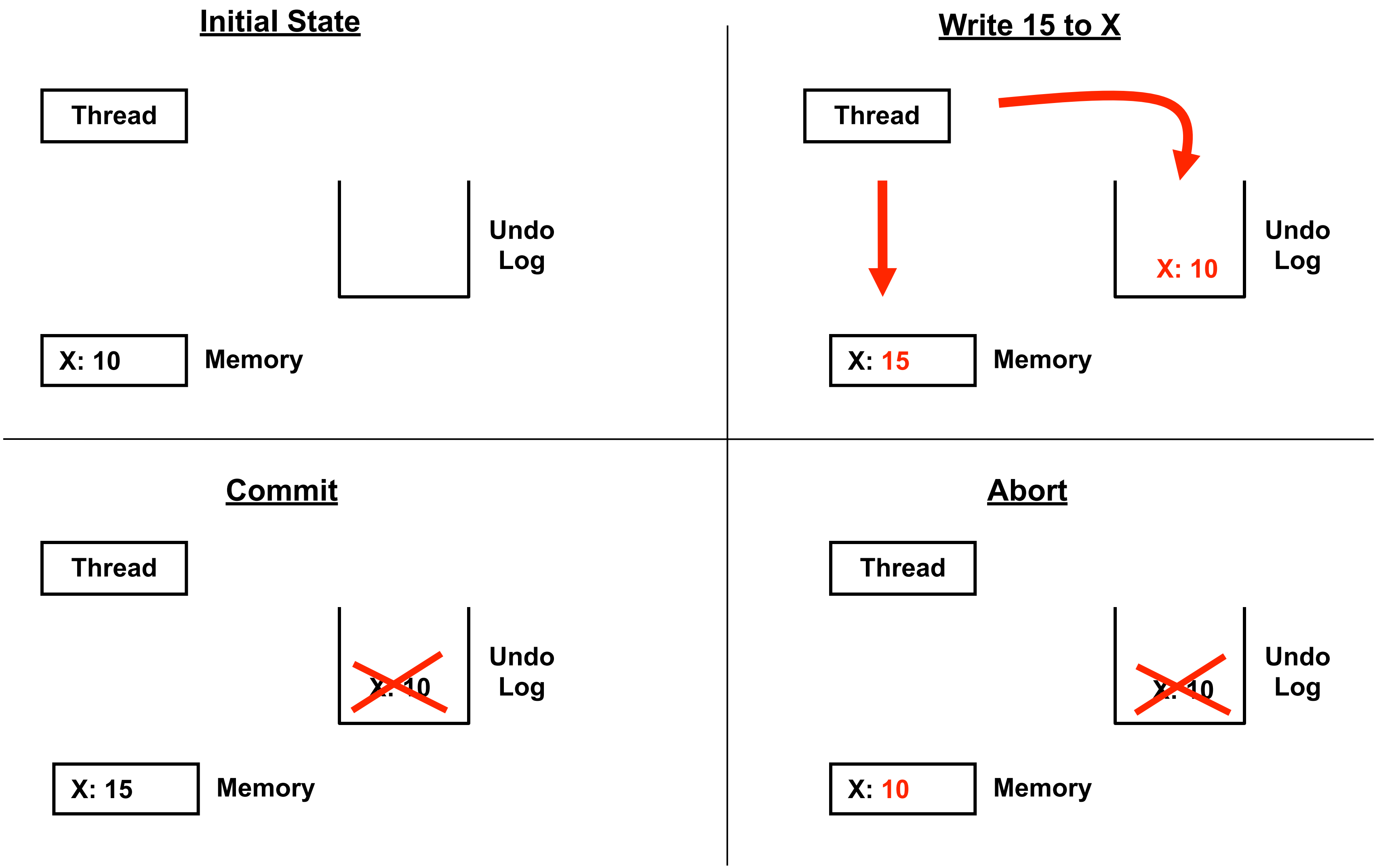
Conflict Resolution

- 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?
  - sets of objects: similar to coarse-locking
    - If two transactions modify the same object, only one goes through
  - sets of bytes: great, but costly (many bytes)
  - sets of cache blocks: probably a good compromise

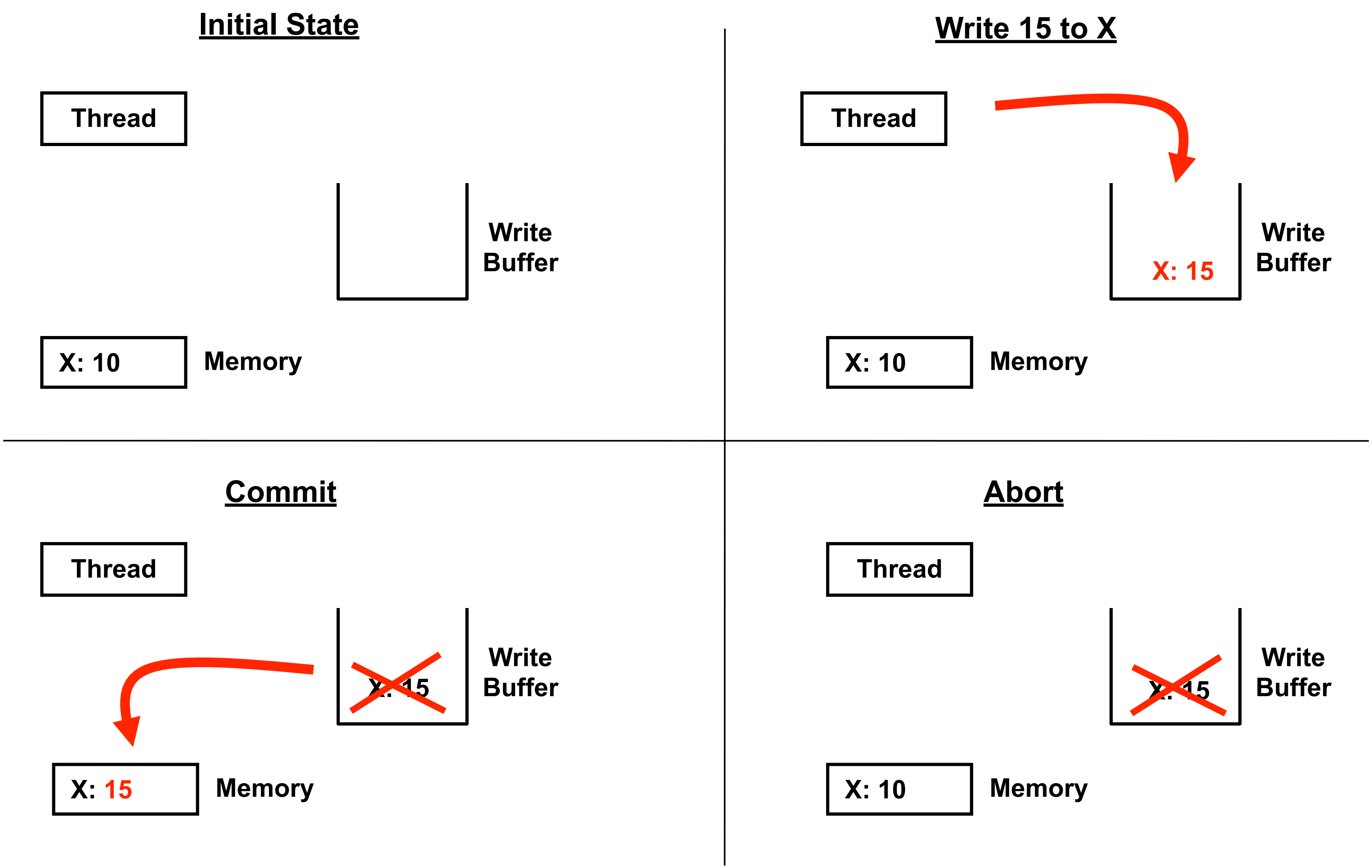
Data Versioning

- 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



Implementation

- Can be implemented in hardware (Hardware Transactional Memory: HTM)
  - Exploits “cache coherence protocols”
    - Turns out that caches in SMP systems do a lot of what’s needed for implementing HTM
  - Fast, but needs hardware resources
- Can be implemented in software (Software Transaction Memory: STM)
  - Slow but can substitute for HTM when it fails
- Studies have shown that transactions are easier to program than traditional locks
  - No surprise there

Is it Coming, is it Good?

- HTM proposed initially in 1993
- Many groups in industry, including Intel, have looked at the hardware and software side of transaction memory
  - Several STM implementations
  - HTM: IBM's BlueGene/Q processor, IBM's EC12 server, IBM's Power 8 processor, Intel's TSX on Haswell and Broadwell processors (but didn't work!) and then on some Skylake processors
- One of those "permanently new" hot technological trends
  - Perhaps it's getting there though…
- Doesn't solve everything
  - Still need to find and expose concurrency
  - Still need to understand what should be in a critical section
  - If many transactions keep aborting, performance is terrible
- Some people think it would lead to a generation of terrible programmers...

Conclusion

- As programmers in the industry you may see the day when you rely on transactional memory systems routinely

- But don’t get too excited (yet)