Locking

Lecture 10
CS 390
3/18/08
Lock structure

- Typical lock structure includes:
  - lock field
  - pointer to current thread holding the lock
  - lock count
  - queue to hold waiting threads

- Locks when implemented this way can be expensive.
  - especially problematic in Java that associates a monitor with every object
  - lots of potential synchronization points that degrade performance of single-threaded programs
Common Cases

- Locking an unlocked object
- Locking an object already locked by the same thread (reentrant lock)
  - a small number of times
  - many times
- Locking an object locked by another thread
  - no other threads waiting
  - other threads waiting
Thin Locks

Assume existence of compare-and-swap

24 bits in header word used for lock

Lock field either represents a thin or fat lock

First bit is 0 if it is a thin lock

Remaining 23 bits divided as:

15 bits: thread id
8 bits: count

Thread id 0 - lock is unlocked
non-zero - index into thread table

When the object is locked, count field represents
the number of locks minus one

(a) Object layout showing lock word

(b) Lock word structure for thin lock

(c) Unlocked

(d) Locked once by thread A

(e) Locked twice by thread A
Inflated Locks

Monitor shape bit is 1

23 bits of lock field contain the index of a fat lock

Fat lock contains:
- a thread identifier for the lock owner
- the count of the number of locks queues

Invariant:
- the lock field of an object, while it is owned by a thread, is never modified by any other thread

Once a thread has locked an object, all subsequent operations can be performed with loads and stores; no need for atomic operations.

(a) Lock word structure for inflated lock

(b) Locked once by thread B

(c) Unlocked
Locking without Contention

- Initially, the object is unlocked (c)

- Thread A performs compare-and-swap on the lock field
  - old value constructed by loading the lock word and masking out the high 24 bits
  - new value is a lock field containing a monitor shape bit of 0, a thread index corresponding to A, and a count of 0.
    - constructed by bitwise OR of the old value and thread index shifted 16 bits to the left.

- If compare-and-swap succeeds, object was not locked
Unlocking without Contention

- Common case:
  - current thread owns the lock, and has locked the object once
    - old value (d), new value (c)
    - don’t need to do compare-and-swap. Why?
Nested Locking

- Suppose thread A again locks the object
- Start with a compare-and-swap
  - will fail
  - check for nested locking
    - monitor shape bit is 0
    - thread index is A’s index
    - count field is less than 256
Locking with Contention

- Thread A has object locked
- Thread B attempts to lock the object
  - compare-and-swap fails
  - it has not already locked the object, so fail again
- Need to change monitor shape bit to 1, but locking discipline requires that only owning thread can change lock state
- Thread B enters a spin-locking loop
- Once A relinquishes lock, thread B obtains a fat lock (b)
- When thread B unlocks, the lock remains in an inflated state
Results

- For microbenchmarks, think locks are 5x faster than original JDK implementation.
- For real programs, achieve median speedup of 1.22 and maximum speedup of 1.7.
- No increase in object size:
  - Fat locks only created under contention.
  - Significant space savings when there are large numbers of synchronized objects.