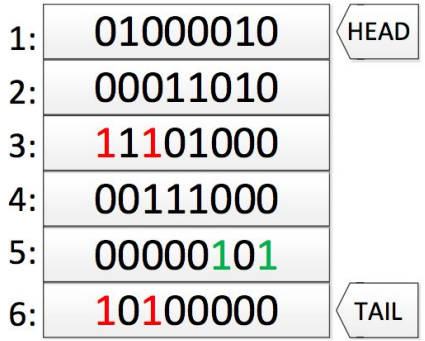
Non-blocking synchronization algorithms on multicore machines

Tianyuan Ding
Preeti Murthy

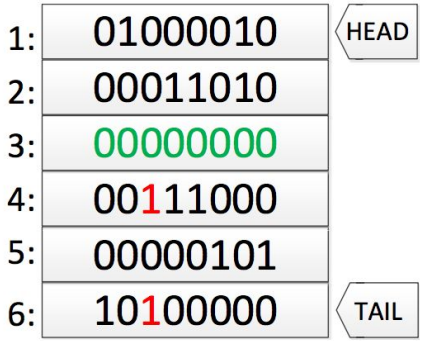
Fast and Scalable Queue-Based Resource Allocation Lock

- Multi-resource lock algorithm that guarantees the FIFO fairness
- Resources are encoded as a bit set
- Internally using a non-blocking queue where competing threads spin on previous conflicting resource requests made by other cores
- All-or-nothing atomic acquisition, wait until all my requested sources are made available
- Upon release, clear the corresponding bit, and remove the request until all bits are cleared so that thread waits for it can proceed

Example



(a) Cell 6 spins on cell 3



(b) Release of cell 3. Cell 6 spins on cell 4

Thoughts

- They use compare-and-swap for
  - spinning on adding requests to the queue
  - spinning until all my requested sources are available
  - spinning until all bits in a request are cleared so that the request can be removed
- Alternatively, rather than each sends messages to a centralized queue, sends requests to anyone who has previously requested any resource bit
- When a thread finishes, forward the waiters to other threads have requested other resources. The waiter can finally run if there are no conflicts anymore

Revisiting the Combining Synchronization Technique

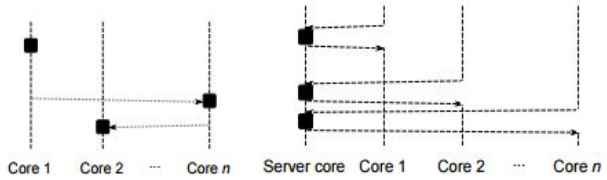
- Proposes combining synchronization technique CC-Synch and DSM-Synch
  - CC-Synch (cache coherent) addresses systems that support coherent caches
  - DSM-Synch (distributed shared memory) works better in cacheless NUMA machines.
- In CC-Synch threads put requests on a queue
- The lock holder who is the top of the queue is called the “combiner”, others spinning on a “wait” flag
  - Combiner not only processes its own critical section, but also serves all other pending requests (or a specified number)
  - DSM-Synch has a similar design, but the combiner will stop until it reaches the second to last element in the queue
- Both try to reduce remote memory reference (RMR)
  - DSM-Synch has bounded RMRs
  - CC-Synch has unbounded RMRs in the DSM model

Thoughts

- Once the combiner is determined, threads can send interrupts to it
  - Message contains relevant information such as critical section routine, etc.
- Upon the combiner releases the lock, it will process all messages and execute the routines
- Security issue from combining synchronization technique in general?

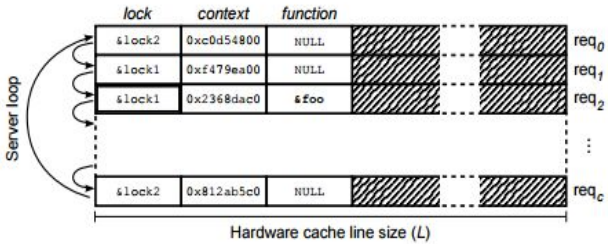
Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

- Reduce lock contention on legacy systems
- Does not require deep understanding of application code
- Applications may not use all cores; Dedicate a core to handle critical section
- Avoids cache contention
- Extract critical section into a function call executed remotely on a core
- Additional service threads to ensure progress of work on remote core
- Tool to identify the potential locks to transform to RCL
- Tool to transform critical section into Remote Procedure Call
- Pitfalls: Deadlock, Priority Inversion, Serialization of all critical sections



(a) POSIX locks (b) RCL

Fig. 1: Critical sections with POSIX locks vs. RCL.



for the third word of req_i to be reset to `NULL`, indicating that the server has executed the critical section. In order to improve energy efficiency, if there are less clients than the number of cores available, the SSE3 `monitor/mwait` instructions can be used to avoid spinning: the client will sleep and be woken up automatically when the server writes into the third word of req_i.

Server side A servicing thread iterates over the requests, waiting for one of the requests to contain a non-`NULL` value in its third word. When such a value is found, the servicing thread checks if the requested lock is free and, if so, acquires the lock and executes the critical section using the function pointer and the context. When the servicing thread is done executing the critical section, it resets the third word to `NULL`, and resumes the iteration.

RH Lock: A Scalable Hierarchical Spin Lock

- NUMA Aware Lock
- Test and Set Locking Primitives: Unfair but avoid node to node bouncing
- Queue based Locking Primitives: Eg: MCS: Fair but can result in node to node bouncing of locks
- RH Lock aims at avoiding node to node bouncing of locks
- Every lock must be local to atmost one node and remote to all other nodes
- Acquire favours local locks over remote locks
- Remote locks are acquired by exponential backoffs
- Remote locks once acquired become local to the node

An Efficient Asymmetric Distributed Lock for Embedded Multiprocessor Systems

- Hardware Locking Primitives are too complex to implement in embedded systems
- Use Software Locking Primitives which have little memory contention
- Implementation very similar to our locking primitive
- Difference lies in having a dedicated core to keep track of the owner of the lock
- This core replies to lock requests with ‘free’ or ‘ask owner’
- A dedicated core to receive ‘ask owner’ requests and to notify owner about waiter

References

1. http://link.springer.com/chapter/10.1007/978-3-319-03850-6_19
2. http://dl.acm.org/citation.cfm?id=2145849
3. https://it.uu.se/research/group/uart/pub/radovic_2002_may/radovic_2002_may.pdf
4. http://dl.acm.org/citation.cfm?id=2442532
5. https://www.usenix.org/system/files/conference/atc12/atc12-final237.pdf