Synchronization in Java SE 6 (HotSpot)

Dave Dice

dice@sun.com

HotSpot JVM Core Engineering

Synchronization Performance

- Contended costs (scalability + latency)
  - Context switching is extremely expensive
  - Unbounded spinning is unacceptable
  - Address via **adaptive spinning**
- Uncontended costs (latency)
  - Atomic CAS has high **local** latency
  - 100s-1000s of cycles
  - Address via
    - **Biased Locking**
    - **Lock Coarsening**
    - Lock Elision through **Escape Analysis**

HotSpot Locking Fundamentals

- Object header - metadata
  - Mark word
  - Class pointer
  - ... followed by constituent fields
- Mark word multiplexed
  - Identity hashCode
  - GC Age bits
  - Synchronization information
  - **Displaced** mark word

Object States – Encoded in Mark Word

- Neutral: Unlocked
- Biased: Locked|Unlocked + Unshared
  - Tantamount to deferring unlock until contention
  - Avoids CAS atomic latency in common case
  - 2nd thread must **revoke** bias from 1st
- Stack-Locked: Locked + Shared but uncontended
  - Mark points to displaced header on owner's stack
- Inflated: Locked|Unlocked + Shared and contended
  - threads are blocked: enter or wait
  - Mark points to heavy-weight objectmonitor structure

Key Observations

- Most objects are never locked
- If an object is locked it is usually locked by at most one thread during its lifetime
  - > Very few objects are locked by more than one thread
- Even fewer objects encounter contention
- Object type and allocation site correlate strongly with future synchronization behavior

Biased Locking

- Leverages the observation that most objects are locked by at most one thread in their lifetime
- *Bias* object O toward Thread T1
- T1 can then preferentially lock and unlock O without expensive atomic instructions (CAS)
- If T2 attempts to lock O we *revoke* bias from T1
  - Either rebias to T2 or revert to normal locking and make O ineligible for further biased locking

Adaptive Spinning

- Spin-then-block strategy
  - Try to avoid context switch by spinning on MP systems
- Spin duration
  - Maintained per-monitor
  - varies based on recent history of spin success/failure ratio
- Adapts to system load, parallelism, application modality
- MP-polite spinning
- Avoid spinning in futile conditions (owner is blocked)

HotSpot Locking Fundamentals (2)

- Fast-path cases inlined by JIT at synchronization site
- Revert to slow-path (native C code) when we need to park or unpark thread
- Platform-specific park-unpark to block and wake threads
- Slow-path monitor code is platform-independent
- Much faster than native mutex constructs for contended & uncontended cases (T2, windows)

Detecting Contention

- IDEs, Profilers or 3rd party tools
- Mpstat on Solaris – vctx rate
- If suspected, sample process with pstack
  - > Look near top of stack for threads blocked in **monitorenter** operations
- JVMStat (jstat) counters
  - > jstat -J-Djstat.showUnsupported=true -snap <pid> | grep _sync_

Detecting Contention (2)

- Dtrace:
  - kernel “sched” provider
  - hotspot-specific probes (*Recommended!*)
- Identify hot locks and break up into finer-grained locking
- Beware: adding more threads can sometimes reduce performance – application specific
  - Particularly on Niagara
  - Amdahl’s speedup law – parallel corallary
  - Communication overhead can overwhelm parallelism benefit

New in 1.6

- No atomic/fence in common-case inline inflated exit path
- Code restructuring:
  - Platform independent monitor code calls ...
  - Platform-specific park-unpark
- Reduce futile wakeups
  - Don’t wake a thread in exit if thread woken in prior exit hasn’t yet run
- Lock-free EntryList
- Adaptive spinning

New in 1.6 (2)

- Notify() moves thread from WaitSet to EntryList
  - Previous versions actually woke notifyee
  - Notifyee would simply jam on lock held by notifier
- Fairness vs throughput
  - Optimized for system-wide throughput at the expensive of short-term thread-specific fairness
  - Succession policy: try to wake recently run threads
  - Improved $ and TLB utilization
- Better JSR166 (java.util.concurrent) support

New in 1.6 (3)

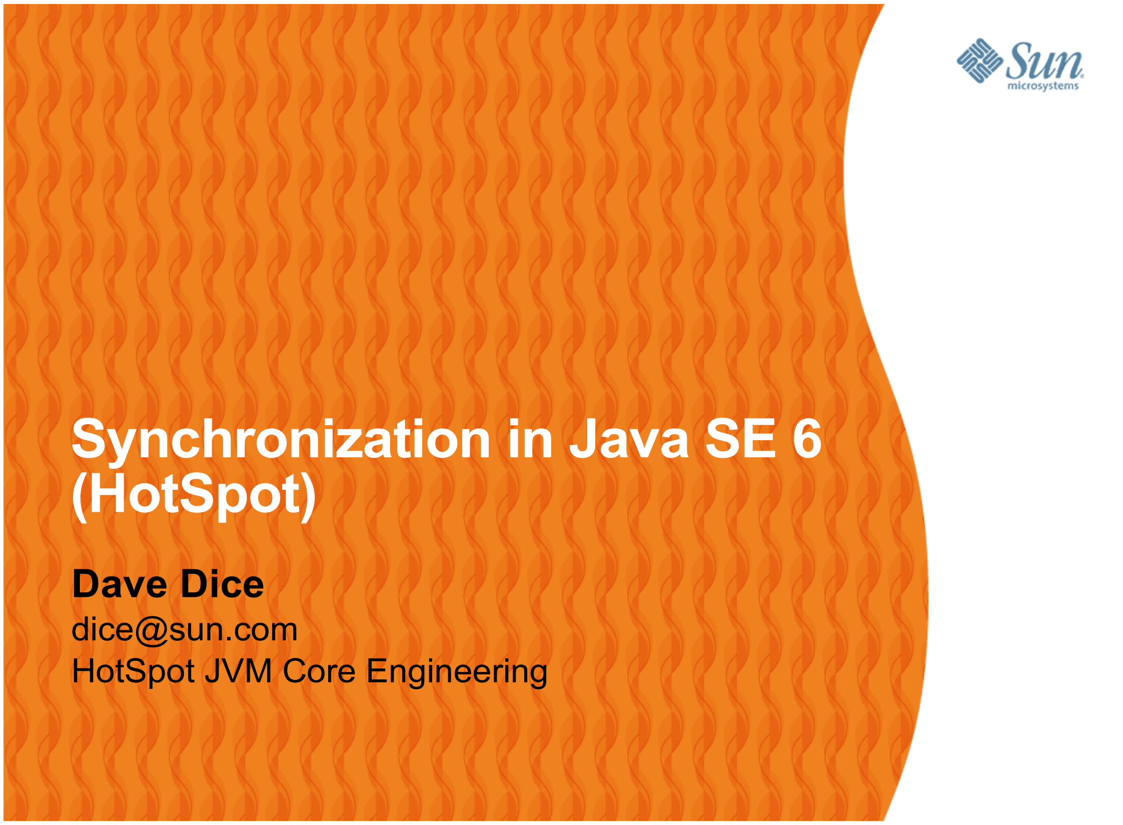
- Small changes to comply with JSR133
  - Java Memory Model (JMM)
  - JLS 3e, Chapter 17
  - -XX:-UseBiasedLocking
- Biased Locking on by default
- Lock Coarsening on by default
  - -XX:-EliminateLocks
- Lock Elision via Escape Analysis
  - -XX:+DoEscapeAnalysis

1.6 Source Roadmap

- Slow-path native
  - Platform-independent : Synchronizer.cpp
  - Platform-specific park-unpark : os_<plaf>.cpp + .hpp
- Fast-path inlined
  - Degenerate form of slow-path code
  - C2 **FastLock** node
    - assembler_sparc.cpp compiler_lock_object()
    - <cpu>.ad for other architectures
  - C1 c1_CodeStubs_<cpu>.cpp
  - Template interpreter

Additional Information

- http://blogs.sun.com/dave



Synchronization in Java SE 6 (HotSpot)

Dave Dice
dice@sun.com
HotSpot JVM Core Engineering

Synchronization Performance

- Contended costs (scalability + latency)
  - Context switching is extremely expensive
  - Unbounded spinning is unacceptable
  - Address via **adaptive spinning**
- Uncontended costs (latency)
  - Atomic CAS has high **local** latency
  - 100s-1000s of cycles
  - Address via
    - **Biased Locking**
    - **Lock Coarsening**
    - Lock Elision through **Escape Analysis**

HotSpot Locking Fundamentals

- Object header - metadata
  - Mark word
  - Class pointer
  - ... followed by constituent fields
- Mark word multiplexed
  - Identity hashCode
  - GC Age bits
  - Synchronization information
  - **Displaced** mark word

Java SE 6

Object States – Encoded in Mark Word

- Neutral: Unlocked
- Biased: Locked|Unlocked + Unshared
  - > Tantamount to deferring unlock until contention
  - > Avoids CAS atomic latency in common case
  - > 2nd thread must **revoke** bias from 1st
- Stack-Locked: Locked + Shared but uncontended
  - > Mark points to displaced header on owner's stack
- Inflated: Locked|Unlocked + Shared and contended
  - > threads are blocked: enter or wait
  - > Mark points to heavy-weight objectmonitor structure

Java SE 6

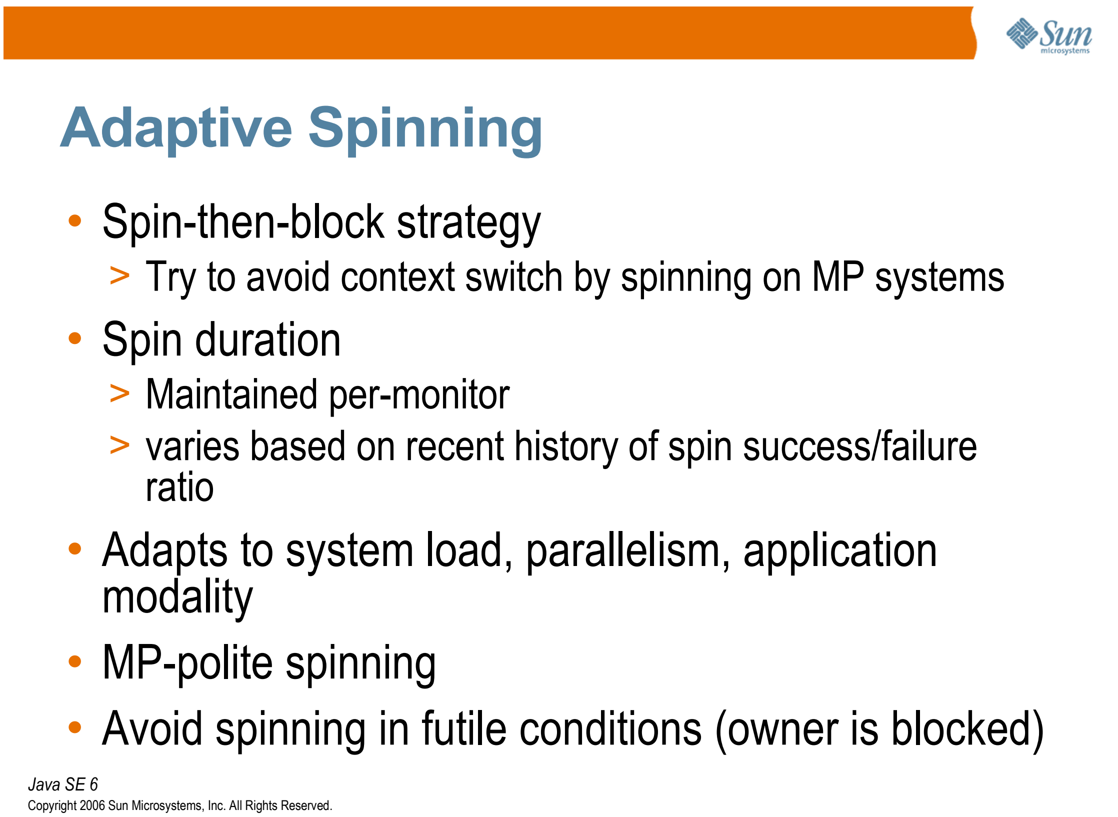
Key Observations

- Most objects are never locked
- If an object is locked it is usually locked by at most one thread during its lifetime
  - Very few objects are locked by more than one thread
- Even fewer objects encounter contention
- Object type and allocation site correlate strongly with future synchronization behavior

Biased Locking

- Leverages the observation that most objects are locked by at most one thread in their lifetime
- *Bias* object O toward Thread T1
- T1 can then preferentially lock and unlock O without expensive atomic instructions (CAS)
- If T2 attempts to lock O we *revoke* bias from T1
  - Either rebias to T2 or revert to normal locking and make O ineligible for further biased locking

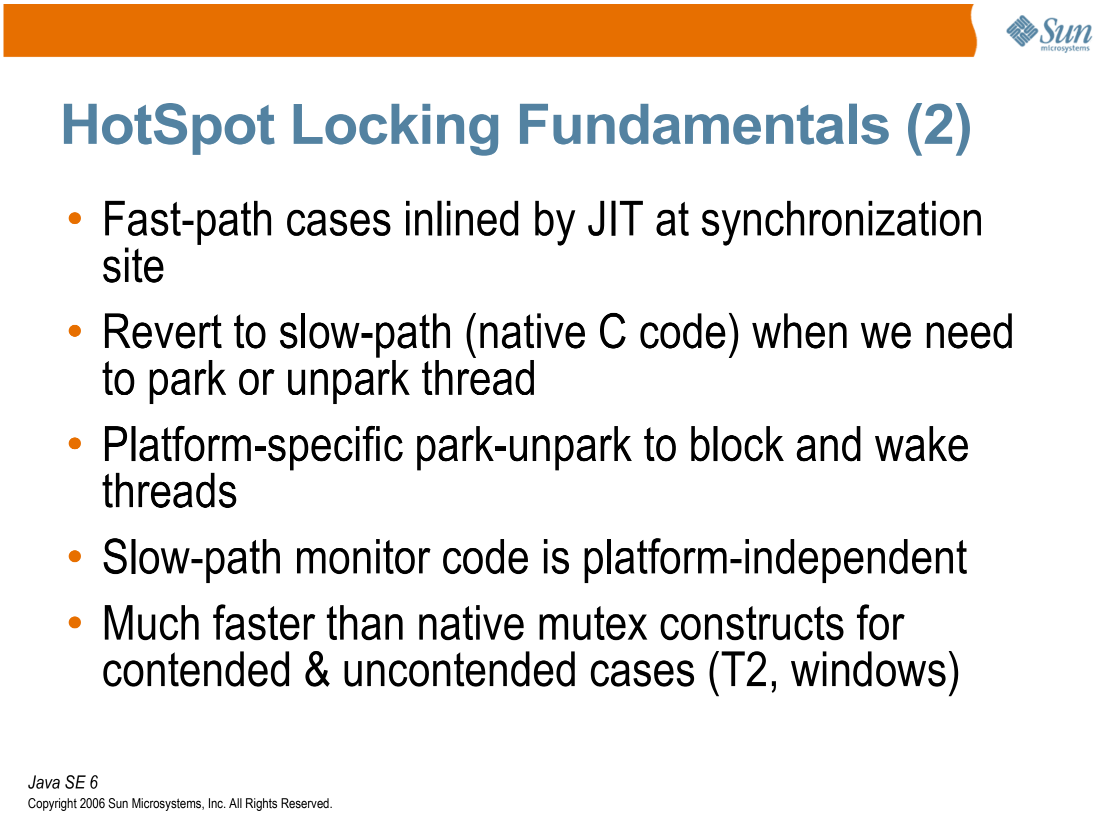
Java SE 6



Adaptive Spinning

- Spin-then-block strategy
  - > Try to avoid context switch by spinning on MP systems
- Spin duration
  - > Maintained per-monitor
  - > varies based on recent history of spin success/failure ratio
- Adapts to system load, parallelism, application modality
- MP-polite spinning
- Avoid spinning in futile conditions (owner is blocked)

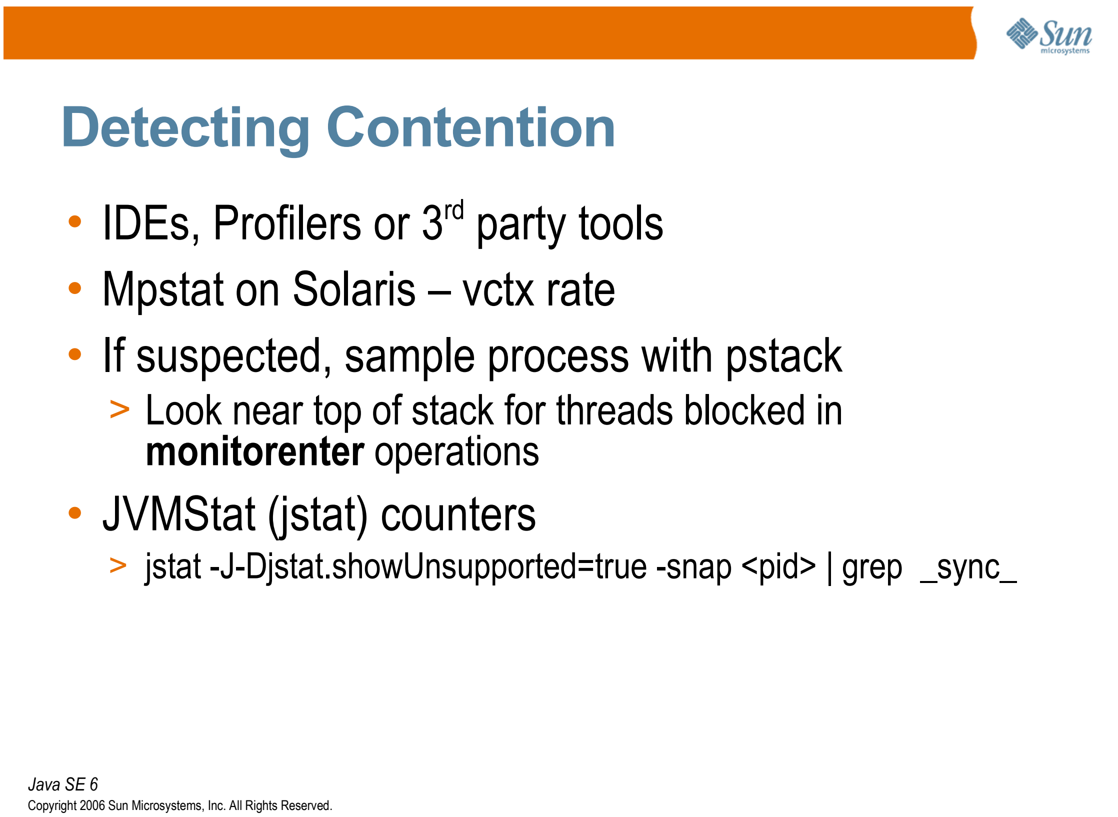
Java SE 6



HotSpot Locking Fundamentals (2)

- Fast-path cases inlined by JIT at synchronization site
- Revert to slow-path (native C code) when we need to park or unpark thread
- Platform-specific park-unpark to block and wake threads
- Slow-path monitor code is platform-independent
- Much faster than native mutex constructs for contended & uncontended cases (T2, windows)

Java SE 6



Detecting Contention

- IDEs, Profilers or 3rd party tools
- Mpstat on Solaris – vctx rate
- If suspected, sample process with pstack
  - > Look near top of stack for threads blocked in **monitorenter** operations
- JVMStat (jstat) counters
  - > jstat -J-Djstat.showUnsupported=true -snap <pid> | grep _sync_

Detecting Contention (2)

- Dtrace:
  - kernel “sched” provider
  - hotspot-specific probes (*Recommended!*)
- Identify hot locks and break up into finer-grained locking
- Beware: adding more threads can sometimes reduce performance – application specific
  - Particularly on Niagara
  - Amdahl’s speedup law – parallel corallary
  - Communication overhead can overwhelm parallelism benefit

New in 1.6

- No atomic/fence in common-case inline inflated exit path
- Code restructuring:
  - Platform independent monitor code calls ...
  - Platform-specific park-unpark
- Reduce futile wakeups
  - Don’t wake a thread in exit if thread woken in prior exit hasn’t yet run
- Lock-free EntryList
- Adaptive spinning

New in 1.6 (2)

- Notify() moves thread from WaitSet to EntryList
  - Previous versions actually woke notifyee
  - Notifyee would simply jam on lock held by notifier
- Fairness vs throughput
  - Optimized for system-wide throughput at the expensive of short-term thread-specific fairness
  - Succession policy: try to wake recently run threads
  - Improved $ and TLB utilization
- Better JSR166 (java.util.concurrent) support

New in 1.6 (3)

- Small changes to comply with JSR133
  - Java Memory Model (JMM)
  - JLS 3e, Chapter 17
  - -XX:-UseBiasedLocking
- Biased Locking on by default
- Lock Coarsening on by default
  - -XX:-EliminateLocks
- Lock Elision via Escape Analysis
  - -XX:+DoEscapeAnalysis

Java SE 6

1.6 Source Roadmap

- Slow-path native
  - > Platform-independent : Synchronizer.cpp
  - > Platform-specific park-unpark : os_<plaf>.cpp + .hpp
- Fast-path inlined
  - > Degenerate form of slow-path code
  - > C2 **FastLock** node
    - > assembler_sparc.cpp compiler_lock_object()
    - > <cpu>.ad for other architectures
  - > C1 c1_CodeStubs_<cpu>.cpp
  - > Template interpreter

Additional Information

- http://blogs.sun.com/dave

Java SE 6