CS315A/EE382B: Lecture 10

SMP2 and Synchronization

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Announcements

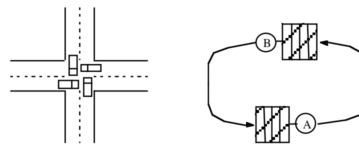
- PS2 due today
  - no late day
- Midterm exam Wed May 10
  - 7-9pm Gates B03
  - Lectures 1-9
  - No class on May 10
- PA2 due Wed May 15

Today's Outline

- Finish off SMP Implementation
- Synchronization
  - Locks
  - Barriers

Protocol Correctness

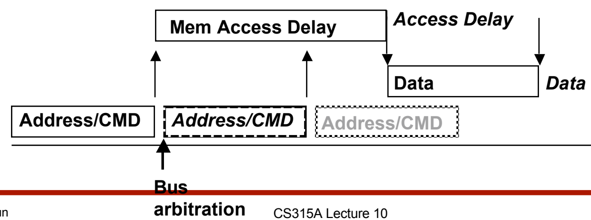
- Protocol must maintain coherence and consistency
- Protocol implementation should prevent:
- Deadlock:
  - all system activity ceases
  - Cycle of resource dependences



- Livelock:
  - no processor makes forward progress
  - constant on-going transactions at hardware level
  - e.g. simultaneous writes in invalidation-based protocol
- Starvation:
  - some processors make no forward progress
  - e.g. a processor always loses bus arbitration

Split-Transaction Bus

- Split bus transaction into request and response sub-transactions
  - Separate arbitration for each phase
- Other transactions may intervene
  - Improves bandwidth dramatically
  - Response is matched to request
  - Buffering between bus and cache controllers
- Reduce serialization down to the actual bus arbitration



Complications

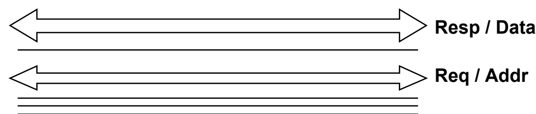
- New request can appear on bus before previous one serviced
  - Even before snoop result obtained
  - Conflicting operations to same block may be outstanding on bus
  - e.g. P1, P2 write block in S state at same time
    - both get bus before either gets snoop result, so both think they've won
- Buffers are small, so may need *flow control*
- Buffering implies revisiting snoop issues
  - When and how snoop results and data responses are provided
  - In order w.r.t. requests? (PPro, DEC Turbolaser: yes; SGI, Sun: no)
  - Snoop and data response together or separately?
    - SGI together, SUN separately

Example (based on SGI Challenge)

- No conflicting requests for same block allowed on bus
  - 8 outstanding requests total, makes conflict detection tractable
- Flow-control through negative acknowledgement (NACK)
  - NACK as soon as request appears on bus, requestor retries
  - Separate command (incl. NACK) + address and tag + data buses
- Responses may be in different order than requests
  - Order of transactions determined by requests
  - Snoop results presented on bus with response
- Look at
  - Bus design, and how requests and responses are matched
  - Snoop results and handling conflicting requests
  - Flow control
  - Path of a request through the system

© 2005 Kunle Olukotun

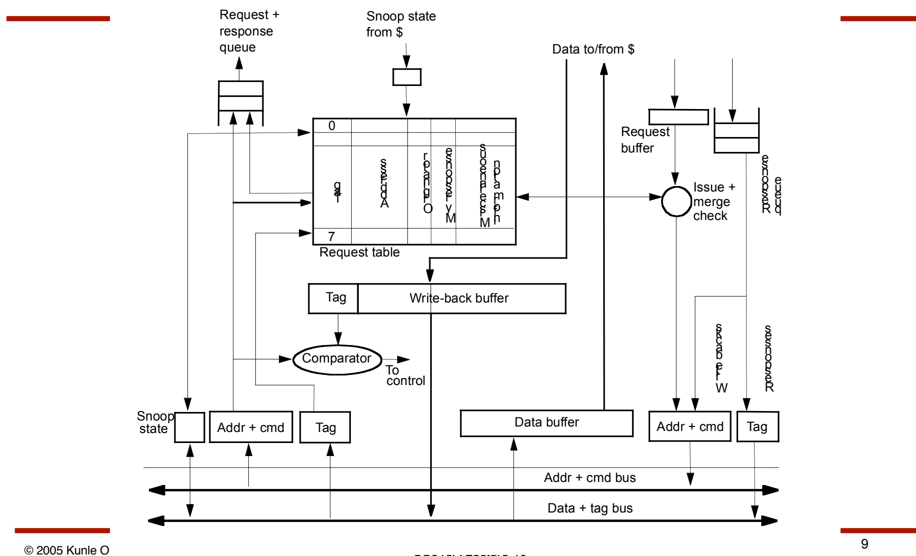
Bus Design and Req-Resp Matching



- Essentially two separate buses, arbitrated independently
  - "Request" bus for command and address
  - "Response" bus for data
- Out-of-order responses imply need for matching req-response
  - Request gets 3-bit tag when wins arbitration
    - max 8 outstanding
  - Response includes data as well as corresponding request tag
  - Tags allow response to not use address bus, leaving it free
- Separate bus lines for arbitration, and for snoop results

© 2005 Kunle Olukotun

Bus Interface with Request Table



Bus Design (continued)

- Tracking outstanding requests and matching responses
  - Eight-entry "request table" in each cache controller
  - New request on bus added to all at same index, determined by tag
  - Entry holds address, request type, state in that cache (if determined already), ...
  - All entries checked on bus or processor accesses for match, so fully associative
  - Entry freed when response appears, so tag can be reassigned by bus

Snoop Results and Conflicting Requests

- Variable-delay snooping
- Shared, dirty and inhibit wired-OR lines
- Snoop results presented when response appears
  - Determined earlier, in request phase, and kept in request table entry
  - Also determined who will respond
  - Writebacks and upgrades don't have data response or snoop result
- Avoiding conflicting requests on bus
  - don't issue request for conflicting request that is in request table
- Writes committed when request gets bus

Flow Control

- Where?
  - incoming request buffers from bus to cache controller
  - response buffer
    - Controller limits number of outstanding requests
- Mainly needed at main memory in this design
  - Each of the 8 transactions can generate a writeback
  - Can happen in quick succession (no response needed)
  - ◆SGI Challenge: separate NACK lines for address and data buses
    - Request (response) cancelled everywhere, and retries later
    - Backoff and priorities to reduce traffic and starvation
  - SUN Enterprise: destination initiates retry when it has a free buffer
    - source keeps watch for this retry
    - guaranteed space will still be there, so only two "tries" needed at most

Handling a Read Miss

- Need to issue BusRd
- First check request table. If hit:
  - If prior request exists for same block, want to grab data too!
    - "want to grab response" bit
    - "original requestor" bit
      - non-original grabber must assert sharing line so others will load in S rather than E state
  - If prior request incompatible with BusRd (e.g. BusRdX)
    - wait for it to complete and retry (processor-side controller)
  - If no prior request, issue request and watch out for race conditions
    - Window of vulnerability
    - conflicting request may win arbitration before this one, but this one receives bus grant before conflict is apparent
      - watch for conflicting request in slot before own, degrade request to "no action" and withdraw till conflicting request satisfied

© 2005 Kunle Olukotun

Upon Issuing the BusRd Request

- All processors enter request into table, snoop for request in cache
- Memory starts fetching block
- 1. Cache with dirty block responds before memory ready
  - Memory aborts on seeing response
  - Waiters grab data
    - some may assert inhibit to extend response phase till done snooping
    - memory must accept response as WB (might even have to NACK)
- 2. Memory responds before cache with dirty block
  - Cache with dirty block asserts inhibit line till done with snoop
  - When done, asserts dirty, causing memory to cancel response
  - Cache with dirty issues response, arbitrating for bus
- 3. No dirty block: memory responds when inhibit line released
  - Assume cache-to-cache sharing not used (for non-modified data)

© 2005 Kunle Olukotun

Handling a Write Miss

- Similar to read miss, except:
  - Generate BusRdX
  - Main memory does not sink response since will be modified again
  - No other processor can grab the data
- If block present in shared state, issue BusUpgr instead
  - No response needed
  - If another processor was going to issue BusUpgr, changes to BusRdX as with atomic bus

Detecting Write Completion

- Problem: invalidations don't happen as soon as request appears on bus
  - They're buffered between bus and cache
  - Need additional mechanisms
- Key property to preserve: processor shouldn't see new value produced by a write before previous writes in bus order are visible to it
  - 1. Don't let certain types of incoming transactions be reordered in buffers
    - in particular, data reply should not overtake invalidation request
    - okay for invalidations to be reordered: only reply actually brings data in
  - 2. Allow reordering in buffers, but ensure important orders preserved at key points
    - e.g. flush incoming invalidations/updates from queues and apply before processor completes operation that may enable it to see a new value

Write Serialization and Atomicity

- Still provided naturally by broadcast nature of bus
  - Order of requests acked on the bus
- Recall that bus implies:
  - writes commit in same order w.r.t. all processors
  - read cannot see value produced by write before write has committed on bus and hence w.r.t. all processors

© 2005 Kunle Olukotun

Synchronization

- Mutual Exclusion (critical sections)
  - Lock & Unlock
- Event Notification
  - point-to-point (producer-consumer, flags)
  - global (barrier)
- LOCK, BARRIER
  - How are these implemented?

© 2005 Kunle Olukotun

Anatomy of A Synchronization Operation

- Acquire Method
  - method for trying to obtain the lock, or proceed past barrier
  - Acquire right to the synch
- Waiting Algorithm
  - Spin or busy wait
  - Block (suspend)
- Release Method
  - method to allow other processes to proceed past synchronization event
  - Enable other processors to acquire right to the synch

- Waiting algorithm is independent of type of synchronization
  - makes no sense to put in hardware

HW/SW Implementation Tradeoffs

- User wants high level (ease of programming)
  - LOCK(lock_variable), UNLOCK(lock_variable)
  - BARRIER(barrier_variable, Num_Procs)
- Hardware
  - The Need for Speed (it's fast)
- Software wants
  - Flexibility

How Not To Implement Locks

- LOCK
  while(**lock_variable** == 1);
  lock_variable = 1;

- UNLOCK
  lock_variable = 0;

- Implementation requires Mutual Exclusion!
  - Can't have two processes successfully acquire the lock
  - Need atomic way to both read and write

Atomic Instructions

- Specifies a location, register, & atomic operation
  - Value in location read into a register
  - Another value (function of value read or not) stored into location
- Many variants
  - Varying degrees of flexibility in second part
- Simple example: test&set
  - Value in location read into a specified register
  - Constant 1 stored into location
  - Successful if value loaded into register is 0
  - Other constants could be used instead of 1 and 0

Simple Test&Set Lock

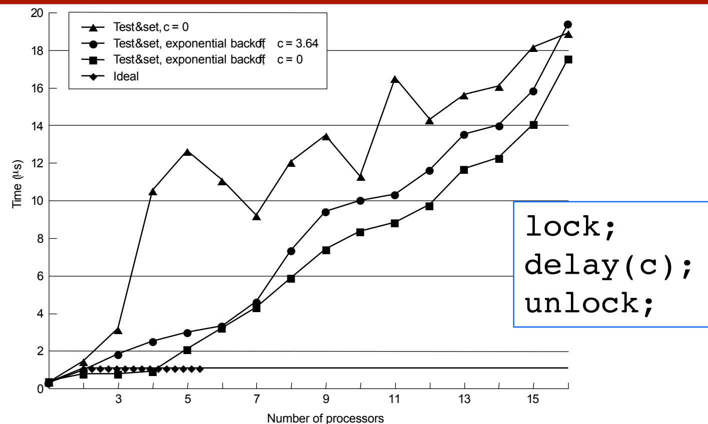
```
lock:     t&s   register, location
          bnz   lock                  /* if not 0, try again */
          ret                         /* return control to caller */
unlock:   st    location, #0          /* write 0 to location */
          ret                         /* return control to caller */
```

- Other read-modify-write primitives
  - Swap
  - Fetch&op
  - Compare&swap
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets, except SPARC
- cacheable or uncacheable

Performance Criteria for Synch. Ops

- Latency (time per op)
  - especially when light contention
- Bandwidth (ops per sec)
  - especially under high contention
- Traffic
  - load on critical resources (e.g. bus)
  - especially on failures under contention
- Storage
- Fairness

T&S Lock Microbenchmark



- Why does performance degrade?
- Bus Transactions on T&S?

Spin Lock with Test & Set

<u>LOCK</u>

```
while (test&set(x) == 1);
```

<u>UNLOCK</u>

```
x = 0;
```

- High contention (many processes want lock)
- Remember the CACHE!
- Each test&set is a read miss and a write miss
  - Not fair
- Problem is?
- Waiting Algorithm!
- Requires a read and write
  - Uninterruptable sequence
  - Complicates coherence protocol

Improved Hardware Primitives: LL-SC

- Goals:
  - Test with reads
  - Failed read-modify-write attempts don't generate invalidations
  - Nice if single primitive can implement range of r-m-w operations
- *Load-Locked* (or -linked), *Store-Conditional*
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - succeed if and only if no other write to the variable since this processor's LL
    - indicated by condition codes, register value
- If SC succeeds, all three steps happened atomically
- If fails, doesn't write or generate invalidations
  - must retry acquire

© 2005 Kunle Olukotun

Simple Lock with LL-SC

```
lock:     ll      reg1, location      /* LL location to reg1 */
          sc      location, reg2      /* SC reg2 into location*/
          beqz    reg2, lock          /* if failed, start again */
          ret
unlock:   st      location, #0        /* write 0 to location */
          ret
```

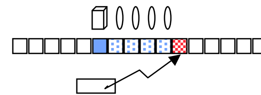
- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't include instructions that would need to be undone (e.g. stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor's SC gets bus first
- LL, SC are not lock, unlock respectively
  - Only guarantee no conflicting write to lock variable between them
  - But can use directly to implement simple operations on shared variables

© 2005 Kunle Olukotun

Better Lock Implementations

- Two choices:
  - Don’t spin so much
  - Spin without generating bus traffic
- Spin lock with backoff
  - Insert delay between attempts to lock (not too long)
  - Expenonential seems good ($k*c^i$)
  - Not fair
- Test-and-Test&Set
  - Spin (test) on local cached copy until it gets invalidated, then issue store conditional
  - Intuition: No point in trying to set the location until we know that it’s not set, which we can detect when it gets invalidated...
  - Still contention after invalidate
  - Still not fair

Ticket Lock



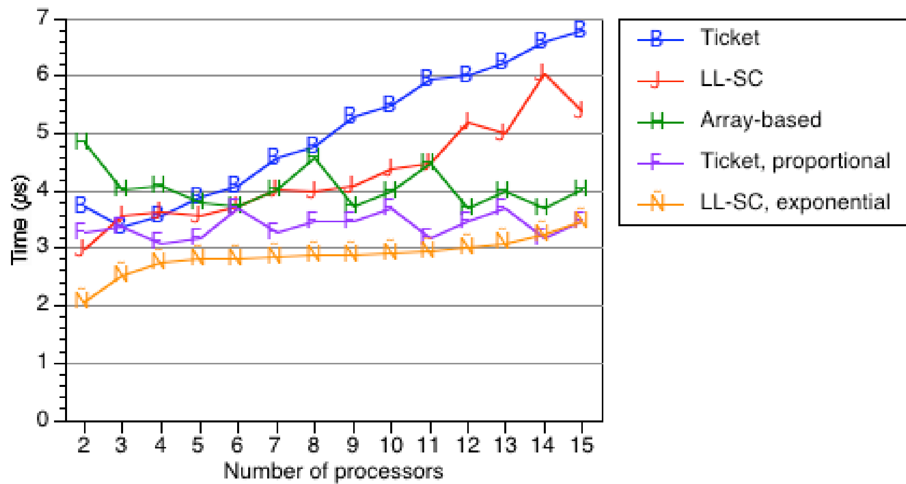
- Two counters per lock (next_ticket, now_serving)
  - Acquire: **fetch&inc next_ticket;**
    wait for now_serving == next_ticket
    - atomic op when arrive at lock, not when it’s free (so less contention)
  - Release: increment now-serving
- Only one r-m-w per acquire
- Performance
  - low latency for low-contention - if fetch&inc cacheable
  - O(p) read misses at release, since all spin on same variable
  - FIFO order
    - like simple LL-SC lock, but no inval when SC succeeds, and fair
- Wouldn’t it be nice to poll different locations ...

Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size *p*
  - Acquire
    - fetch&inc to obtain address on which to spin (next array element)
    - ensure that these addresses are in different cache lines or memories
  - Release
    - set next location in array, thus waking up process spinning on it
  - *O(1)* traffic per acquire with coherent caches
  - FIFO ordering, as in ticket lock, but, *O(p)* space per lock

© 2005 Kunle Olukotun

Lock Performance on SGI Challenge



© 2005 Kunle Olukotun

Point-to-Point Event Synchronization I

- Often use normal variables as flags

P1	P2
`a = f(x);`	`while (flag == 0);`
`flag = 1;`	`b = g(a);`

- If we know value of a before hand

P1	P2
`a = f(x);`	`while (a == 0);`
	`b = g(a);`

- Assumes Sequential Consistency!!

© 2005 Kunle Olukotun

Synchronization Summary

- Rich interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads
- Simple software algorithms with common hardware primitives do well on bus

© 2005 Kunle Olukotun