EECS 570
Lecture 6
Synchronization

Winter 2016
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinharddt, Roth, Smith, Singh, and Wenisch.
Readings

For Today:
- Michael Scott. *Shared-Memory Synchronization*. Morgan & Claypool Synthesis Lectures on Computer Architecture (Ch. 1, 4.0-4.3.3, 5.0-5.2.5).

For Monday:
Why study database apps?

- They are economically important

- They share characteristics of many other apps (filesystems, web search, etc.)

- The vendors have spent a lot of time optimizing (generally, they won’t have silly bottlenecks)
Key characteristics

• Large, complex, monolithic software systems

• Designed for MP systems
  □ Clusters (distributed databases)
  □ Shared Memory

• Subsumes many OS functions
  □ File system
  □ Scheduling and multi-threading
  □ Memory management

• Designed for high reliability (ACID properties)
  □ Atomicity: a transaction happens or doesn’t
  □ Consistency: the state of the DB remains consistent
  □ Isolation: transactions are independent
  □ Durability: once performed, transactions are permanent
  □ Aside: we will see these ideas pop up in architecture context again with transactional memory
How are they different from Sci Apps?

- Requires tuning: knowledge-intensive, difficult
- Competitive market: deliberate obfuscation/ benchmark gaming
- Large instruction footprints (I$ matters)
- Huge data footprints (TLBs matter)
- Weird access types (cross-endian, non-cacheable, etc.)
- Latency, not bandwidth bound
- Dynamic memory allocation, sometimes garbage collection
- More pointer-chasing, fewer arrays
- No single obvious “working set”
  - multiple working sets with varying temporal locality
- Unpredictable sharing patterns
- Data & lock contention
DBMS Structure

Source: Silberschatz, Korth, Sudarshan. Database System Concepts
Fundamental Data Structures

**Page**

<table>
<thead>
<tr>
<th>PAGE HEADER</th>
<th>RH1</th>
<th>0962</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>30</td>
<td>RH2</td>
</tr>
<tr>
<td>45</td>
<td>RH3</td>
<td>3589</td>
</tr>
<tr>
<td>Susan</td>
<td>5523</td>
<td>52</td>
</tr>
</tbody>
</table>

Source: Ailamaki, DeWitt & Hill

**B+ Tree**

Where does time go: Microbenchmarks

- Compute time < 50% of total time

Where does time go: Memory stalls breakdown

- L1 instruction and L2 data stalls dominate

Standardized Benchmarks

- **Transaction Processing Council (TPC)**
  - Strict scaling, disclosure, auditing rules
  - Running these for real is hard: big hardware, 20-50 engineers, months of effort
  - Running them in simulations is also hard: scaling, non-determinism

- **Two flavors of benchmark**
  - Online transaction processing (OLTP): TPC-C
    - Lots of small transactions
    - Lots of locking, concurrency, I/O; memory-latency bound
  - Decision support system (DSS): TPC-H
    - Large, complex read-only queries
    - Often compute bound (given enough disks)
    - Highly parallel
      - Data partitioning
      - Parallel operators
Performance of DB Workloads on Shared Memory with OoO CPUs

[Ranganathan et al - ISCA 98]

Examines impact of ILP and multiprocessing on DSS & OLTP

• Based on extensive simulations

• Explores:
  - Multiple issue
  - Out-of-order (including window size)
  - Number of outstanding misses
  - Instruction/branch prediction effects
  - Impact of multiprocessing & memory consistency
  - Ways to mitigate instruction & coherence misses
• Lots of ILP
• Multiple issue helps, but OoO helps more
• L2 hits account for most data stalls
• Multiple outstanding misses are critical
OLTP - Impact of ILP

- Instruction cache misses & synch are now issues
- Less ILP, but 2-way still helps a lot
- Coherence (dirty) & DTLB misses cause most read stalls
- 2 outstanding misses is critical, but more doesn’t help
Impact of Multiprocessing

- Coherence misses & sync are dramatic in OLTP
Impact of Memory Consistency

- SC = sequential consistency
- PC = a system with a write buffer (loads bypass stores)
- RC = wait only at synchronization instructions

- Massive performance difference!
  - We will revisit this later in the course...
Cloud Computing Software (scale-out)
What about “cloud computing” software?
[Ferdman et al - ASPLOS 2012]

Emerging workloads:
• Scale out
• Often data intensive
• Like conventional server workloads

Different from CPU benchmark suites:
• Use of FP
• Do not exercise the memory hierarchy
• Similar to conventional server workloads [CIDR’07]
CloudSuite: A Benchmark Suite of Emerging Scale-Out Workloads

Publicly released

Alpha version:
- Analytics (Classification)
- Data serving (YCSB)
- Simulation (Cloud9)
- Streaming (Darwin)
- Web frontend (Cloudstone)
- Web search (Nutch)
Ran Experiments on Nehalem Blades

<table>
<thead>
<tr>
<th>Hardware Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
</tr>
<tr>
<td>Intel Xeon 5670, 6 cores, 32nm @2.93GHz</td>
</tr>
<tr>
<td><strong>CMP Size</strong></td>
</tr>
<tr>
<td>6 OoO cores</td>
</tr>
<tr>
<td><strong>Superscalar width</strong></td>
</tr>
<tr>
<td>4-wide issue</td>
</tr>
<tr>
<td><strong>Reorder buffer</strong></td>
</tr>
<tr>
<td>128 entries</td>
</tr>
<tr>
<td><strong>Load/Store buffer</strong></td>
</tr>
<tr>
<td>48/32 entries</td>
</tr>
<tr>
<td><strong>Reservation stations</strong></td>
</tr>
<tr>
<td>36 entries</td>
</tr>
<tr>
<td><strong>L1 Cache</strong></td>
</tr>
<tr>
<td>split I/D, 32KB, 4-cycles access latency</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
</tr>
<tr>
<td>6-core CMP: 256KB per core, 12-cycles access latency</td>
</tr>
<tr>
<td><strong>LLC (L3) cache</strong></td>
</tr>
<tr>
<td>12MB, cycles 39-cycles access latency</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td>24GB, 180/280 cycles access latency local/remote DRAM</td>
</tr>
</tbody>
</table>

*Where does time go?*
• Unlike desktop/RMS apps, memory stalls dominate
• Design should be centered around memory
Front-End Inefficiencies

- Instruction fetch: 10-60% of total stalls
- Next-line prefetch. (in the CPU) not efficient
Core Inefficiencies

- Low IPC & MLP despite 4-wide OoO core
- Using SMT doubles MLP
- But, SMT achieves only 30% performance gain
  - Threads compete for core resources
  - Intel’s SMT fetch not effective
Cache Capacity (LLC) Inefficiencies

- Large LLC consumes area, but has diminishing returns
- Results (not shown) indicate much LLC accesses are instructions
Data Prefetching Inefficiencies

- Existing prefetchers are ineffective
- Pointer-intensive patterns [Wenisch 2005]
Bandwidth Inefficiencies

- Low sharing among working threads
- No need for on-chip shared caches
- Today, pin bandwidth is overprovisioned
CloudSuite Conclusions

Corroborate prior findings [CIDR’07]
Scale-out workloads need:
• Simple (multithreaded) cores
• Partitioned caches (no sharing)
• Large on-chip instruction footprints
• Advanced prefetchers
Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers

Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, Jason Mars

University of Michigan — Ann Arbor, MI
• **Sirius**: full end-to-end with inputs, pre-trained models, and databases
• **Sirius-suite**: 7 kernels with inputs to study each service individually
How does Sirius work?

Query Taxonomy
- Voice Command (VC)
- Voice Query (VQ)
- Voice-Image Query (VIQ)

IPA Services
- Automatic-Speech Recognition (ASR)
- Question Answering (QA)
- Image Matching (IMM)

Algorithmic Components
- HMM/GMM or HMM/DNN
- Regular Expression
- Stemmer
- Feature Extraction
- Feature Description

HMM/GMM or HMM/DNN
Conditional Random Fields

Sirius: An Open End-to-End Voice and Vision Personal Assistant
Sirius-suite

Automatic-Speech Recognition (ASR)  Question Answering (QA)  Image Matching (IMM)
GMM (85%)  Stemmer (46%)  FE (41%)
DNN (78%)  Regex (22%)  FD (56%)
CRF (17%)

IPA Services

7 kernels: 92% total execution of Sirius Suite entirely written in C/C++/CUDA
Release includes inputs and models

DjiNN and Tonic: DNN as a Service
## Upgrading Datacenters with COTS Systems

<table>
<thead>
<tr>
<th>Platform</th>
<th>Model</th>
<th>Clock</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore CPU</td>
<td>Intel Xeon E3-1240 V3</td>
<td>3.40 GHz</td>
<td>8</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA GTX 770</td>
<td>1.05 GHz</td>
<td>12288</td>
</tr>
<tr>
<td>Intel Phi</td>
<td>Phi 5110P</td>
<td>1.05 GHz</td>
<td>240</td>
</tr>
<tr>
<td>FPGA</td>
<td>Xilinx Virtex-6 ML605</td>
<td>400 MHz</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## Upgrading Datacenters with COTS Systems

<table>
<thead>
<tr>
<th>Platform</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore CPU</td>
<td>Minor SW changes</td>
<td>Limited speedup</td>
</tr>
<tr>
<td>GPU</td>
<td>Many threads</td>
<td>Programability</td>
</tr>
<tr>
<td>Intel Phi</td>
<td>Manycore</td>
<td>Limited compiler support</td>
</tr>
<tr>
<td>FPGA</td>
<td>Flexible</td>
<td>New implementation</td>
</tr>
</tbody>
</table>
Acceleration Overview

<table>
<thead>
<tr>
<th>Platform</th>
<th>GMM</th>
<th>DNN</th>
<th>Stemmer</th>
<th>Regex</th>
<th>CRF</th>
<th>FE</th>
<th>FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>3.5</td>
<td>6.0</td>
<td>4.0</td>
<td>3.9</td>
<td>3.7</td>
<td>5.2</td>
<td>5.9</td>
</tr>
<tr>
<td>GPU</td>
<td>70.0</td>
<td>54.7</td>
<td>6.2</td>
<td>48.0*</td>
<td>3.8*</td>
<td>10.5</td>
<td>120.5</td>
</tr>
<tr>
<td>Intel Phi</td>
<td>1.1</td>
<td>11.2</td>
<td>5.6</td>
<td>1.1</td>
<td>4.7</td>
<td>2.5</td>
<td>12.7</td>
</tr>
<tr>
<td>FPGA</td>
<td>169.0</td>
<td>110.5*</td>
<td>30.0</td>
<td>168.2*</td>
<td>7.5*</td>
<td>34.6*</td>
<td>75.5*</td>
</tr>
</tbody>
</table>
DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers

Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, Lingjia Tang

University of Michigan — Ann Arbor, MI
Deep Neural Networks (DNNs)

Network Architecture

DjiNN and Tonic: DNN as a Service
**Tonic Suite Applications**

- **Image Task**
  - IMC
  - DIG
  - FACE

- **Speech Recognition (ASR) Task**
  - "It's business, Superman"

- **Natural Language Processing Task**
  - **POS**
    - "business" (noun)
    - "Superman" (P. noun)
  - **CHK**
    - "it's" (VP, B-NP)
  - **NER**
    - "business" (NP, I-NP)
    - "Superman" (PERSON)

**DjiNN DNN Service**

- DNN Architecture
- Trained Models
  - IMC
  - DIG
  - FACE
  - ASR
  - POS
  - CHK
  - NER

---

**DjiNN and Tonic: DNN as a Service**
DNN as a Service

Image Classification

Digit Recognition

Facial Recognition

Speech Recognition

Natural Language Processing

Unified, highly optimized appliance for DNN

DjiNN and Tonic: DNN as a Service
Synchronization
Synchronization objectives

• Low overhead
  ❑ Synchronization can limit scalability
    (E.g., single-lock OS kernels)

• Correctness (and ease of programmability)
  ❑ Synchronization failures are extremely difficult to debug

• Coordination of HW and SW
  ❑ SW semantics must be tightly specified to prove correctness
  ❑ HW can often improve efficiency
Synchronization Forms

• Mutual exclusion (critical sections)
  □ Lock & Unlock

• Event Notification
  □ Point-to-point (producer-consumer, flags)
  □ I/O, interrupts, exceptions

• Barrier Synchronization

• Higher-level constructs
  □ Queues, software pipelines, (virtual) time, counters

• Next lecture: optimistic concurrency control
  □ Transactional Memory
Anatomy of a Synchronization Op

• Acquire Method
  ❑ Way to obtain the lock or proceed past the barrier

• Waiting Algorithm
  ❑ Spin (aka busy wait)
    ❑ Waiting process repeatedly tests a location until it changes
    ❑ Releasing process sets the location
    ❑ Lower overhead, but wastes CPU resources
    ❑ Can cause interconnect traffic
  ❑ Block (aka suspend)
    ❑ Waiting process is descheduled
    ❑ High overhead, but frees CPU to do other things
  ❑ Hybrids (e.g., spin, then block)

• Release Method
  ❑ Way to allow other processes to proceed
HW/SW Implementation Trade-offs

- User wants high-level (ease of programming)
  - LOCK(lock_variable); UNLOCK(lock_variable)
  - BARRIER(barrier_variable, numprocs)
- SW advantages: flexibility, portability
- HW advantages: speed
- Design objectives:
  - Low latency
  - Low traffic
  - Low storage
  - Scalability (“wait-free”-ness)
  - Fairness
Challenges

- Same sync may have different behavior at different times
  - Lock accessed with low or high contention
  - Different performance needs: low latency vs. high throughput
  - Different algorithms best for each, need different primitives

- Multiprogramming can change sync behavior
  - Process scheduling or other resource interactions
  - May need algorithms that are worse in dedicated case

- Rich area of SW/HW interactions
  - Which primitives are available?
  - What communication patterns cost more/less?
Locks
Lock-based Mutual Exclusion

Synchronization period

- Acquire starts
- Acquire done
- Release starts
- Release done

No contention:
- Want low latency

Contention:
- Want low period
- Low traffic
- Fairness
How Not to Implement Locks

• LOCK
  while (lock_variable == 1);
  lock_variable = 1

• UNLOCK
  lock_variable = 0;

Context switch!
Solution: Atomic Read-Modify-Write

- Test&Set(r,x)
  \{ r=m[x]; m[x]=1; \}
  \( r \) is register
  \( m[x] \) is memory location \( x \)

- Fetch&Op(r1,r2,x,op)
  \{ r1=m[x]; m[x]=op(r1,r2); \}

- Swap(r,x)
  \{ temp=m[x]; m[x]=r; r=temp; \}

- Compare&Swap(r1,r2,x)
  \{ temp=r2; r2=m[x]; if r1==r2 then m[x]=temp; \}
Implementing RMWs

• Bus-based systems:
  - Hold bus and issue load/store operations without any intervening accesses by other processors

• Scalable systems
  - Acquire exclusive ownership via cache coherence
  - Perform load/store operations without allowing external coherence requests
Load-Locked Store-Conditional

• Load-locked
  □ Issues a normal load...
  □ ...and sets a flag and address field

• Store-conditional
  □ Checks that flag is set and address matches...
  □ ...only then performs store

• Flag is cleared by
  □ Invalidation
  □ Cache eviction
  □ Context switch

```
lock: while (1) {
    load-locked r1, lock_variable
    if (r1 == 0) {
        mov r2 = 1
        if (SC r2, lock) break;
    }
}  
unlock: st lock_variable, #0
```
Test-and-Set Spin Lock (T&S)

• Lock is “acquire”, Unlock is “release”

  • acquire(lock_ptr):
    while (true):
      // Perform “test-and-set”
      old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
      if (old == UNLOCKED):
        break   // lock acquired!
      // keep spinning, back to top of while loop

  • release(lock_ptr):
    store[lock_ptr] <- UNLOCKED

• Performance problem
  □ CAS is both a read and write; spinning causes lots of invalidations
Test-and-Test-and-Set Spin Lock (TTS)

- **acquire(lock_ptr):**
  
  ```
  while (true):
    // Perform “test”
    load [lock_ptr] -> original_value
    if (original_value == UNLOCKED):
      // Perform “test-and-set”
      old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
      if (old == UNLOCKED):
        break  // lock acquired!
    // keep spinning, back to top of while loop
  ```

- **release(lock_ptr):**
  
  ```
  store[lock_ptr] <- UNLOCKED
  ```

- **Now “spinning” is read-only, on local cached copy**
• Performance issues remain
  □ Every time the lock is released...
  □ All the processors load it, and likely try to CAS the block
  □ Causes a storm of coherence traffic, clogs things up badly

• One solution: backoff
  □ Instead of spinning constantly, check less frequently
  □ Exponential backoff works well in practice

• Another problem with spinning
  □ Processors can spin really fast, starve threads on the same core!
  □ Solution: x86 adds a “PAUSE” instruction
    ○ Tells processor to suspend the thread for a short time

• (Un)fairness
Ticket Locks

• To ensure fairness and reduce coherence storms

• Locks have two counters: next_ticket, now_serving
  □ Deli counter

• acquire(lock_ptr):
  □ my_ticket = fetch_and_increment(lock_ptr->next_ticket)
  □ while(lock_ptr->now_serving != my_ticket); // spin

• release(lock_ptr):
  □ lock_ptr->now_serving = lock_ptr->now_serving + 1
    ○ (Just a normal store, not an atomic operation, why?)

• Summary of operation
  □ To “get in line” to acquire the lock, CAS on next_ticket
  □ Spin on now_serving
Ticket Locks

• Properties
  - Less of a “thundering herd” coherence storm problem
    - To acquire, only need to read new value of now_serving
  - No CAS on critical path of lock handoff
    - Just a non-atomic store
  - FIFO order (fair)
    - Good, but only if the O.S. hasn’t swapped out any threads!

• Padding
  - Allocate now_serving and next_ticket on different cache blocks
    - struct { int now_serving; char pad[60]; int next_ticket; } ...
  - Two locations reduces interference

• Proportional backoff
  - Estimate of wait time: (my_ticket - now_serving) * average hold time
Array-Based Queue Locks

• Why not give each waiter its own location to spin on?
  □ Avoid coherence storms altogether!

• Idea: “slot” array of size N: “go ahead” or “must wait”
  □ Initialize first slot to “go ahead”, all others to “must wait”
  □ Padded one slot per cache block,
  □ Keep a “next slot” counter (similar to “next_ticket” counter)

• Acquire: “get in line”
  □ my_slot = (atomic increment of “next slot” counter) mod N
  □ Spin while slots[my_slot] contains “must_wait”
  □ Reset slots[my_slot] to “must wait”

• Release: “unblock next in line”
  □ Set slots[my_slot+1 mod N] to “go ahead”
Array-Based Queue Locks

- Variants: Anderson 1990, Graunke and Thakkar 1990

- Desirable properties
  - Threads spin on dedicated location
    - Just two coherence misses per handoff
    - Traffic independent of number of waiters
  - FIFO & fair (same as ticket lock)

- Undesirable properties
  - Higher uncontended overhead than a TTS lock
  - Storage $O(N)$ for each lock
    - 128 threads at 64B padding: 8KBs per lock!
    - What if $N$ isn’t known at start?

- List-based locks address the $O(N)$ storage problem
  - Several variants of list-based locks: MCS 1991, CLH 1993/1994
List-Based Queue Lock (MCS)

• A “lock” is a pointer to a linked list node
  - next node pointer
  - boolean must_wait
  - Each thread has its own local pointer to a node “I”

• acquire(lock):
  I->next = null;
  predecessor = fetch_and_store(lock,I)
  if predecessor != nil //some node holds lock
    I->must_wait = true
    predecessor->next = I //predecessor must
    wake us
  repeat while I->must_wait //spin till lock is free

• release(lock):
  if (I->next == null) //no known successor
    if compare_and_swap(lock,I,nil) //make sure…
      return //CAS succeeded; lock freed
    repeat while I->next = nil //spin to learn successor
    I->next->must_wait = false //wake successor
MCS Lock Example: Time 0

- acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait

- release(lock):
  if (I->next == null)
    if CAS(lock,I,nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false
MCS Lock Example: Time 1

- $t_1$: Acquire(L)

- acquire(lock):
  
  ```
  I->next = null;
  pred = FAS(lock, I)
  if pred != nil
      I->must_wait = true
  pred->next = I
  repeat while I->must_wait
  ```

- release(lock):
  
  ```
  if (I->next == null)
      if CAS(lock, I, nil)
          return
  repeat while I->next == nil
  I->next->must_wait = false
  ```
MCS Lock Example: Time 2

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)

### acquire(lock):
- $I->next = null$
- pred = FAS(lock, I)
  - if pred != nil
    - $I->must\_wait = true$
    - pred->next = I
  - repeat while $I->must\_wait$

### release(lock):
- if (I->next == null)
  - if CAS(lock, I, nil)
    - return
  - repeat while I->next == nil
  - $I->next->must\_wait = false$
MCS Lock Example: Time 3

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)
- $t_3$: Acquire(L)

```plaintext
• acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait

• release(lock):
  if (I->next == null)
    if CAS(lock,I,nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false
```
MCS Lock Example: Time 4

- $t_1$: Acquire($L$)
- $t_2$: Acquire($L$)
- $t_3$: Acquire($L$)
- $t_1$: Release($L$)

- **acquire(lock):**
  
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait

- **release(lock):**
  
  if (I->next == null)
    if CAS(lock,I,nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false
MCS Lock Example: Time 5

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)
- $t_3$: Acquire(L)
- $t_1$: Release(L)
- $t_2$: Release(L)

- acquire(lock):
  $I->next = \text{null}$;
  pred = FAS(lock,$I$)
  if pred ! = nil
    $I->\text{must\_wait} = \text{true}$
  pred->next = $I$
  repeat while $I->\text{must\_wait}$

- release(lock):
  if ($I->next$ == null)
    if CAS(lock,$I$,nil)
      return
  repeat while $I->next$ == nil
  $I->next->\text{must\_wait} = \text{false}$
MCS Lock Example: Time 6

- $t_1$: Acquire($L$)
- $t_2$: Acquire($L$)
- $t_3$: Acquire($L$)
- $t_1$: Release($L$)
- $t_2$: Release($L$)
- $t_3$: Release($L$)

\begin{itemize}
  \item acquire($lock$):
    \begin{verbatim}
      I->next = null;
      pred = FAS(lock,I)
      if pred != nil
        I->must_wait = true
      pred->next = I
      repeat while I->must_wait
    \end{verbatim}
  \item release($lock$):
    \begin{verbatim}
      if (I->next == null)
        if CAS(lock,I,nil)
          return
      repeat while I->next == nil
      I->next->must_wait = false
    \end{verbatim}
\end{itemize}

release() w/o CAS is more complex; see paper
Queue-based locks in HW: QOLB

- **Queue On Lock Bit**
  - HW maintains doubly-linked list between requesters
    - This is a key idea of “Scalable Coherence Interface”, see Unit 3
  - Augment cache with “locked” bit
    - Waiting caches spin on local “locked” cache line
  - Upon release, lock holder sends line to 1st requester
    - Only requires one message on interconnect
## Fundamental Mechanisms to Reduce Overheads

[Kägi, Burger, Goodman ASPLOS 97]

- **Basic mechanisms**
  - Local Spinning
  - Queue-based locking
  - Collocation
  - Synchronous Prefetch

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Local Spin</th>
<th>Queue</th>
<th>Collocation</th>
<th>Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>No</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>Yes</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>Yes</td>
<td>Optional</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

![Graph showing Relative sync period vs # CPUs for T&S, T&T&S, MCS, and QOLB. The graph includes a reference to Kägi 97.]
Performance of Locks

• Contention vs. No Contention
  □ Test-and-Set best when no contention
  □ Queue-based is best with medium contention
  □ Idea: switch implementation based on lock behavior
    ○ Reactive Synchronization – Lim & Agarwal 1994
    ○ SmartLocks – Eastep et al 2009

• High-contention indicates poorly written program
  □ Need better algorithm or data structures
Point-to-Point Event Synchronization

- Can use normal variables as flags
  
  ```
  a = f(x);
  flag = 1;
  ```
  
  ```
  while (flag == 0);
  ```
  
  ```
  b = g(a);
  ```

- If we know initial conditions
  
  ```
  a = f(x);
  ```
  
  ```
  while (a == 0);
  ```
  
  ```
  b = g(a);
  ```

- Assumes Sequential Consistency!

- Full/Empty Bits
  - Set on write
  - Cleared on read
  - Can’t write if set, can’t read if clear
Barriers
Barriers

- Physics simulation computation
  - Divide up each timestep computation into N independent pieces
  - Each timestep: compute independently, synchronize

- Example: each thread executes:

  ```
  segment_size = total_particles / number_of_threads
  my_start_particle = thread_id * segment_size
  my_end_particle = my_start_particle + segment_size - 1
  for (timestep = 0; timestep += delta; timestep < stop_time):
    calculate_forces(t, my_start_particle, my_end_particle)
    barrier()
    update_locations(t, my_start_particle, my_end_particle)
    barrier()
  ```

- Barrier? All threads wait until all threads have reached it
Example: Barrier-Based Merge Sort

Step 1

Step 2

Step 3
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it

- Strawman implementation (wrong!)

```
global (shared) count : integer := P

procedure central_barrier
  if fetch_and_decrement(&count) == 1
    count := P
  else
    repeat until count == P
```

- What is wrong with the above code?
Sense-Reversing Barrier

• Correct barrier implementation:

```
global (shared) count : integer := P
global (shared) sense : Boolean := true
local (private) local_sense : Boolean := true

procedure central_barrier
    // each processor toggles its own sense
    local_sense := !local_sense
    if fetch_and_decrement(&count) == 1
        count := P
        // last processor toggles global sense
        sense := local_sense
    else
        repeat until sense == local_sense

• Single counter makes this a “centralized” barrier
```
Other Barrier Implementations

• Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - O(n) if threads arrive simultaneously, slow for lots of processors

• Combining Tree Barrier
  - Build a log_k(n) height tree of counters (one per cache block)
  - Each thread coordinates with k other threads (by thread id)
  - Last of the k processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - O(log n) in best case

• Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release