Synchronization (with a focus on J2SE)

Dave Dice - 2005.10.18

Overview

- Preliminaries: Parallelism & synchronization
- Java synchronization implementations
- Uncontended synchronization
- Contended synchronization
- Futures – where are we headed

Parallelism – Varieties (1)

- Distributed parallelism
  - Communicate over a network (clusters, etc)
  - High communication latency
  - Code typically failure-aware
  - Message-based : send-recv – OS IPC primitives
- SMP parallelism with explicit threads
  - Communicate through shared memory – LD,ST,Atomics
  - Low(er) communication latency
  - Code not designed failure tolerant
- Common theme but different philosophies
  ➔ Thread Level Parallelism (TLP)

Parallelism – Varieties (2)

- SIMD – SSE
- Instruction Level Parallelism (ILP) – Multi-Issue
  - Execution time: Out-of-Order
    - Recognize and extract ||ism from independent ops in serial instruction stream
    - Intel P4 and P6
  - Execution time: HW scout threads
  - Compile-time auto-parallelize: Transform ILP → TLP (*hard*)
  - VLIW: IA64 compiler expresses ||ism to CPU

Parallelism – Dogma

- Explicit Parallelism is NOT a feature – it's a remedy with undesirable side effects
- Pick one: 1x4Ghz system or 4x1Ghz system?
  - Ideal -- Assume IPC = 1.0
  - 1X is the clear winner
  - Not all problems decompose (can be made ||)
  - Threads awkward and error-prone
- Pull your sled
  - 4 dogs – 2x2 Opteron
  - 32 Cats - (p.s., we're the cat trainers)

Processor Design Trends (1)

- These trends affect threading and synchronization
- Classic Strategy: Improve single-CPU performance by increasing core clock frequency
- Diminishing returns – *memory wall*
  - Physical constraints on frequency (+ thermal, too)
  - RAM is 100s cycles away from CPU
  - Cache miss results in many wasted CPU cycles
  - **RAM *bandwidth* & capacity keep pace with clock but *latency* is slower to improve** (disks,too)
- Compensate for processor:RAM gap with Latency Hiding …

Processors: Latency Hiding (2)

- Bigger caches – restricted benefit
  - Itanium2 L3$ consumes >50% die area
  - Deeper hierarchy – L4
- Speculative Execution – prefetch memory needed in future
  - Constrained – can't accurately speculate very far ahead
- ILP: try to do more in one cycle: IPC > 1
  - Deep pipelines
  - Superscalar and Out-of-Order Execution (OoO)
  - Allow parallel execution while pending miss
  - Limited ILP in most instructions streams - dependencies
- Double die area dedicated to extracting ILP ➔ < 1.2x better
- RESULT: Can't keep CPU fed – most cycles @ 4GHz are idle, waiting memory
- 1T speed effectively capped for near future

Processor Design Trends (3)

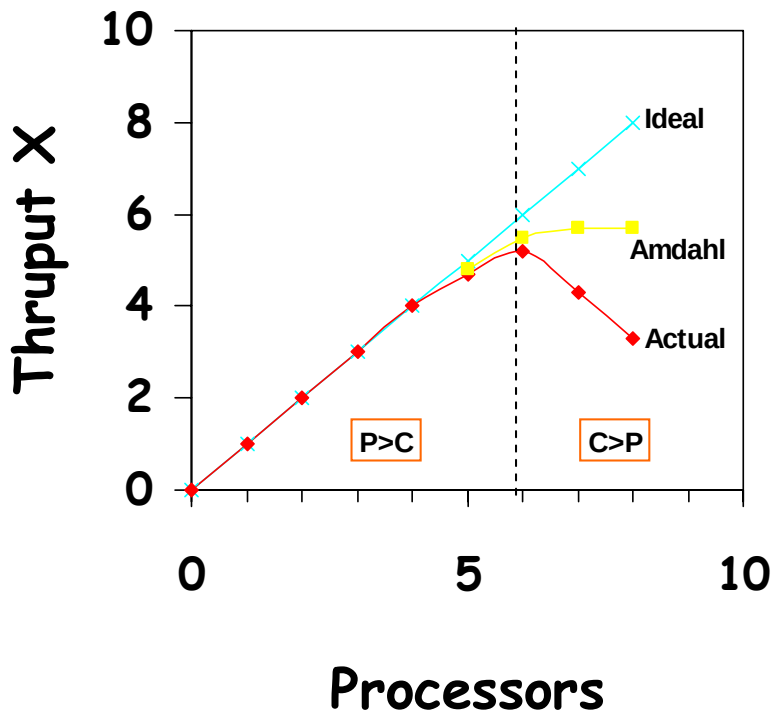
- Trend: wide parallelism instead of GHz
- Laptops & embedded (2→4) servers (32→256)
- Aggregate system throughput vs single-threaded speed (Sun Speak = Throughput computing)
- Impact on Software
  - wanted speed; got parallelism
  - Can we use 256 cores?
  - Previously: regular CPU speed improvements covered up software bloat
  - Future: The free lunch is over (Herb Sutter)
  - Amdahl's law – hard to realize benefit from ||ism
    - Even small critical sections dominate as the # of CPUs increases
  - Effort required to exploit parallelism
  - Near term: explicit threads & synchronization

Classic SMP Parallelism

- Model: set of threads that cooperate
- Coordinate threads/CPUs by *synchronization*
  Communicate through **coherent** shared memory
- **Definition**: synchronization is a mechanism prevent, avoid or recover from inopportune interleavings
- Control **when**
- Locking (mutual exclusion) is a special case of synchronization
- Social tax – cycles spent in synchronization don't contribute to productive work. Precautionary
- True parallelism vs preemption
  - Absurdity: OS on UP virtualizes MP system, forces synchronization tax

SMP Parallelism (2)

- Coordination costs: Synchronization + cache-coherent communication
- Coherency bus speed slower than CPU speed
- Most apps don't scale perfectly with CPUs
- Beware the classic pathology for over-parallel apps …



SMP Parallelism (3)

- Amdahl's speedup law – parallel corollary:
  - Access to shared state must be serialized
  - Serial portion of operation limits ultimate parallel speedup
  - Theoretical limit = 1/s
- Amdahl's law say curve should have flattened – asymptotic
- Why did performance Drop?
- In practice communication costs overwhelmed parallelism benefit
- Amdahl's law doesn't cover communication costs!
- Too many cooks spoil the broth

Fundamental Economics

- The following drive our performance-related design decisions:
  - Instruction latency – LD/ST/CAS/Barrier
  - CPU-Local Reordering & store-buffer
    - Program order vs visibility (memory) order
  - Coherence and communication costs
    - Caches and cache coherency
    - SMP sharing costs
    - Interconnect: Latency and bandwidth
      - Bandwidth ➔ overall scalability : throughput
  - Operating system context switching

Fundamental Economics (2)

- CAS local latency – 100-500+ cycles on OoO
- MEMBAR is expensive, but typically << CAS
  - IBM Power5 – existence proof that fences *can* be fast. 3x cycle improvement over Power4
- CAS and MEMBAR are typically **CPU-Local** operations
- CAS trend *might* be reversing – Intel Core2
  - Improved CAS and MFENCE
  - MFENCE must faster than CAS

Fundamental Economics (3)

- Context switching and blocking are expensive
  - 10k-80k cycles
  - Voluntary (park) vs involuntary (preemption)
  - Cache-Reload Transient (CRT) after the thread is dispatched (ONPROC)
  - Repopulate $ and TLBs
  - Lots of memory bandwidth consumed (reads)
  - Can impede other threads on SMP - scalability
  - Impact depends on system topology: Memory:Bus:$:CPU relationship

Fundamental Economics (3)

- Context switching continued …
  - Long quantum is usually better:
    - Reduces involuntary context switching rate
    - Dispadmin tables
    - On Solaris lower priority → longer quantum
  - Thread migration
    - Migration is generally bad
    - OS tries to schedule for affinity
    - Solaris rechoose interval

(ASIDE) Solaris Scheduler (1)

- Kernel schedules LWPs (LWP = Kernel thread)
- Process agnostic
- Scheduling classes: IA,TS, RT, FX, Fair-share
  - Priocntl and priocntl()
- Per CPU (die) ready queues (runq, dispatch queue)
- Avoid imbalance:
  - Pull : Stealing when idle
  - Push: Dealing when saturated
- Decentralized – no global lists, no global decision making
  - Except for RT
- Goals of scheduling policy
  - Saturate CPUs – maximize utilization
  - Affinity preserving – minimize migration
  - Disperse over cores
  - Power

(ASIDE) Solaris Scheduler (2)

- Local scheduling decisions collectively result in achieving the desired global scheduling policy
- Thus no gang scheduling
- Historical: libthread "T1" performed preemptive user-land scheduling
  - Multiplexed M user threads on N LWPs

Coherency and Consistency

- Cache *coherence*
  - Replicate locally for speed – reduce communication costs
  - … but avoid stale values on MP systems
  - Coherence is a property of a single location
- Memory *Consistency* (*Memory Model*)
  - Property of 2 or more locations
  - Program order vs Visibility (memory) order
  - A memory consistency model defines observable orderings of memory operations between CPUs
  - E.G., SPARC TSO, Sequential Consistency (SC)
  - A processor is always Self-Consistent
  - Weaker or relaxed model → admits more reorderings but allows faster implementation (?)
  - Control with barriers or fence instructions

(Re)ordering

- Both HW and SW can reorder …
- Source-code order as described by programmer
- Instruction order – compiler reorderings (SW)
  - AKA program order
- Execution order – OoO execution (HW)
- Storage order – store buffers, bus etc (HW)
  - AKA: visibility order, memory order
  - Hardware reordering is **typically** a processor-local phenomena

Reorderings

- Reordering ➔ Program order and visibility (memory) order differ
- How do reordering arise?
  - Compiler/JIT reorders code – volatile
  - Platform reorders operations – fences
    - CPU-local
      - OoO execution or speculation
      - Store buffer (!!!)
    - Inter-CPU: bus
- Trade speed (maybe) for software complexity
  - Dekker Algorithms
  - Double-checked locking

Spectrum of HW Memory Models

- Sequentially Consistent (SC)
  - "Strongest", no HW reorderings allowed
  - IBM z-series, 370, etc
  - All UP are SC. No need for barriers!
- SPARC TSO: ST A; LD B
  - Store can languish in store buffer – invisible to other CPUs – while LD executes. LD appears on bus before ST
  - ST appears to bypass LD
  - Prevent with MEMBAR #storeload
- IA32+AMD64: PO and SPO
  - Fairly close to SPARC TSO
  - Prevent with fence or any serializing instruction
- Weak memory models
  - SPARC RMO, PPC, IA64

Memory Model

- Defines how memory read and writes may be performed by a process relative to their program order
  - IE, Defines how memory order may differ from program order
- Define how writes by one processor may become visible to reads by other processors

Coherence costs: Communication

- Communicating between CPUs via shared memory
- Cache coherency protocols
  - MESI/MOESI snoop protocol
    - Only one CPU can have line in M-State (dirty)
    - Think RW locks – only one writer allowed at any one itme
    - All CPUs snoop the shared bus - broadcast
    - Scalability limit ~~ 16-24 CPUs
    - Most common protocol
  - Directory
    - Allows Point-to-point
    - Each $ line has a "home node"
    - Each $ line has pointer to node with live copy
  - Hybrid snoop + directory

Coherence costs: Niagara-1

- L2$ is coherency fabric – on-chip
- Coherence Miss latency ≈ L2$ miss
- Coherency latency is short and bandwidth plentiful
- Sharing & communication are low-cost
- 32x threads RMW to single word – 4-5x slowdown compared to 1x
  - Not bad considering 4 strands/core!

Coherence costs: 96X StarCat

- Classic SMP
- Coherency interconnect – fixed bandwidth (bus contention)
- Snoop on 4x board – directory inter-board
- Sharing is expensive
- Inter-CPU ST-to-LD latency is +100s of cycles (visibility latency)
- 96x threads RMW or W to single word – >1000x slowdown
- 88MHz interconnect and memory bus
- Concurrent access to $Line:
  - Mixed RW or W is slow
  - Only RO is fast
- False sharing: accidentally collocated on same $line

SMP Coordination Primitives

- ST+MEMBAR+LD
  - Mutual exclusion with Dekker or Peterson
  - Dekker duality:
    - Thread1: STA; MEMBAR; LDB
    - Thread2: STB; MEMBAR; LDA
  - Needs MEMBAR(FENCE) for relaxed memory model
    - Serializing instruction on IA32
    - SPARC TSO allows ST;LD visibility to reorder
    - Drain CPU store buffer
    - Atomics are typically barrier-equivalent
  - Word-tearing *may* be barrier-equivalent STB;LDW (*bad)*
- Atomics
  - Usually implemented via cache coherency protocol

Atomic Zoo - RMW

- Fetch-And-Add, Test-and-set, xchg, CAS
- IA32 LOCK prefix – LOCK: add [m],r
- LL-SC (ppc,Power,mips) – one word txn
- Can emulate CAS with LL-SC but not (easily) vice-versa
- Bounded HW Transactional Memory
  - TMn (TM1 == LL-SC)
- Unbounded HTM – research topic (google TCC)
- Hybrid HW/SW TM – HW failover to SW
- CASn - many distinct locations -- CASn ≈ TMn
- CASX – wide form – adjacent (cmpxchg8b, 16b)
- CASv – verifies add'l operand(s)
- A-B-A problem:
  - LL-SC, TM immune
  - Simple CAS vulnerable

Concept: Lock-Freedom

- Locks – pessimistic
  - Prevent races
  - If owner stalls other threads impeded
  - Analogy: token-ring
- Lock-free - optimistic
  - LD … CAS – single-word transaction – j.u.random
  - Detect and recover (retry) from potential interference
  - Analogy: CSMA-CD + progress
  - Analogy: HotSpot source code putback process
  - Improved progress
  - Increased use in 1.6 native C++ code
  - Progress guarantees for aggregate system, but not individual threads

Concept: Lock-Freedom (2)

- Excellent throughput under contention
- Immune to deadlock
- No convoys, priority inversion …
- Somebody always makes progress
  (CAS fails ➔ somebody else made progress)
- Composable abstraction -- more so than locks
- Shorter txn more likely to complete -- minimize LD…CAS distance
- Always better than locking? No – consider contrived case where failed LF attempt is more expensive than a context-switch
- But harder to get right – Locks *conceptually* easier
- RT needs wait-free -- LF lets a given thread fail repeatedly

Locking Tension

- lock coverage – performance vs correctess
  - Too little - undersynchronized ➔ admit races, bugs
  - Too much - oversynchronized ➔ Serialize execution and reduced ||ism
- Data Parallelism vs Lock Parallelism
- Example: Hashtable locks – *Lock break-up*
  - Lock Table: coarse-grain
  - Lock Bucket: compromise
  - Lock Entry: fine-grain but lots of lock-unlock ops
- Complexity
  - Tricky locking & sync schemes admit more parallelism
  - but difficult to get correct
  - Requires domain expertise
  - More atomics increase latency

Coarse|Fine-Grain parallelism

- Coarse-Grain vs fine-grain parallelism
- Balance communication costs vs parallel utilization
- Potential parallelism dictated by the problem (matmul)
- Not all problems decompose conveniently
- Sometimes the designer gets to decide:
  - consider GC work-strealing and the "size" of work units
- Fine-grain
  - Higher communication costs to coordinate threads
  - Maximize CPU utilization - ||ism
  - Reduced sensitivity to scheduling
- Coarse-grain
  - Reduced communication costs
  - Extreme case: specjbb – embarrassingly ||, little inter-thread communication. Contrived.

Java Synchronization

- Thread Models
- Requirements
- Various implementations
- Terminology
- Performance
  - Uncontended
  - Contended

Thread Model in 1.6

- Kernel controls [Ready ⇐⇒ Run] transitions
- JVM moderates [Run ➔ Blocked] and [Blocked ➔ Ready] transitions (Except for blocking page faults or *implicit blocking*)
- JVM manages explicit per-monitor EntryList and WaitSet lists
- 1:1:1 model (Java:Libthread:LWP)
- Alternate thread models
  - Green threads – M:1:1
    - Often better but lack true ||ism
    - No CAS needed for pure Green M:1:1 model
    - Control over scheduling decisions + cheap context switch
    - Must provide preemption
  - IBM Jikes/RVM experience – M:N:N
  - Raw hypervisors – M:P:P
  - JNI problems – mixing internal model with real world
  - Don't recreate libthread T1 !

HotSpot Thread Scheduling

- Thread abstraction layering …
- JVM maps java threads to user-level threads (1:1)
- Libthread maps user-level threads to kernel-LWPs (1:1)
- Kernel maps LWPs to logical CPUs (strands)
- Silicon maps logical CPUs onto cores (CMT)

Java Synchronization

- Threads - explicitly describe parallelism
- Monitors
  - Explicitly proscribe parallelism
  - Conceptually: condvar + mutex + aux fields
  - *Typically* implemented via mutual exclusion
  - Source code: Synchronized (lock-unlock), wait(), notify()
  - Bytecode: ACC_SYNCHRONIZED, monitorenter, monitorexit
- Better than C/pthreads, but still nightmarish user bugs
- java.util.concurrent operators appeared 1.5.0 (j.u.c)
  - JSR166 based on Doug Lea's package
  - Many impls lock-free

Typical Java Sync Usage

- **Generalities** -- Identify common cases we need to optimize
- Critical sections are usually short
- Locking is frequent in time but very localized in space
- Locking operations happen frequently – 1:30 lock ops per heap references in specjbb2k
- Few objects are ever locked during their lifetime
- Fewer yet are shared
  - Shared → ever locked by more than 1 thread
  - Most locks are monogamous
  - Inter-thread sharing and locking is infrequent
- Even fewer yet ever see contention
  - But more ||ism → more contention and sharing
  - When contention occurs it persists – modality
  - Usually restricted to just a few hot objects

Typical Java Sync Usage (2)

- Locking is almost always provably balanced
- Nested locking of same object is rare in app code but happens in libraries
- It's rare for any thread to every hold >= 2 distinct locks
- hashCode assignment & access more frequent than synchronization
- Hashed ∩ synchronized-on is tiny
- Behavior highly correlated with obj type and allocation site

Typical Java Sync Usage (3)

- Synchronized java code
  - Oversynchronized ?
  - Library code is extremely conservative
    - "Easy" fix for pure reader accessors
    - Use scheme akin to Linux SeqLocks
- CAS is mostly precautionary -- wasted cycles to avoid a race that probably won't happen
- Latency impacts app performance
- ~20% of specjbb2k cycles – unproductive work

Aside – why use threads ?

- True parallelism – requires MP system
  - Threads are mechanism we use to exploit parallelism
  - T >> C doesn't make sense, except …
- 1:1 IO binding - thread-per-socket idiom
  - Artifact of flawed library IO abstraction
  - Volano is classic example – IO funnel
  - Rewrite with NIO would reduce from 100s of threads to C …
  - Which would then reduce sync costs
  - Partition 1 thread per room would result in 0 sync costs

Requirements - Correctness

- Safety: mutual exclusion – property always holds. At most one thread in critical section at any given time.
- Progress: liveness – property eventually holds (avoid stranding)
- JLS/JVM required exceptions – ownership checks
- JSR133 Memory Model requirements

JSR133 – Java Memory Model

- JSR133 integrated into JLS 3E, CH 17 – clarification
- Volatile, synchronized, final
- Key clarification: Write-unlock-to-lock-read visibility for *same* lock
- Describes relaxed form of lazy release consistency
- Admits important optimizations: 1-0 locking, Biased Locking, TM
- Beware of HotSpot's volatile membar elision
  - Volatile accesses that abut enter or exit
- See Doug Lea's "cookbook"

Balanced Monitor Operations

- Balance: unlock always specifies most-recently locked object
- Synchronized methods always balanced - by defn
- JVM spec doesn't require monitorenter/monitorexit balance
- Verifier doesn't check balance -
  {aload; monitoreter; return } verifiable
- Javac:
  - Always emits balanced enter-exit bytecodes
  - Covers enter-exit pairs with try-finally blocks – cleanup – prevent escape without exit
- jni_monitorenter-exit – mixed-mode operations proscribed, but not checked

Balance (2)

- C1/C2 prove balance for a method
  - Balanced -> enable stack-locks
  - !Balanced -> forever banished to interpreter
- HotSpot Interpreter
  - Extensible per-frame list of held locks
  - Lock: add to current frame's list
  - Unlock list at return-time
  - Throw at monitorexit if object not in frame's list
- IBM, EVM, HotSpot very different
  - latitude in spec
  - relaxed in 2.0

Performance Requirements (1)

- Make synchronization fast at run-time
  - Uncontended → Latency
    Dominated by Atomics and MEMBARs
  - Contended → Scalability and throughput
    - spin to avoid context switching
    - Improve or preserve affinity
- Space-efficient
  - header words & heap consumption
  - # extant objectmonitors
- Get the space-time tradeoffs right
- Citizenship – share the system
- Predictable – consistent
  - Don't violate the principle of least astonishment

Performance Requirements (2)

- Throughput
  - Maximize aggregate useful work completed in unit time
  - Maximize overall IPC (I = Useful instructions)
- How, specifically?
  - Minimize context-switch rate
  - Maximize affinity – reduce memory and coherency traffic

Fairness vs Throughput Balance

- HotSpot favors **Throughput**
- Succession policy – who gets the lock next
  - Competitive vs handoff
  - Queue discipline: prepend / append / mixed
  - Favor recently-run threads
- Maximize aggregate tput by minimizing $ and TLB misses and coherency traffic
- Lock metadata and data stay resident on owner's CPU
- Threads can dominate a lock for long periods
- Trade-off: individual thread latency vs overall system tput
- Remember: fairness is defined over some interval!
- Highly skewed distribution of lock-acquire times
- Good for commercial JVM, bad for RTJ
- Use JSR166 j.u.c primitives if fairness required
- It's easier to implement fair in terms of fast than vice-versa

Non-Requirements

- Implementation latitude
  - Not legislated
  - Simply a matter of implementation quality
- Yield() - strictly advisory
  - Green threads relic – non-preemptive thread model
  - Expect to see it made into a no-op
- Thread priorities – strictly advisory
  - See the 1.5 web page for the sad history
- Fairness and succession order

Use Naïve Monitor Construct ?

- Somebody always asks why we don't …
- Map Java monitor 1:1 to a native condvar-mutex pair with Owner, recursion counter, fields
- Yes, it works but …
- JVM contended & uncontended performance ***better*** than native "C" Solaris T2, Linux NPTL
- Space: always inflated on 1st lock operation
- Space: lifecycle – use finalizer to disassociate
- Time: can't inline fast path
- Time: cross PLT to call native library
- At mercy of the native implementation
- Thread.interrupt() requirements (signals!)

Safety and Progress

- CAS in monitorenter provides for exclusion (safety)
- CAS or MEMBAR in monitorexit provides for progress (liveness)
  - Exiting thread must wake successor, if any
  - Avoid missed wakeups
  - CAS closes race between fast-path exit and slow-path contended enter
  - C.F. unbounded spin locks – ST instead of CAS

Taxonomy: 1-1, 1-0, 0-0

- Describes latency of uncontended enter-exit path
- 1-1: enter-exit pair requires CAS+CAS or CAS+MEMBAR
- 1-0: enter-exit pair requires CAS-LDST
- 0-0: enter-exit pair requires 0 atomics
  - AKA: QRL, Biased Locking, 0-0 Locking
- Consider MEMBAR or FENCE ≈ Atomic because of similar latency

Terminology – Fast/Slow path

- Front-end (Fast path)
  - uncontended operations for biased, inflated & stack-locked
  - usually emitted inline into method by JIT Compiler
  - Fast-path triages operation, calls slow-path if needed
  - Also in interpreter(s) – but no performance benefit
  - Degenerate form of slow-path code (hot cases)
- Back-end (Slow-path)
  - Contention present – block threads or make threads ready
  - JNI sync operations
  - Calling back-end usually results in inflation
  - Wait() notify()

Object states in HotSpot 1.6

- Mark:
  - Word in object header – 1 of 2
  - Low-order bits encode sync state
  - Multiplexed: hashCode, sync, GC-Age
- Identity hashCode assigned on-demand
- State determines locking flavor
- Mark encoding defined in markOop.hpp
- HotSpot uses pointer-based encoding
  - Tagged pointers
  - IBM & BEA use value-based thin-locks

Mark Word Encodings (1)

- Neutral: Unlocked
- Stack-Locked: Locked – **displaced** mark
- Biased: Locked & Unlocked
- Inflated: Locked & Unlocked – **displaced** mark
- Inflation-in-progress – transient 0 value

Mark Word Encodings (2)

- Neutral: Unlocked
- Biased:
  - Locked or Unlocked and unshared
    - Monogamous lock
    - Very common in real applications
  - Optional optimization
  - Avoid CAS latency in common case
  - Tantamount to deferring unlock until contention
  - 2nd thread must then **revoke** bias from 1st

Mark Word Encodings (3)

- Stack-locked
  - Locked + shared but uncontended
  - Mark points to **displaced** header value on Owner's stack (BasicLock)
- Inflated
  - Locked|Unlocked + shared + contended
  - Threads are blocked in monitorenter or wait()
  - Mark points to heavy-weight objectMonitor native structure
  - Conceptually, objectMonitor extends object with extra metadata sync fields
  - Original header value is displaced into objectMonitor
  - Inflate on-demand – associate object with unique objectmonitor
  - Deflate by scavenging at stop-the-world time
  - At any given time …
  - An object can be assoc with at most one objectMonitor
  - An objectMonitor can be assoc with at most one object

Terminology - Inflation

- Inflated -> markword points to objectMonitor
- objectMonitor = optional extension to object
  - Native structure (today)
  - Owner, WaitSet, EntryList, recursion count, displaced markword
  - Normally not needed – most objs never inflated
- Inflate on demand -- wait(), contention, etc
- Object associated with at most one monitor at any one time
- Monitor associated with at most one object at any one time
- Deflate at STW-time with scavenger
- Could instead deflate at unlock-time
  - Awkward: hashCode & sync code depends on mark ⇔ objectmonitor stability between safepoints
- Issue: # objectMonitors in circulation
  - Space and scavenge-time

HotSpot 2-tier locking scheme

- Two-tier: stack-locks and inflated
- Stack-locks for simple uncontended case
  - Inlined fast-path is 1-1
- Revert to inflated on contention, wait(), JNI
  - Inlined inflated fast-path is 1-0

HotSpot Stack-Locks (1)

- Uncontended operations only
- JIT maps monitorenter-exit instruction BCI in method to BasicLock offset on frame
- Terms: Box, BasicLock → location in frame
- Lock()
  - Neutral → Stack-locked
  - Save mark into on-frame box, CAS &box into mark
- Unlock()
  - stack-locked → neutral
  - Validate, then CAS displaced header in box over &box in mark
- Lock-unlock inlined by JIT : 1-1

Stack-Locks (2)

- When locked:
  - Mark points to BasicLock in stack frame
  - Displaced header word value saved in box
  - Mark value provides access to displaced header word
  - Mark value establishes identity of owner
- Unique to HotSpot – other JVMs use thin-lock or always inflate
- Requires provably balanced locking

Monitor Implementations

- EVM – MetaLock (1-1) and Relaxed Locks
  - inflate at lock, deflate at unlock
  - No scavenging or STW assist required
  - Singular representation of "LOCKED"
  - Relaxed Locks used in Mono (CLR)
- IBM – Bacon/Tasuki (1-1 originally, now 1-0
  - Thin Encoding = (ThreadID,recursions,tag)
  - Revert to inflation as needed
  - Usually requires extra header word
  - Distinct hashCode & sync words
  - Now used by BEA, too

Digression – Source Map for 1.6

- Mark-word encoding
  - markOop.hpp
- Platform independent back-end
  - synchronizer.cpp
- Platform-specific park() and unpark()
  - os_<Platform>.cpp and .hpp
- C2 fast-path emission – FastLock node
  - SPARC: assembler_sparc.cpp compiler_lock_object()
  - Others: <CPU>.ad
- C1 fast-path emission
- Template interpreters fast-path
  - Revisit this decision!

More CAS economics

- CAS has high **local** latency -- 100s of clocks
- Empirical: worse on deeply OoO CPUs
- CAS accomplished in local D$ if line already in M-state.
- IA32 LOCK prefix previously drove #LOCK. No more.
- Scales on MP system – unrelated CAS/MEMBAR operations don't interfere or impede progress of other CPUs
- CAS coherency traffic no different than ST - $Probes
- Aside: Lock changes ownership → often D$ coherency miss. Cost of CAS + cost of miss
- ST-before-CAS penalty – drain store buffer
- CAS and MEMBAR are typically **CPU-Local** operations
- $line upgrade penalty: LD then ST or CAS
  - 2 bus txns : RTS then RTO
  - PrefetchW? Fujitsu!

Uncontended Performance

- Reduce sync costs for apps that aren't actually synchronizing (communicating) between threads
- eliminate CAS when there's no real communication
- Compile-time optimizations – Pessimistic/conservative
  - Lock coarsening – hoist locks in loops
    - Tom in 1.6
  - Lock elision via escape analysis
    - Seems to have restricted benefit
- Run-time optimizations – optimistic/aggressive
  - 0-0 or 1-0 locks – Detect and recover

Eliminate CAS from Exit (1-0)

- Eliminate CAS or MEMBAR from fast-exit
  - Admits race - vulnerable to progress failure.
  - Thread in fast-exit can race thread in slow-enter
  - Classic missed wakeup – thread in slow-enter becomes *stranded*
- Not fatal – can detect and recover with timed waits or helper thread
- 1.6 inflated path is 1-0 with timers
- Trade MEMBAR/CAS in hot fast-exit for infrequent but expensive timer-based recovery action in slow-enter
- Low traffic locks ➔ odds of race low
- High traffic locks ➔ next unlock will detect standee
- Optimization: Only need one thread on EntryList on timed wait
- Downside:
  - Worst-case stranding time – bounded
  - Platform timer scalability
  - Context switching to poll for stranding
- IBM and BEA both have 1-0 fast-path

Biased Locking

- Eliminate CAS from enter under certain circumstances
- Morally equivalent to leaving object locked after initial acquisition
- Subsequent lock/unlock is no-op for original thread
- Single bias holding thread has preferential access – lock/unlock without CAS
- Defer unlock until 1st occurrence of contention - Revoke
- Optimistic -- Assume object is never shared. If so, detect and recover at run-time
- Bias encoding can't coexist with hashCode
- Also know as: 0-0, QRL

Biased Locking (2)

- In the common case
  …T locks object O frequently
  .. And no other thread ever locked O
  … Then *Bias* O toward T
- At most one *bias-holding-thread* (BHT) T for an object O at any one time
- T can then lock and unlock O without atomics
- BHT has preferential access
- S attempts to lock O → need to **revoke** bias from T
- Then revert to CAS based locking (or rebias to S)

Biased Locking (3)

- Improves single-thread **latency** – eliminated CAS
- Doesn't improve scalability
- Assumes lock is typically dominated by one thread
  - Bets against true sharing – bet on single accessor thread
- Revoke may be expensive (STW or STT)
- Trade CAS/MEMBAR in fast-path enter exit for expensive revocation operation in slow-enter
  - Shifted expense to less-frequently used path
- Profitability – 0-0 benefit vs revoke cost – costs & frequencies

Biased Locking (4)

- Revoke Policy: rebias or revert to CAS-based locking
- Policy: When to bias :
  - Object allocation time – toward allocating thread
  - 1st Lock operation
  - 1st Unlock operation -- better
  - Later – after N lock-unlock pairs
- Represent BIASED state by encoding aligned thread pointer in mark word + 3 low-order bits
- Can't encoded hashed + BIASED

Biased Locking (5)

- Revocation storms
- Disable or make ineligible to avoid revocation: globally, by-thread, by-type, by-allocation-site
- Ken's variant:
  - Stop-the-world, sweep the heap, unbias by type,
  - Optionally adjust allocation sites of offending type to subseq create ineligible objects,
  - No STs at lock-unlock. "Lockness" derived from stack-walk
- Detlefs: bulk revocation by changing class-specific "epoch" value at STW-time (avoids sweep)

Biased Locking (6)

- Compared to sync elision via escape analysis
  - optimistic instead of conservative – better eligibility
  - Run-time instead of compile-time
  - Detect and recover instead of prevent or prove impossible
- Less useful on bigger SMP systems
  - Simplistic model: revoke rate proportional to #CPUs
  - Parallelism ➔ Sharing ➔ revocation
  - Real MT apps communicate between threads & share
  - Good for SPECjbb'

Biased Locking - Revocation

- Revoker and Revokee must coordinate access to mark
- Unilateral exclusion: revoker acts on BHT
  - Stop-the-thread, **stop-the-world safepoints**
    - **Scalability impediment – loss of parallelism**
  - Signals
  - Suspension
- Cooperative – based on Dekker-like mechanism
  - MEMBAR if sufficiently cheap
  - Passive: Delay for maximum ST latency
  - External serialization – revoker serializes BHT
    - X-call, signals
    - Scheduling displacement – force/wait OFFPROC
    - Mprotect() – TLB access is sequentially consistent

Biased Locking - Improvements

- Cost model to drive policies
- Reduce revocation cost
  - Avoid current STW (STT?)
- Reduce revocation rate
  - Limit # of objects biased
  - Feedback: make object ineligible
    - By Type, By Thread, By allocation site
  - Bulk
  - Batching
  - Rebias policies

Recap – fast-path forms

- 0-0 locks
  + fast uncontended operations
  - revocation costs
- 1-0 locks
  + no revocation – predictable and scalable
  + all objects always eligible
  - theoretical stranding window - bounded
  - still have one CAS in fast-path
  + no heuristics

Contended Locking – new in 1.6

- Inflated fast-path now 1-0
- Based on park-unpark – platform-specific primitives
- Reduce context switching rates – futile wakeups
- In exit: if a previous exit woke a successor (made it ready) but the successor hasn't yet run, don't bother waking another
- Notify() moves thread from WaitSet to EntryList
- Lock-free EntryList operations – List operations shouldn't become bottleneck or serialization point
- Adaptive spinning

Contended Locking (2)

- Classic Lock implementation
  - Inner lock = protects thread lists – monitor metadata
  - Outer lock = mutex itself – critical section data
  - double contention! – Outer + Inner
- We use partially Lock-free queues
  - paradox - impl blocking sync with lock-free operations
  - Avoids double contention
- Minimize effective critical section length
- Amdahl : don't artificially extend CS
- Wake >> unlock to avoid lock jams
- Don't wake thread at exit time if successor exists
- Two queues: contention queue drains into EntryList

Park-Unpark primitives

- Per-thread restricted-range semaphore
- Scenario:
  - T1 calls park() – blocks itself
  - T2 calls unpark(T1) – T1 made ready - resumes
- Scenario:
  - T2 calls unpark(T1)
  - T1 calls park() – returns immediately
- "Event" structure in type-stable memory - Immortal
- Implies explicit threads lists – queues
- Simple usage model: park-unpark can both be implemented as no-ops. Upper level code simply degenerates to spinning

Spinning

- Spinning is stupid and wasteful …
- unless the thread can find and perform useful alternate work (marking?)
- Spinning = caching in the time domain
  - Stay on the CPU …
  - So the TLBs and D$ stay warm …
  - So future execution will be faster
  - Trade time for space for time

Adaptive Spinning (1)

- Goal – avoid context switch
- Context switch penalty
  - High latency - 10k-80k cycles
  - Cache Reload Transient – D$, TLBs – amortized penalty
  - CRT hits immediately after lock acquisition – effectively lengthens critical section. Amdahl's law ➔ impedes scalability
- If context switch were cheap/free we'd never spin : Green threads
- Thought experiment – system saturation
  - Always spin : better tput under light load
  - Always block: better tput under heavy load
- Spin limit ~ context switch round trip time

Adaptive Spinning (2)

- Spin for duration D – Vary D based on recent success/failure rate for that monitor
- Spin when spinning is profitable – rational cost model
- Adapts to app modality, parallelism, system load, critical section length
- OK to bet if you know the odds
- Damps load transients – prevents phase transition to all-blocking mode
  - Blocking state is sticky. Once system starts blocking it continues to block until contention abates
  - Helps overcome short periods of high intensity contention – Surge capacity

Adaptive Spinning – Classic SMP

- CPU cycles are a local resource – coherency shared and scarce
- Context switch
  - disrupts local $ and TLBs
  - usually local latency penalty
- Spinning
  - Waste local CPU cycles
  - Bet we can avoid context switch
  - Back-off to conserve global coherency bus bandwidth
- Aggressive spinning - impolite
  - Minimizes Lock-Transfer-Time
  - Increases coherency bus traffic
  - Can actually increase release latency as unlocking thread must contend to get line into M-State
- Polite: exponential back-off to reduce coherency/mem bus traffic

Adaptive Spinning - Niagara

- Niagara or HT Xeon - Inverted
- CPU cycles shared – coherency plentiful
- Context switch disruptive to other strands
- Spinning impedes other running strands on same core!
- Spinning:
  - Try to conserve CPU cycles
  - Short or no back-off
- Use WRASI or MEMBAR to stall
- Other CPUs – Polite spinning
  - Intel HT – PAUSE
  - SSE3 - MWAIT
  - Power5 – drop strand priority while spinning

Adaptive Spinning (5)

- Find something useful to do – beware polluting D$ + TLBs
  - Morally equivalent to a context switch – Cache Reload Transient
- Don't spin on uniprocessors
- Back-off for high-order SMP systems
  - Avoid coherency bus traffic from constant polling
- Cancel spin if owner is not in_java (Excludes JNI)
- Cancel spin if pending safepoint
- Avoid transitive spinning

Contended Locking – Futures (1)

- Critical section path length reduction (JIT optimization?)
  - Amdahl's parallel speedup law
  - Move code outside CS
  - Improves HTM feasibility
- Use priority for queue ordering
- Spinner: power-state on windows
- Spinner: loadavg()
- Spinner: cancel spin if owner is OFFPROC
  - Schedctl

Contended Locking – Futures (2)

- Minimize Lock Migration
  - Wakeup locality – succession
    - Pick a recently-run that thread shares affinity with the exiting thread
    - JVM Need to understand topology – shared $s
    - Schedctl provides fast access to cpuid
  - Pull affinity – artificially set affy of cold wakee
    - Move the thread to data instead of vice-versa
- YieldTo() – Solaris loadable kernel driver
  - Contender donates quantum to Owner
  - Owner READY (preempted) ➔ helps
  - Owner BLOCKED (pagefault) ➔ useless

Contended Locking – Futures (3)

- Set schedctl nopreempt, poll preemptpending
  - Use as quantum expiry warning
  - Reduce context switch rate – avoid preemption during spinning followed by park()
- Polite spinning: MWAIT-MONITOR, WRASI, Fujitsu OPL SLEEP, etc
- Just plain strange:
  - Transient priority demotion of wakee (FX)
  - Deferred unpark
  - Restrict parallelism – clamp # runnable threads
    - Variation on 1:1:1 threading model
    - Gate at state transitions
      - Allow thread to run, or
      - Place thread on process standby list
    - Helps with highly contended apps
    - Puts us in the scheduling business

Futures (1)

- HTM – juc, native, java
- PrefetchW before LD … CAS : RTS → RTO upgrades
- Native C++ sync infrastructure badly broken
- Escape analysis & lock elision
- 1st class j.u.c support for spinning
- Interim step: convert 1-1 stacklocks to 1-0
  - Eliminate CAS from exit – replace with ST
  - Admits orphan monitors and hashCode hazards
  - Fix: Idempotent hashCode
  - Fix: Timers – detect and recover
  - Works but is **ugly**

Futures (2)

- Convert front-end to Java ?
  - BEA 1.5
  - j.l.O.Enter(obj, Self)
  - Use unsafe operators to operate on mark
  - CAS, MEMBAR, etc
  - Park-Unpark
  - ObjectMonitor becomes 1st-class Java object - Collected normally

Synchronization TOI

Dave Dice – Java Core Technology

2005.05.09

Sun microsystems

Memory Model [OPTIONAL]

- Defines how memory read and writes may be performed by a process relative to their program order
  - IE, Defines how memory order may differ from program order
- Define how writes by one processor may become visible to reads by other processors