Module 7: Synchronization

Lecture 13: Introduction to Atomic Primitives

The Lecture Contains:

Previous Next

Module 7: Synchronization

Lecture 13: Introduction to Atomic Primitives

Synchronization

Types

- Mutual exclusion
  - Synchronize entry into critical sections
  - Normally done with locks
- Point-to-point synchronization
  - Tell a set of processors (normally set cardinality is one) that they can proceed
  - Normally done with flags
- Global synchronization
  - Bring every processor to sync
  - Wait at a point until everyone is there
  - Normally done with barriers

Synchronization

- Normally a two-part process: acquire and release; acquire can be broken into two parts: intent and wait
  - Intent: express intent to synchronize (i.e. contend for the lock, arrive at a barrier)
  - Wait: wait for your turn to synchronization (i.e. wait until you get the lock)
  - Release: proceed past synchronization and enable other contenders to synchronize
- Waiting algorithms do not depend on the type of synchronization

Previous Next

Waiting Algorithms

- Busy wait (common in multiprocessors)
  - Waiting processes repeatedly poll a location (implemented as a load in a loop)
  - Releasing process sets the location appropriately
  - May cause network or bus transactions
- Block
  - Waiting processes are de-scheduled
  - Frees up processor cycles for doing something else
- Busy waiting is better if
  - De-scheduling and re-scheduling take longer than busy waiting
  - No other active process
  - Does not work for single processor
- Hybrid policies: busy wait for some time and then block

Implementation

- Popular trend
  - Architects offer some simple atomic primitives
  - Library writers use these primitives to implement synchronization algorithms
  - Normally hardware primitives for acquire and possibly release are provided
  - Hard to offer hardware solutions for waiting
  - Also hardwired waiting may not offer that much of flexibility

Previous Next

Hardwired Locks

- Not popular today
  - Less flexible
  - Cannot support large number of locks
- Possible designs
  - Dedicated lock line in bus so that the lock holder keeps it asserted and waiters snoop the lock line in hardware
  - Set of lock registers shared among processors and lock holder gets a lock register (Cray Xmp)

Software Locks

- Bakery algorithm

Shared: choosing[P] = FALSE, ticket[P] = 0;

Acquire : choosing[i] = TRUE; ticket[i] = max(ticket[0],…,ticket[P-1]) + 1;

choosing[i] = FALSE;
for j = 0 to P-1

while (choosing[j]);
while (ticket[j] && ((ticket[j], j) < (ticket[i], i)));

endfor

Release : ticket[i] = 0;

- Does it work for multiprocessors?
  - Assume sequential consistency
  - Performance issues related to coherence?
- Too much overhead: need faster and simpler lock algorithms
  - Need some hardware support

Previous Next

Module 7: Synchronization
Lecture 13: Introduction to Atomic Primitives

Hardware Support

- Start with a simple software lock

  Shared: lock = 0;
  Acquire : while (lock); lock = 1;
  Release or Unlock : lock = 0;

- Assembly translation

  Lock: lw register, lock_addr /* register is any processor register */
  bnez register, Lock
  addi register, register, 0x1
  sw register, lock_addr
  Unlock: xor register, register, register
  sw register, lock_addr

- Does it work?
  - What went wrong?
  - We wanted the read-modify-write sequence to be atomic

Atomic Exchange

- We can fix this if we have an atomic exchange instruction

  addi register, r0, 0x1 /* r0 is hardwired to 0 */
  Lock: xchg register, lock_addr /* An atomic load and store */

  bnez register, Lock
  Unlock remains unchanged

- Various processors support this type of instruction
  - Intel x86 has xchg , Sun UltraSPARC has ldstub (load-store-unsigned byte), UltraSPARC also has swap
  - Normally easy to implement for bus-based systems: whoever wins the bus for xchg can lock the bus
  - Difficult to support in distributed memory systems

Previous Next

Test & Set

- Less general compared to exchange

  Lock: ts register, lock_addr
  bnez register, Lock
  Unlock remains unchanged

- Loads current lock value in a register and sets location always with 1
  - Exchange allows to swap any value
- A similar type of instruction is fetch & op
  - Fetch memory location in a register and apply op on the memory location
  - Op can be a set of supported operations e.g. add, increment, decrement, store etc.
  - In Test & set op=set

Fetch & op

- Possible to implement a lock with fetch & clear then add (used to be supported in BBN Butterfly 1)

  addi reg1, r0, 0x1
  Lock: fetch & clr then add reg1, reg2, lock_addr
  /* fetch in reg2, clear, add reg1 */
  bnez reg2, Lock

- Butterfly 1 also supports fetch & clear then xor
- Sequent Symmetry supports fetch & store
- More sophisticated: compare & swap
  - Takes three operands: reg1, reg2, memory address
  - Compares the value in reg1 with address and if they are equal swaps the contents of reg2 and address
  - Not in line with RISC philosophy (same goes for fetch & add)

Previous Next

Compare & Swap

```
addi reg1, r0, 0x0 /* reg1 has 0x0 */
addi reg2, r0, 0x1 /* reg2 has 0x1 */
Lock: compare & swap reg1, reg2, lock_addr
bnez reg2, Lock
```

Traffic of Test & Set

- In some machines (e.g., SGI Origin 2000) uncached fetch & op is supported
  - Every such instruction will generate a transaction (may be good or bad depending on the support in memory controller; will discuss later)
- Let us assume that the lock location is cacheable and is kept coherent
  - Every invocation of test & set must generate a bus transaction; Why? What is the transaction? What are the possible states of the cache line holding lock_addr ?
  - Therefore all lock contenders repeatedly generate bus transactions even if someone is still in the critical section and is holding the lock
- Can we improve this?
  - Test & set with backoff

Previous Next

Backoff Test & Set

Instead of retrying immediately wait for a while

- How long to wait?
- Waiting for too long may lead to long latency and lost opportunity
- Constant and variable backoff
- Special kind of variable backoff : exponential backoff (after the i th attempt the delay is k* c i where k and c are constants)
- Test & set with exponential backoff works pretty well

```
delay = k
Lock: ts register, lock_addr
bez register, Enter_CS
pause (delay) /* Can be simulated as a timed loop */
delay = delay*c
j Lock
```

Test & Test & Set

- Reduce traffic further

  Before trying test & set make sure that the lock is free

```
Lock: ts register, lock_addr
bez register, Enter_CS
Test: lw register, lock_addr
bnez register, Test
j Lock
```

- How good is it?
  - In a cacheable lock environment the Test loop will execute from cache until it receives an invalidation (due to store in unlock); at this point the load may return a zero value after fetching the cache line
  - If the location is zero then only everyone will try test & set

TTS Traffic Analysis

- Recall that unlock is always a simple store
- In the worst case everyone will try to enter the CS at the same time
  - First time P transactions for ts and one succeeds; every other processor suffers a miss on the load in Test loop; then loops from cache
  - The lock-holder when unlocking generates an upgrade (why?) and invalidates all others
  - All other processors suffer read miss and get value zero now; so they break Test loop and try ts and the process continues until everyone has visited the CS

    (P+(P-1)+1+(P-1))+((P-1)+(P-2)+1+(P-2))+… = (3P-1) +
    (3P-4) + (3P-7) + … ~ 1.5P 2 asymptotically

  - For distributed shared memory the situation is worse because each invalidation becomes a separate message (more later)

Goals of a Lock Algorithm

- Low latency: If no contender the lock should be acquired fast
- Low traffic: Worst case lock acquire traffic should be low; otherwise it may affect unrelated transactions
- Scalability: Traffic and latency should scale slowly with the number of processors
- Low storage cost: Maintaining lock states should not impose unrealistic memory overhead
- Fairness: Ideally processors should enter CS according to the order of lock request (TS or TTS does not guarantee this)

Previous Next