Synchronization

Serif Yesil, Josep Torrellas

2/19/2019

Outline

1) High level synchronization primitives
2) Components of a synchronization primitive
   a) Waiting algorithms and their tradeoffs
3) Hardware synchronization primitives
   a) Most common primitives
   b) Simple lock implementation & bus contention
   c) LL/SC primitives
4) Compare and Swap & ABA problem
5) Examples of hardware synchronization primitives
   a) Lock prefixes (Intel X86)
   b) Full/Empty Bits (HEP)
   c) Message combining (NYU Ultracomputer)
   d) Synchronization words (Illinois CEDAR)
6) Key points of the paper by Goodman et al.

Why Do We Need Synchronization?

- Addition and deletion of elements from a shared (work) queue
- Access to critical sections
- Enforcement of low-level data dependencies within loop iterations
- Synchronizing across multiple processors/threads

High Level Mechanisms for Synchronization

- Mutual exclusion, point-to-point events and global events
- Locks/mutexes: grant access to one process only
- Barriers: no process advances beyond it until all have arrived
- Semaphores: control access to a shared resource in a concurrent execution
- Monitors: synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true
- All implemented in libraries/systems
  - Other examples in runtimes?

Components of Sync. Events

- Acquire method: how do we acquire the synchronization event?
- Waiting algorithm: what happens if we try to acquire the synchronization event but it is acquired by some other process/thread
- Release method: how to inform other processes when we past synchronization event?
- Acquire and release methods: defined semantically by the operation

Waiting algorithm

Blocking: preempt waiting process

- The process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait.
- It will be awoken and made ready to run again when the release it was waiting for occurs.

Busy-wait: process repeatedly tests shared variables to determine when it can proceed

- Busy-waiting means that the process spins in a loop that repeatedly tests for a variable to change its value.
- A release of the synchronization event by another processor changes the value of the variable, allowing the process to proceed.

Tradeoffs?

Blocking

- Higher overhead: suspending and resuming a process involves the operating system, the runtime system
- Makes the processor available to other threads with useful work

Busy-waiting

- Avoids the cost of suspension
- Consumes the processor and memory system bandwidth while waiting.

Tradeoffs?

Blocking is strictly more powerful than busy waiting, because if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.

Busy-waiting is better:

- When the waiting period is short.
- Network/Cache can tolerate hot spots.
- Cannot be pre-empted (OS)

Blocking is better:

- When the waiting period is long and there are other processes to run.

Hardware Mechanisms for Synchronization

All high level synchronization mechanisms can be implemented in hardware

- Speed advantage
- Functionality and flexibility disadvantage

What is the minimum hardware support that can implement all high level synchronization mechanisms?

- Focus on shared memory

Primitives for Synchronization

Uninterruptible instruction or instruction sequence

- Capable of atomic read-modify-write (RMW)
- Atomic exchange
- Fetch-and-increment
- Test & Set

Non-atomic sequence of instructions that detect if intervening access

- Load-linked and Store-conditional
- Can be used to implement more complex primitives

Semantics of Primitives

bool TAS(bool *a): atomic { t := *a; *a := true; return t }
word Swap(word *a, word w): atomic { t := *a; *a := w; return t }
int fetchAndIncrement(int *a): atomic { t := *a; *a := t + 1; return t }
int fetchAndAdd(int *a, int n): atomic { t := *a; *a := t + n; return t }
bool CAS(word *a, word old, word new): atomic { t := (*a = old); if (t) *a := new; return t }

- Test-and-set: set a value if unset.
- Swap: exchange values in memory locations with register
- Fetch-And-Increment: increments value, returns previous value stored in the memory location
- Fetch-And-Add: increments value with a constant, return value stored in the memory location
- Compare-and-swap: compares the value stored in memory location with a given value, if same swaps it with new value

Implementing a Simple Lock

acquire	Lock: test-and-set R1, lock_mem bnz R1, Lock ret
release	Unlock: st lock_mem, #0 ret

acquire	Lock: ld R1, lock_mem bnz R1, Lock test-and-set R1, lock_mem bnz R1, Lock ret
release	Unlock: st lock_mem, #0 ret

Think about cache coherence & bus transactions

Implementing a Simple Lock

acquire	Lock: test-and-set R1, lock_mem bnz R1, Lock ret
release	Unlock: st lock_mem, #0 ret

- When lock is successfully acquired -> 1 exclusive read operation is done on the bus
- When lock acquire is unsuccessful, every check generates an exclusive read operation on the bus
- High number of transactions on the bus

Implementing a Simple Lock

acquire	Lock: ld R1, lock_mem bnz R1, Lock test-and-set R1, lock_mem bnz R1, Lock ret
release	Unlock: st lock_mem, #0 ret

- When locking is successful, 1 read and 1 exclusive read is observed on the bus
- When locking is unsuccessful
  - Each processor reads the value from its own cache with ld. Loops until lock is released
- $O(N^2)$ transactions on the bus
  - Each time lock is unset, all processors issue an exclusive access, but only 1 is successful

Is there a method or methods of locking that get better performance than Test and Test and Set lock?

Note on Lock Performance

- Latency
  - Latency of operations to acquire the lock
  - 1 operation in Test&Set lock
  - 2 operations in Test&Test&Set lock
- Interconnect traffic (bus requests)
  - How many requests are we generating on the bus
  - Unlimited vs $O(N^2)$
- Storage cost
  - 1 word for both of them
- Fairness?
  - Every processor gets the same chance to acquire the lock?

Reducing Implementation Complexity

Problem with CAS: it combines a load and a store into a single instruction

bool CAS(word *a, word old, word new):
atomic { t := (*a = old); if (t) *a := new; return t }

word LL(word *a):
atomic { remember a; return *a }

bool SC(word *a, word w):
atomic { t := (a is remembered, and has not been evicted since LL)
if (t) *a := w; return t }

- Use 2 instructions, where the 2nd one returns a value from which it can be deduced whether the pair was executed as if atomic
- LL: returns value of location. Remembers the value
- SC: fails if contents of location have been changed between LL and SC
  - also fails if processor context switches between LL and SC
- Can be used to implement other primitives like Fetch & increment

CAS with LL/SC

```
cas(addr, old, new):
   A=LL(addr)
  if (A == old){
    if (SC(new, addr)) return 1;
    else return 0;
  } return 0;
```

- LL/SC is universal
- Other primitive with LL/SC
  - Fetch&Add
  - Fetch&Inc etc.

CAS ABA Problem

CAS depends on value comparison

May introduce correctness issues

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

Note: CAS compares the address of nodes not the value of a node!

THREAD1

pop(&top)

THREAD2

pop(&top)

push(&top, &B)

push(&top, &A)

Initial Stack:

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

THREAD1

pop(&top)

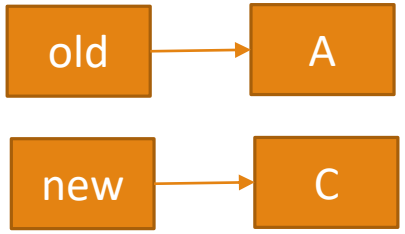
- Executes until line 6
- Stuck at line 7

THREAD2

pop(&top)

push(&top, &B)

push(&top, &A)



Initial Stack:

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

THREAD1

pop(&top)

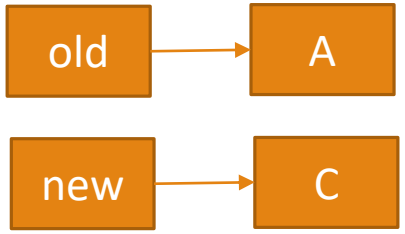
- Executes until line 6
- Stuck at line 7

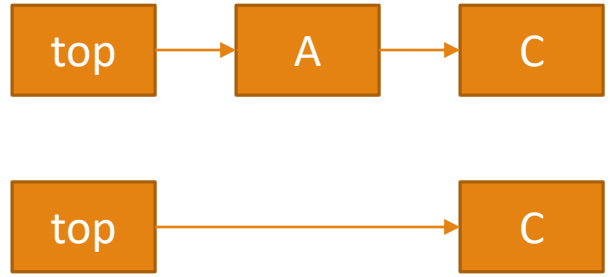
THREAD2

A = pop(&top)

push(&top, &B)

push(&top, &A)





```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

THREAD1

pop(&top)

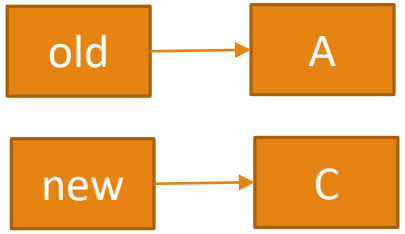
- Executes until line 6
- Stuck at line 7

THREAD2

pop(&top)

push(&top, &B)

push(&top, &A)



Before:

After:

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

THREAD1

pop(&top)

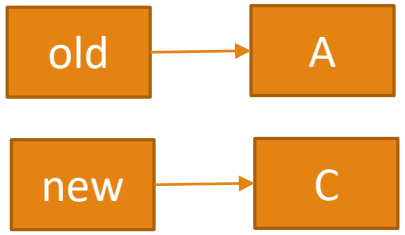
- Executes until line 6
- Stuck at line 7

THREAD2

pop(&top)

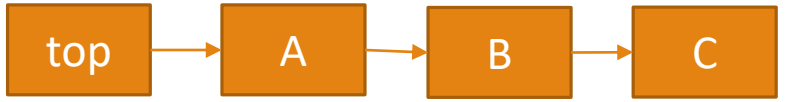
push(&top, &B)

push(&top, &A)



Before:

After:



```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

old==A & top==A → CAS successful
New top=C

THREAD1

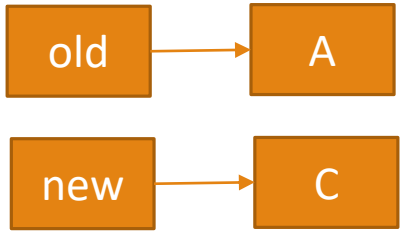
pop(&top)
- Continue executing line 7

THREAD2

pop(&top)

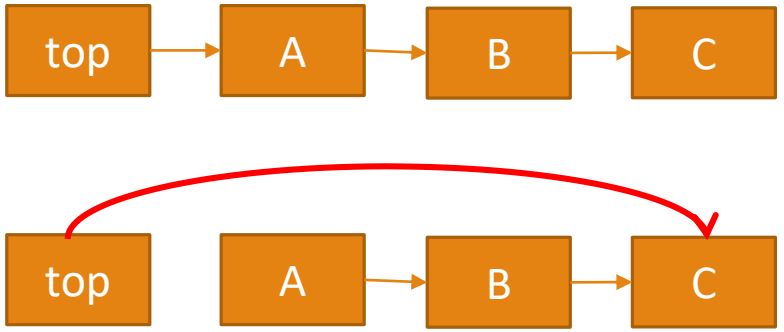
push(&top, &B)

push(&top, &A)



Before:

After:



Solutions

Devote part of each to-be-CAS ed word to a sequence number that is updated in pop on a successful CAS.

Two word long compare and swap operations

- Cmpexchg16b in Intel

Re-write the code to pass a push value, and had the method allocate a new node to hold it. Symmetrically, pop would deallocate the node and return the value it contained.

Use LL/SC. If the memory location received invalidation, SC will fail.

Examples of Hw. Sync. Primitives

- IBM 370: Compare and swap instruction
- Intel X86: Lock prefixes
- SPARC: swap involving a register and memory
- MIPS: LL/SC instructions
- HEP
- NYU Ultracomputer
- IBM RP3
- Illinois Cedar

Lock Prefixes on Intel

- Bus blocking
  - Hold the bus until load and store components are finished
- Cache blocking
  - Get exclusive ownership of data with cache coherence
  - Prevent other processor accessing the line in the cache

- "In the days of Intel 486 processors, the lock prefix used to assert a lock on the bus along with a large hit in performance.
- Starting with the Intel Pentium Pro architecture, the bus lock is transformed into a cache lock.
- A lock will still be asserted on the bus in the most modern architectures if the lock resides in uncacheable memory or if the lock extends beyond a cache line boundary splitting cache lines.
  - Both of these scenarios are unlikely, so most lock prefixes will be transformed into a cache lock which is much less expensive." *

*https://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures

HEP

- Each word in memory has Full/Empty (F/E) bit
- Bit is tested in hardware before a RD/WR if special symbol is prepended to the var name
- The RD/WR blocks until the test succeeds: RD until full WR until empty
  - Producer consumer type access
- When test succeeds, the bit is set to the opposite value, indivisibly with the RD/WR

HEP

Advantages:

- Very efficient for low level dependences (compare to locks)

Disadvantages:

- F/E bits
- Logic to initialize the bits
- Support to queue a process if test fails
- Logic to implement indivisible ops

NYU Ultracomputer

Atomic Fetch & Add:

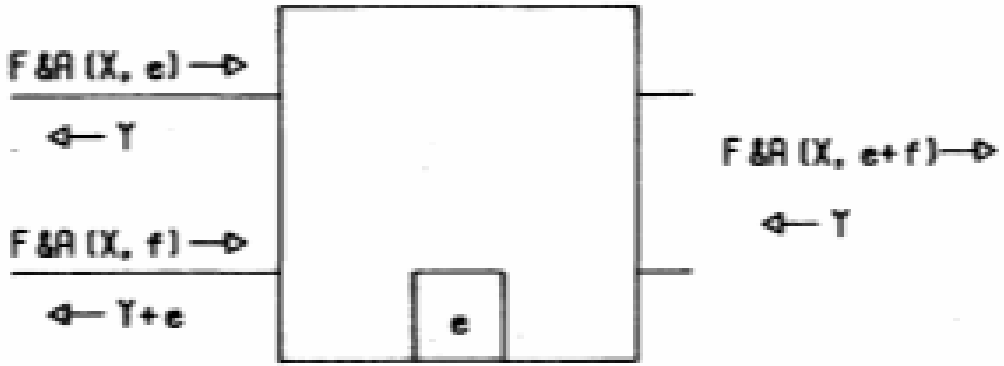
- Send a message to a memory location with a constant

Advantages:

- Useful in certain cases: get the next iteration of a loop
- If the network has hardware to combine messages to the same location, primitive tolerates contentions

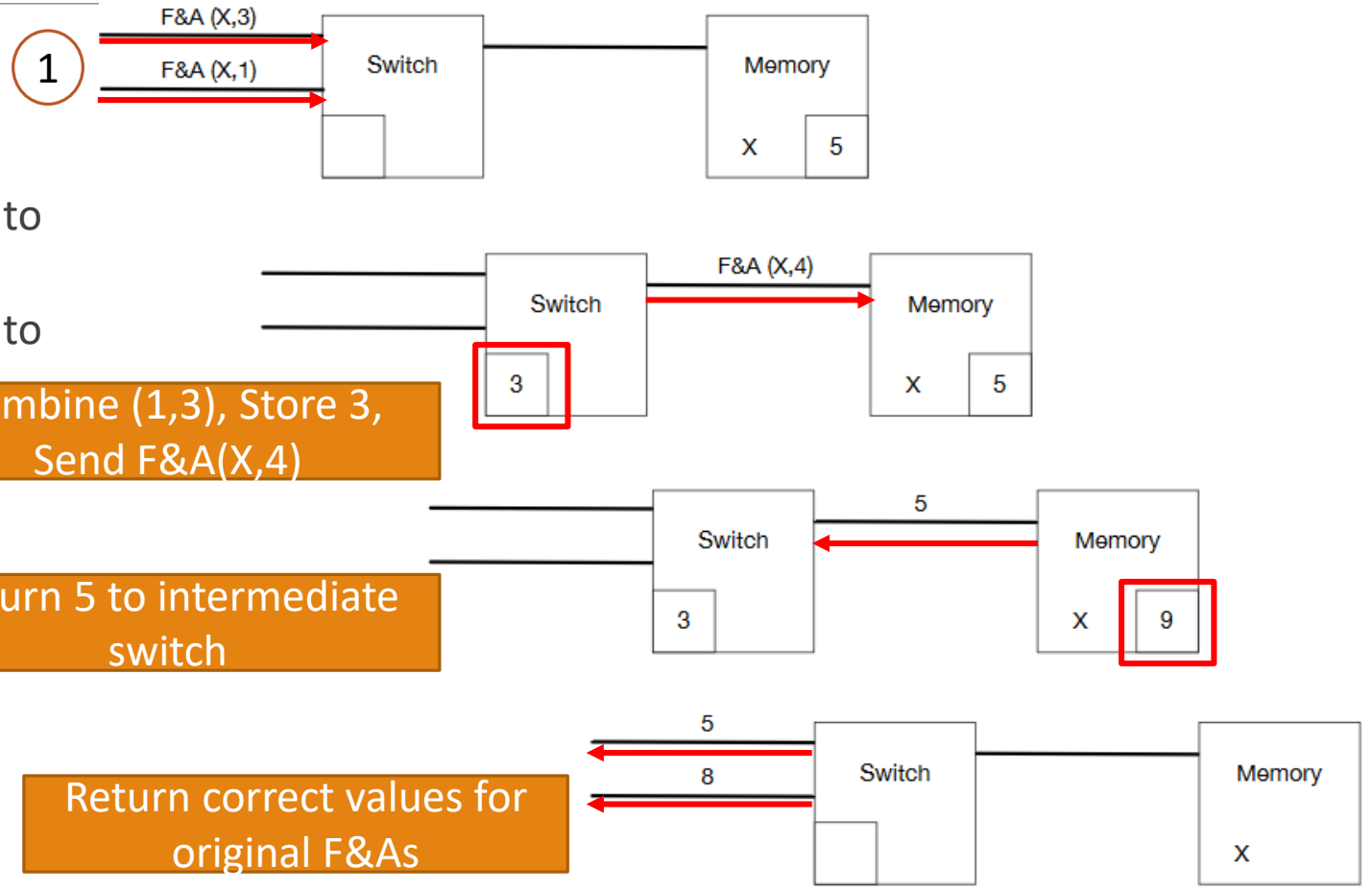
Message Combining

- Y← F&A(X, v)
- When two fetch-and-adds referencing the same shared variable, say F&A(X, e) and F&A(X,f), meet at a switch
  - the switch forms the sum e + f transmits the combined request F&A(X, e +f), and stores the value e in its local memory
- When the value Y is returned to the switch in response to F&A(X, e +f)
  - the switch transmits Y to satisfy the original request F&A(X, e)
  - and transmits Y + e to satisfy the original request F&A(X,f).



Example

- Initial value of X is 5
- Two F&As are executed
  - F&A(X,3): should return 5 and update X to be 8
  - F&A(X,1): should return 8 and update X to be 9



Message Combining

Advantages:

- Multiple requests in parallel
- Less traffic (scalable)
- Time complexity depends on the network not the number of parallel requests

Disadvantages:

- Very complex network
- Slows down the rest of the messages
- Complex hardware
  - For Fetch & Add: adder in each memory module
  - For message combining: Special, complex queuing logic at each switch in the network

IBM RP3

Atomic Fetch & Phi:

- Add, And, Or, Min, Max, Store, Store if zero
- Hardware required: logic in the shared memory to implement the 7 atomic operations

Illinois Cedar

Scheme to complement a vectorizing compiler by resolving data dependences at runtime

- More parallelism can be obtained from a program
- Targets data dependence in loops

```
      DO I1 = 1, 10
        DO I2 = 1, 10
S1        A(I1,I2) = B(I1+1,I2) + C(I1,I2)
S2        B(I1,I2) = A(I1-1,I2) * D(I1,I2)
        ENDDO
      ENDDO
```

Two solutions

- Execute outer loop in serial
- Distribute into two loops for S1 and S2

```
      DO I1 = 1, 10
        DO I2 = 1, 10
S1        A(I1,I2) = B(I1+1,I2) + C(I1.I2)
        ENDDO
      ENDDO

      DO I1 = 1, 10
        DO I2 = 1, 10
S2        B(I1,I2) = A(I1-1,I2) * D(I1,I2)
        ENDDO
      ENDDO
```

On HEP

- Mark A and B arrays to use Full/Empty bits
- In S1, read on B sets the bit, write on A sets the bit
- Operation of A:
  - On S1 check full/empty bit of A(I1, I2) → Is Empty?
    - If unset write the value and set the bit
  - On S2 check full/empty bit of A(I1-1, I2) → Is Full?
    - If set write the value and unset the bit
    - Otherwise wait until set

```
      DOALL I1 = 1, 10
        DOALL I2 = 1, 10
S1      #A(I1,I2) = #B(I1+1,I2) + C(I1,I2)
S2      #B(I1,I2) = #A(I1-1,I2) * D(I1,I2)
        ENDDOALL
      ENDDOALL
```

Illinois Cedar

- Assume that the synchronization is required on an array variable A.
  - Synchronization variables: Each data element of the synchronization variable A(I)
- General atomic instruction that operates on synchronization variables
- Synch var is 2 words: Key and Value
  - Operations on A(I).key: regular Fetch, Store, Increment, Decrement, Increment&Fetch, Decrement&Fetch, and No Action.
  - Operations on A(I).value: regular Fetch, Store, and NoAction.
- HW required: special processor at each mem module

Can derive more general atomic primitives. Check the paper for more examples.

{addr; (cond); op on key; op on value}
if * in condition: spin until true

{X; (X.key == 1)*; decrement; fetch}
this is F/E bit test for a read option

Notes on Paper

Three main features:

- A mechanism for first-come first-serve queueing. Reduces the complexity of acquiring a semaphore to O(N)
- No hardware FetchAndPhi operations in hardware
  - A software combining method is proposed
- A notify primitive implemented in hardware for global barrier completion

Assumptions of the paper:

- Shared memory processor with cache coherence
- Broadcast is supported in the interconnect
- Hardware combining is not implemented

Synchronization Primitives and `syncbits`

Each memory (cache) line is associated with a `syncbit`

Advantages:

- Synchronization memory allocated proportional to data memory
- Operations on `syncbits` can be implemented as extensions to cache coherence

Disadvantage:

- `Syncbits` are associated with a cache line. Two words requiring `syncbits` can't be mapped to the same line

Test&Set, Unset, Queue_on_Syncbit(QOSB)

QOSB: non-blocking operation on syncbit address that adds the issuing processor to the syncbit queue

Executes QOSB instruction before Test&Set operations

- If requesting processor is not in the queue, adds it to the queue
- Processor spins on Test&Set locally
- When syncbit is unset (head of the queue removed), next processor waiting in the queue is notified

Broadcast Notify: A restricted write broadcast to eliminate hotspot contention

Fetch-and-Phi: software combining method is used

Hardware Extensions/Operations for QOSB

QOSB instruction performs two operations

- It allocates space for a shadow copy of the line in the local cache with the shadow syncbit set. Allocates an entry in the queue
- It performs a remote access for getting exclusive copy of the data
- If a shadow copy exists in the local cache, no remote accesses performed

Two additional cache state

- Shadow: invalid copy in the queued processors
- Sticky: valid copy at the head of the queue
- In memory data invalid state

Interaction with Primitives

When a processor holding the line receives a QOSB

- Requesting processor is queued
- Note that shadow lines are invalid, data space for shadow lines can be used to keep track of pointers for the next element in the queue
- Note that memory line is invalid, memory line can be used to store head pointer

Test&Set:

- Fail if a local shadow copy exists

Unset:

- It removes the head of queue.
- Initiate transfer for the next element in the queue

References

- Section 5.6 of Culler&Singh Book
  - Small, bus-based shared memory machines
- Shared Memory Synchronization (Michael L. Scott) M&C Synthesis Lectures in Computer Architecture
- J. Goodman et al. "Efficient Synchronization Primitives for Large Scale Cache-Coherent Multiprocessors". ASPLOS 1989.
- Gottlieb, Allan et al. “NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer”. IEEE Transactions on Computers 1983.
- Chuan-Qi Zhu et al. “A Scheme to Enforce Data Dependence on Large Multiprocessor Systems”. IEEE Transactions on Software Engineering 1987.
- Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures. https://software.intel.com/en-us/articles/implementing-scalable-atomic-locks-for-multi-core-intel-em64t-and-ia32-architectures