CS510 Concurrent Systems

Why the Grass May Not Be Greener on the Other Side: A Comparison of Locking and Transactional Memory

Why Do Concurrent Programming?

- Hardware has been forced down the path of concurrency:
  - can't make cores much faster
  - but we can have more of them
- Concurrent programming is required for scalable performance of software on many-core platforms
  - synchronization performance and scalability has become a critical issue

Lock-Based Synchronization

- Pessimistic concurrency control
- Simple approach
  - identify shared data
  - associate locks with data
  - acquire before use
  - release after use
- Enforces mutual exclusion
  - why?
  - is this scalable?

Why Is Locking Tricky?

- Lock contention, especially for non-partitionable data
- Difficulty in partitioning data
- Lock overhead
- Inter-thread dependencies

Non-Blocking Synchronization

- Lock-free “optimistic” concurrency control
- Charge ahead, and if necessary roll-back and retry
- With sufficient programming effort can support fault tolerance properties
  - wait-freedom
  - obstruction-freedom, etc
- Avoids the need to access locks and data

Problems With Non-Blocking Synchronization

- Programming complexity
- Heavy use of atomic instructions
- Useless parallelism
- Excessive retries or helping under contention
- Excessive memory contention under heavy load

Strengths of Locking

- Simple and elegant
- No special hardware needed
- Wide-spread support
- Localized contention effects
- Power-friendly
- Good interaction with debuggers and other synchronization mechanisms
- Supports I/O and non-idempotent operations
- Ability to support disjoint access parallelism (with sufficient effort)

Problems and Their Solutions

- Lock contention
  - partition data, redesign algorithms (if you can!)
  - non-partitionable data structures are a problem
- Lock overhead
  - use RCU for readers
  - don't over-partition (but this conflicts with lock contention solution!)
  - update-heavy workloads are a problem

Problems and Their Solutions

- Deadlock among threads and among interrupt handlers and threads
  - locking hierarchy
  - try lock primitive with surrender and retry
  - detection and recovery
  - RCU for readers
  - special primitives to combine lock acquisition and interrupt disabling

Problems and Their Solutions

- Priority inversion
  - priority inheritance
  - RCU for readers
- Convoying
  - synchronization-aware scheduling
  - RCU for readers

Problems and Their Solutions

- Composability
  - expose synchronization at the interface level
- Thread failures
  - reboot

Transactional Memory

- Simple and elegant
- Enforces atomicity and isolation
- Composable
- Generally non-blocking
  - can be implemented in pessimistic form
- Automatic disjoint access parallelism
  - but doesn't help you ensure accesses are disjoint!
- Deadlock free
- Limited hardware support is possible

Transactional Memory Weaknesses

- Hardware support non-portable
  - need portable transaction virtualization support
- Hardware generally unavailable
  - need to wait for industry to provide it
- Software implementations have poor performance
  - solutions tend to weaken properties

Transactional Memory Weaknesses

- Does not support I/O, non-idempotent operations, or distributed systems
  - use pessimistic concurrency control
  - reintroduces problems of locking
- Contention management
  - need good contention management policy
  - and integration with scheduler
  - could use relativistic reads
- Privatization support sacrifices isolation

Transactional Memory Weaknesses

- High overhead
  - use TM for heavy weight operations
- Debugging
  - break points cause unconditional aborts in HTM
  - need better integration between HTM and STM

Scenario	Best Technique	Why?
Partitionable data structures	Locking	Disjoint Access Parallelism
Large Non Partitionable data structures	TM	Automatic Disjoint Access Parallelism
Read Mostly Situations	Locking/TM with Hazard Pointers/ RCU	Readers Scalable
Update Heavy Situations	TM	Writers Scalable
Complex fine grain locking design, No clear lock hierarchy exists	TM	Deadlock Avoidance
Atomic operations spanning multiple independent data structures, eg pop from one stack and push to another	TM	Composability
Single threaded software having embarrassingly parallel core containing only idempotent operations	TM	Performance benefits without much programming effort
Non Idempotent Operations	Locking	Supportability of non idempotent operations.
Large Critical Sections	Locking	Lock acquisition cost small compared to retry
Commodity Hardware	Locking	Commodity HW suffices. HTM requires specialized H/W and depends on cache geometry details. Else performance limited by STM

Conclusion

- Use the right tool for the job
- Understand all the techniques, their strengths and weaknesses, and potential interactions