Thin Locks: Featherweight Synchronization for Java

D. Bacon[1] R. Konuru[1] C. Murthy[1] M. Serrano[1]
Presented by: Calvin Hubble[2]

[1]IBM T.J. Watson Research Center

[2]Department of Computer Science
University of CA, San Diego

16th November 2005

Outline

Outline

Introduction

- Java synchronization is a double-edged word
  - Java has threads and synchronized methods
  - Synchronization is “dog slow”
- Stuck with a tradeoff
  - Bad Performance, Safe Code
  - Good performance, bug-prone code
- Can we modify Java to be faster yet still thread-safe to the everyday programmer?

Problem

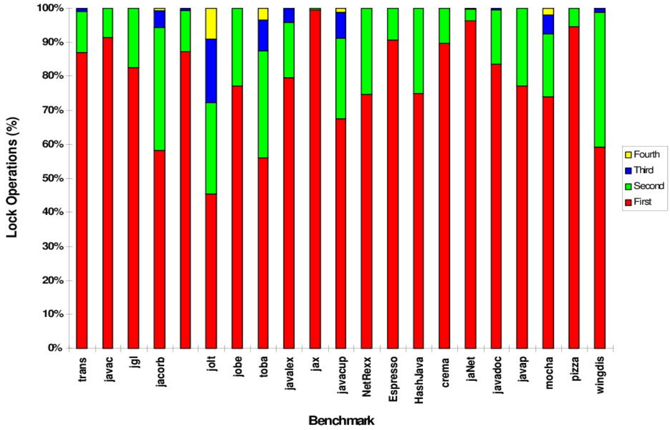
- Because Java is an explicitly multi-threaded language, general-purposes libraries are thread-safe
  - Non-trivial public methods of standard utility classes like **Vector** or **Hashtable** are synchronized
  - Example: Library call to set a bit in a bit vector:
    - 50 instructions to lock and unlock the object
    - 10 instructions method call overhead
    - 5 instructions to actually set the bit
  - Locking overhead frequently $25 - 50\%$
  - Even in single-threaded applications!!

Locking Scenarios by Frequency

1. Locking an unlocked object
2. Locking an object already locked by current thread a small number of times (shallow nested locking)
3. Locking an object already locked by the current thread many times (deeply nested locking)
4. Being the first to queue on a locked object
5. Trying to lock an object with a queue

Measurements: median of 80% of all lock operations are on unlocked objects, and nesting is very shallow.

Locking Frequency



Goal

Goal: a locking algorithm with very low overhead for single-threaded programs, but with excellent performance in the presence of multithreading and contention.

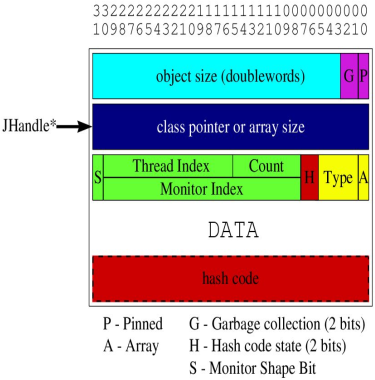
Outline

Thin Locks

- Assume pre-existing heavy-weight locking system
  - "Fat Locks"
- Thin Locks - a lightweight system for 2 most common cases
  1. Object is unlocked
  2. Shallow nested locking
- Locks are defaulted to thin and inflated if needed
- Once a lock is inflated, it can never be defaulted

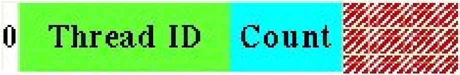
New Lock Structure

- Reserve 24 bits in the header of each object for a thin lock
  - "Obtained 24 free bits using various encoding techniques for the other values typically stored in the header"
- First bit: Monitor shape lock
  - 0 - denotes lock is "thin"
  - 1 - denotes lock is "fat"



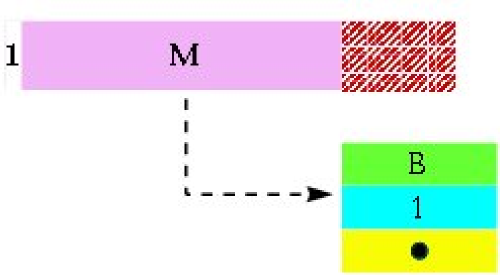
When Lock is Thin

- Lock Structure
  - Monitor Shape bit - 0
  - Next 15 bits - Thread Identifier
  - Last 8 bits - Nested lock count (+1)
- Maximum of 255 nested locks



When Lock is Fat

- Lock Structure
  - Monitor Shape bit - 1
  - Next 23 bits - index of fat lock



Assumptions

- Hardware support
  - Used **compare-and-swap**
    - CMP&SWP(addr, old, new) - If contents of addr == old value, store new value and return true, otherwise return false
- key invariant: The lock field is never modified by any thread except the current “owner”.

Locking without Contention

- Initially - lock field is 0, thread *A* wishes to lock.
- Algorithm:
  - compare-and-swap lock word
    - "Old" value: High 24-bits masked to 0
    - "New" value: monitor shape 0, thread index *A*, count 0
  - If succeeds, object was not locked by another thread and we now own lock

Unlocking without Contention (no nesting)

- Algorithm
  - Construct "old" value: monitor shape 0, thread index A, count 0
  - Read lock word and check if compares to old value, if so replace with all 0s
- Does not need compare-and-swap since no other thread can modify lock if we own it

Nested Locking and Unlocking

- Locking
  - Compare-and-swap (from before) will fail
  - If (monitor shape $== 0$) and (thread index $== A$) and (count < 255)
    - Increment count field - If count overflows then inflate lock
- Unlocking
  - Similar to above only decrement lock-count

Locking with Contention

B tries to aquire a lock held by *A*

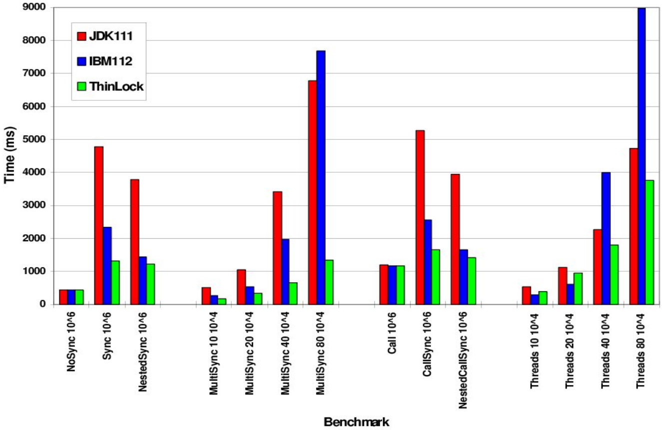
- *B*'s compare-and-swap will fail
- *B*'s check that *B* owns the lock (nested lock) will fail
- *B* needs to force a transition from thin to fat
  - *B* enters a spin-locking loop
  - Once *A* unlocks, *B* will obtain
  - *B* creates a fat lock, assigns monitor index to new monitor
  - *B* changes monitor shape bit to 1

Outline

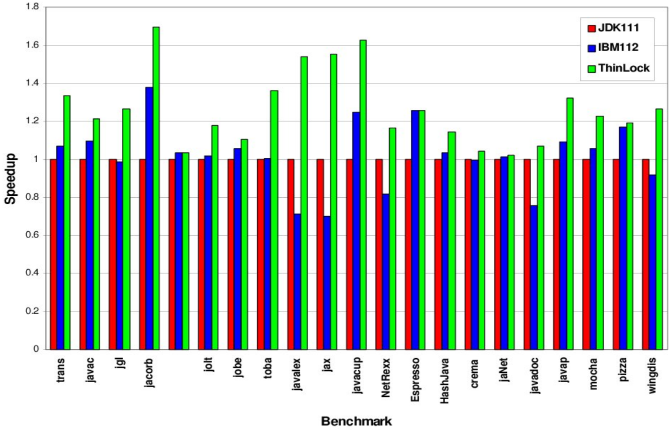
Setup

- JDK111
  - Straightfoward port of Sun's JDF 1.1.1 to AIX
- IBM112
  - IBM's 1.1.2 version of the JDK for AIX
    - Assumes that most apps have a small number of heavily used locks
    - Pre-allocates 32 "hot locks"
    - Suffers when a large number of locks are used
- ThinLock
  - Implementation of thin locks in JDK 1.1.2 for IBM's AIX OS

Micro Benchmarks



Macro Benchmarks



Conclusions

- Efficient
  - 5-10 instructions to lock/unlock object
  - no increase in object size
- Good speedups
- Portable
  - All architectures offer some locking primitive

Outline

Questions

anyone? ..... anyone? ..... Bueller?