Locks and Lock Alternatives

Why We Need Locks?
- Critical sections require exclusive access to data structures
  - Guarantee mutual exclusion
  - Writing to common data structures
  - Transactions that require atomicity (all or none) for correct execution
- Producer/consumer or writer/reader
- Barriers for synchronization
  - Ensure all participants reach same point before proceeding
  - Can degrade performance if frequent

Atomic Operations
- Necessary to implement locks
- Hardware support needed
  - Correctness
  - Performance
- Executed on an "all or nothing" basis
  - All parts of atomic operation succeed OR
  - No part of operation succeeds
- Most common operations require an atomic read, modify, and write for a memory location (also called fetch_and_Φ)
  - test_and_set
  - fetch_and_store
  - fetch_and_add
  - compare_and_swap

- Spin on test-and-set (paper table I)
  - All participating processors perform test-and-set until lock is free
  - One processor acquires lock, others continue spinning
- Disadvantages?
  - Processor holding lock contends for lock release
    - Cache line needs to be in "Modified" state before release
  - Creates hot spot in caches
  - Livelocks
  - Significant increase in interconnect traffic
    - Many processors read a shared variable
    - Significant invalidate traffic for caches

Other Spin-Lock Alternatives
- Spin on read (test-and-test-and-set) (Anderson, Table II)
  - Processors spin on reading the value of lock (no network traffic if block is in cache)
  - When lock is free, execute test-and-set
- Disadvantages?
  - For small critical sections, transient behavior dominates
    - When lock is released and acquired by a waiting processor, all other waiting processors cannot resume loop right away due to outstanding memory requests
  - Potential livelock
  - test-and-set with linear or exponential backoff
    - When test-and-set fails, wait some time before executing another test-and-set
    - Reduces network traffic, but still has livelock issues

Scalable Synchronization (Mellor-Crummey&Scott, ACM TOCS, 1991)
- Scalability problems with spin-locks
  - As number of participating processors increases, network congestion increases
  - As number of participating processors increases, probability of livelock increases
- Need simple algorithms that can be scalable to large systems
  - Correctness
  - Performance
  - Fairness
- Techniques implemented in software, but need hardware support for simple atomic operations
Mutual Exclusion

- Features of MCS lock
  - Guarantees FIFO order of lock acquisitions (fairness)
  - Spins on locally-accessible flag variables (not global shared lock variable)
  - Requires constant amount of space per lock
  - Requires O(1) network transactions per lock acquisition
- Implementation (Algorithm 1 in paper)
  - Each lock is represented by a queue of participating processors
  - Each processor spins on its local queue node
  - When processor holding the lock releases it, local queue node for next processor in queue is updated

Reader-Writer Control

- Fairness policy
  - Read request is granted when all previous write requests have been completed
  - Write request is granted when all previous read and write requests have completed
- Implementation (Algorithm 3 in paper)
  - Linked list of requesting processors is maintained
  - Each requestor can read and write fields in its predecessor’s record
  - A reader can begin reading if its predecessor is an active reader, but it must first unblock its successor if it is a waiting reader
  - A writer can proceed if its predecessor is done, and there are no active readers
  - Need to keep track of the number of active readers
  - Need to keep track of last writer

Barrier Synchronization

- Barrier algorithm features
  - Spins on locally-accessible flags only
  - Requires O(P) space for P processors
  - Performs theoretical minimum number of network transactions on machines without broadcast (2P - 2)
  - Performs O(log P) network transactions on its critical path
- Implementation (paper Algorithm 4)
  - Uses two P-node trees
  - Each processor is assigned a unique tree node
    - Linked to arrival tree by a parent link
    - Linked into a wakeup tree by a set of child links
  - Processor signals its arrival at barrier by setting a flag in its parent’s node
  - When done, processor notifies its children by writing a flag in their nodes

Performance

- Performance of spin locks
  - Performance on the Butterfly (paper figures 1 and 2)
  - Performance on the Symmetry (paper figure 3)
  - Network latency on Butterfly for 60 processors (paper table 1)
- Barrier performance
  - Performance on the Butterfly (paper figures 4 and 5)
  - Performance on the Symmetry (paper figure 6)

Speculative Lock Elision

- Problem: False inter-thread dependencies
- SLE Goals
  - Remove dynamically unnecessary lock-induced serialization
  - Enable highly-concurrent multithreaded execution
- SLE Mechanism
  - Predict synchronization operations as being unnecessary
    - Allows multiple threads to concurrently execute critical sections protected by the same lock
  - Execute critical section
  - Check for conflicts
    - If conflict happened, rollback execution to previous checkpoint
    - If conflict did not happen, commit elision without acquiring the lock

Reading Assignment

- Kevin Moore et al., "LogTM: Log-based Transactional Memory," HPCA, 2006 (Skim)
- Project progress reports due today