



# Featherweight Monitors with Bacon Bits

David F. Bacon

IBM T.J. Watson Research Center

# Contributors

---

- ◆ Chet Murthy
- ◆ Tamiya Onodera
- ◆ Mauricio Serrano
- ◆ Mark Wegman
- ◆ Rob Strom
- ◆ Kevin Stoodley

# Introduction

---

- ◆ It's the same old sad story:
  - ☺ Java has threads and synchronized methods
  - ☹ But synchronization is dog-slow
  - ☠ So synchronization is optional
- ◆ Shoot the foot of your choice:
  - ⌚ Get bad performance, or
  - 💣 Get bug-prone code

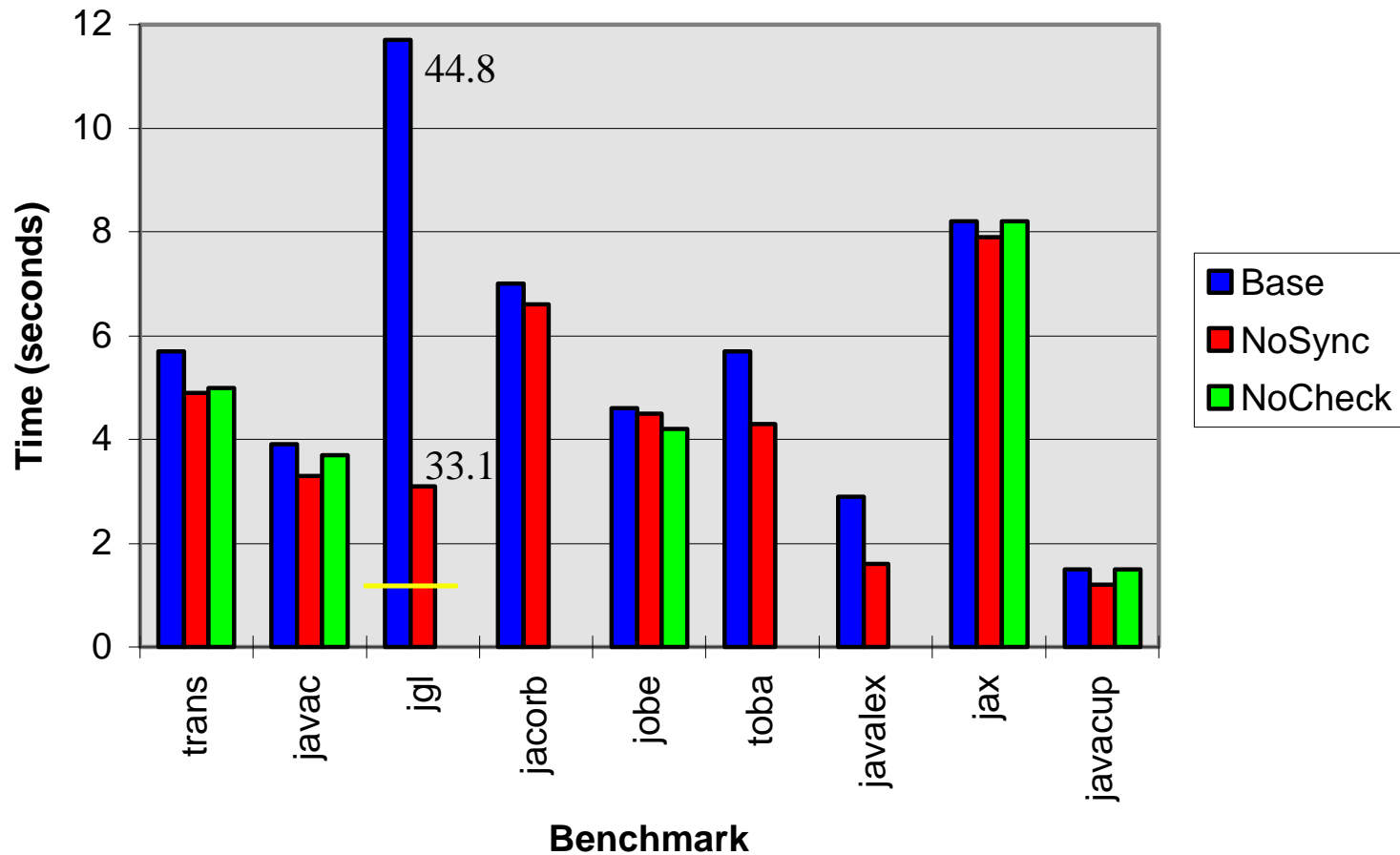
# Its's Worse Than That

---

- ◆ Libraries must be thread-safe
  - All non-trivial methods are synchronized
  - 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 above 25%
  - even in single-threaded applications!

# Java Locking Overhead

HPJ Alpha6



# Java Synchronization Features

---

- ◆ Thread can lock an object repeatedly
  - locks nest
  - nesting count must be kept
- ◆ On exception, thread must release locks
  - call stack implicitly names all locked objects
  - therefore, list of locked objects not needed

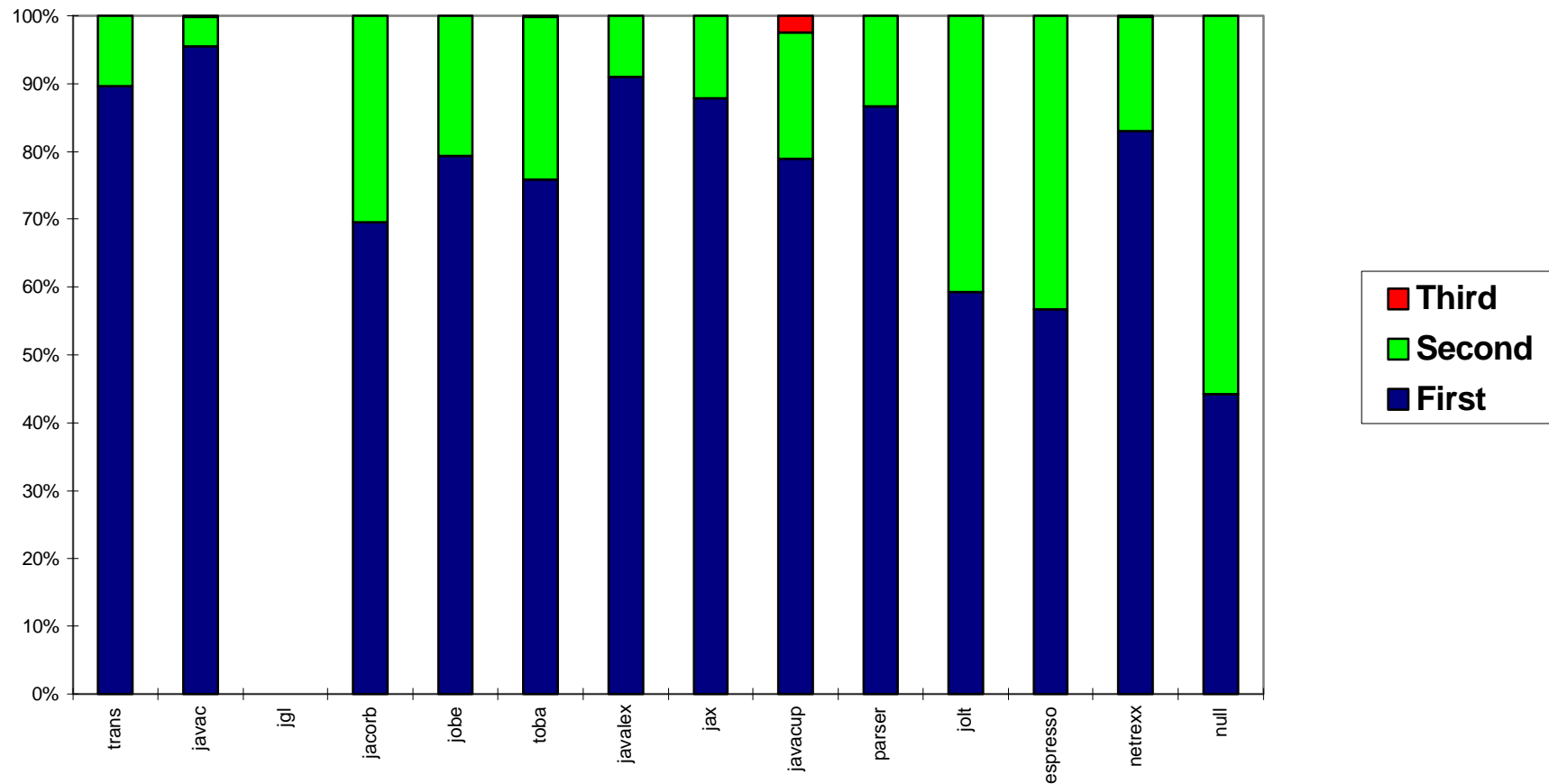
# Locking Scenarios by Frequency

---

- Object is unlocked.
  - We already locked the object a few times
    - We already locked the object a lot of times
      - Object is locked and we are the first to queue up
        - Object is locked and other threads are waiting

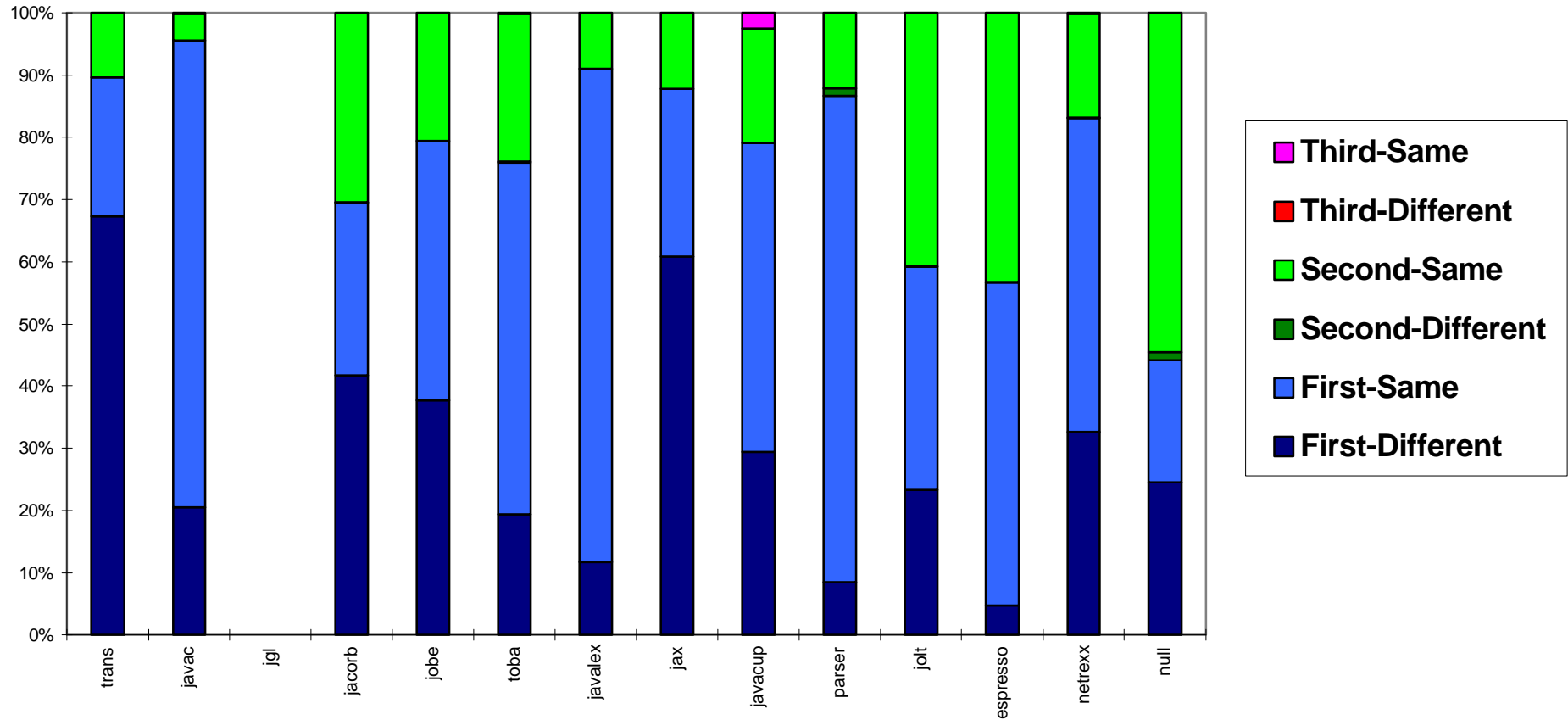
# Nested Locking Depth

---



# Repeated Locking (by depth)

---



# Assumptions

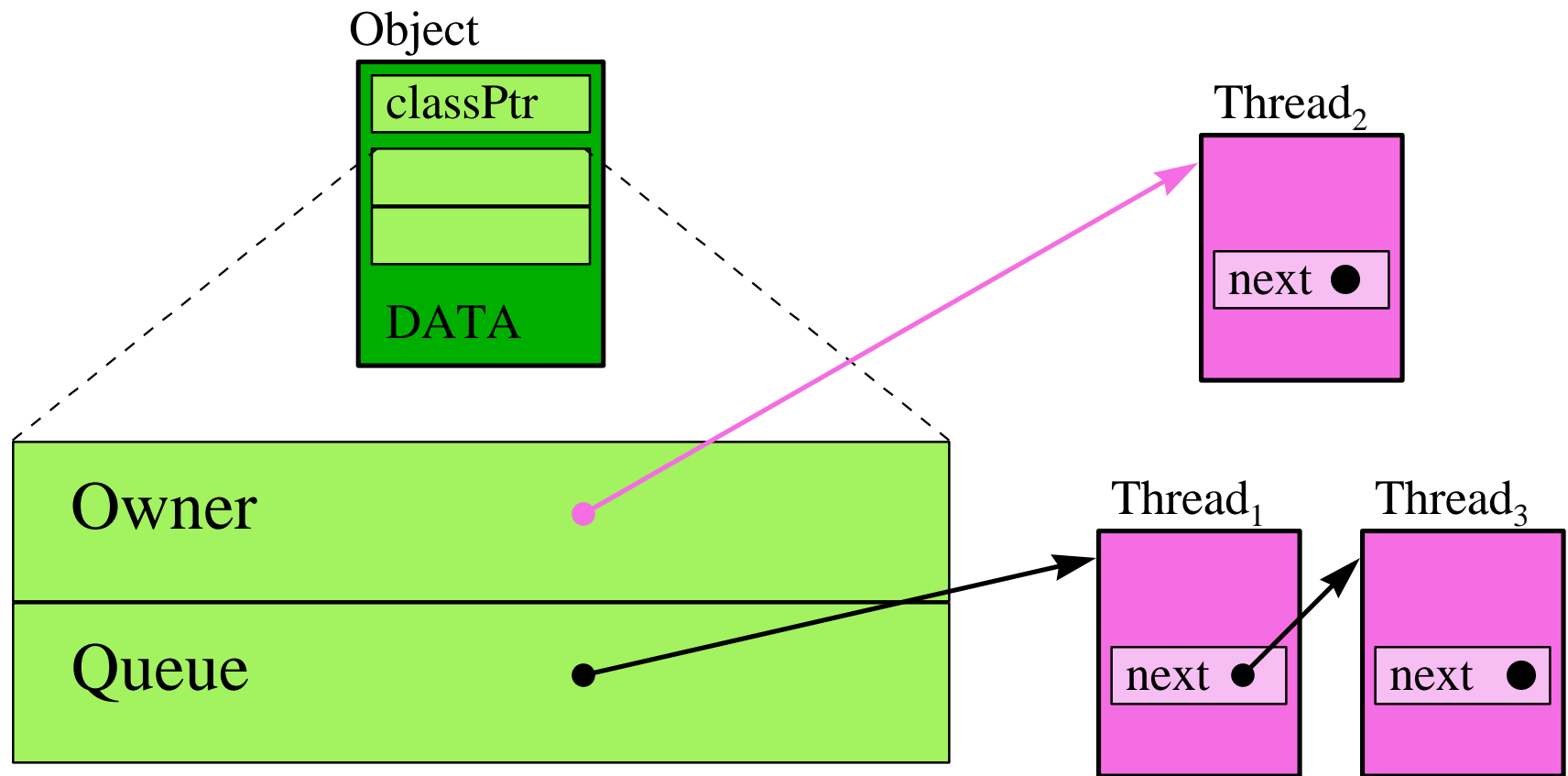
---

- ◆ Atomic compare-and-swap available
- ◆ Thread objects aligned on 8-byte boundaries.

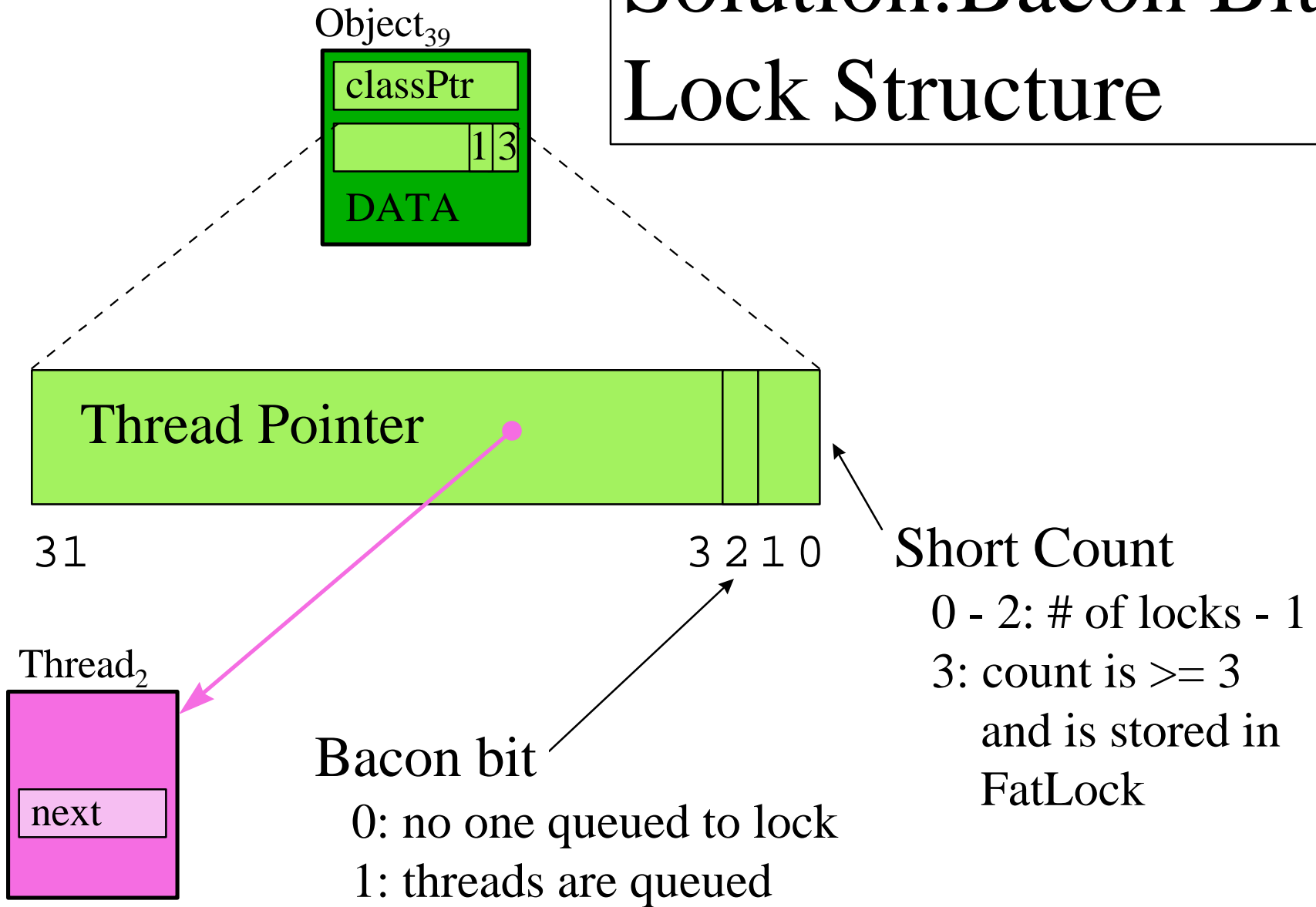
# Why Fast (Un)locking is Hard

---

- ◆ Must atomically release lock and check queue



# Solution: Bacon Bit Lock Structure



# Inline Lock Operation

---

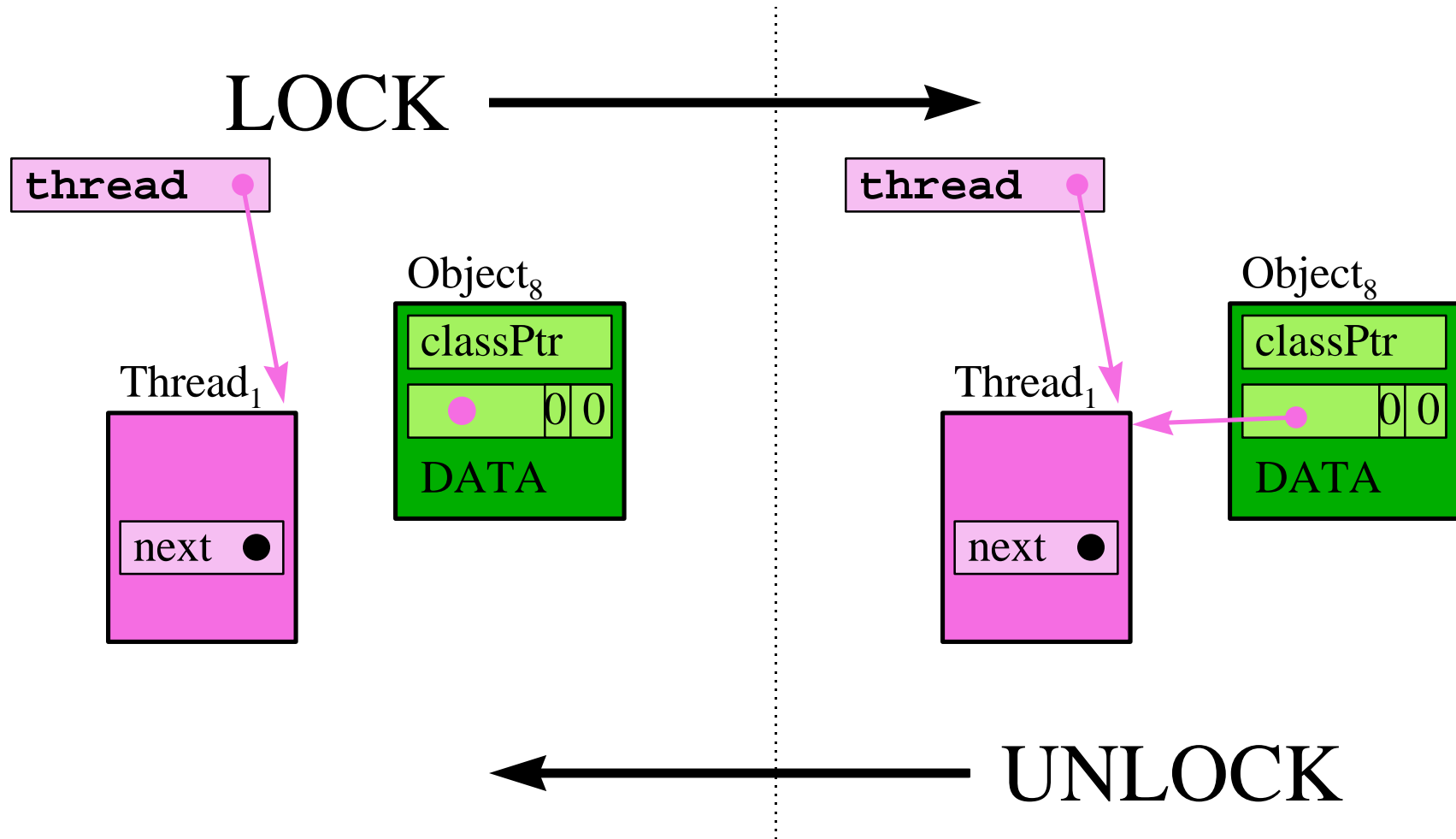
## ◆ Must check:

- no one owns the lock

```
inline void Monitor::enter() {  
    if (! CompareAndSwap(lockWord, 0, thread))  
        outOfLineEnter();  
}
```

# How Locking Works

---



# Inline Unlock Operation

---

## ◆ Must check:

- we own the lock
- lock count is 1
- no one is waiting for the lock

```
inline void Monitor::exit() {  
    if (! CompareAndSwap(lockWord, thread, 0))  
        outOfLineExit();  
}
```

# Race Conditions and Lock Transfer

---

- ◆ Problem: simultaneous unlock and enqueue
- ◆ Solution is in the Bacon bit
  - always set if a thread is enqueued on object.
  - never modified without acquiring globalLock on lockTable.
  - all changes to monitor lockWord must be via **CompareAndSwap( )**.
    - ◆ unless object *and* globalLock are locked

```

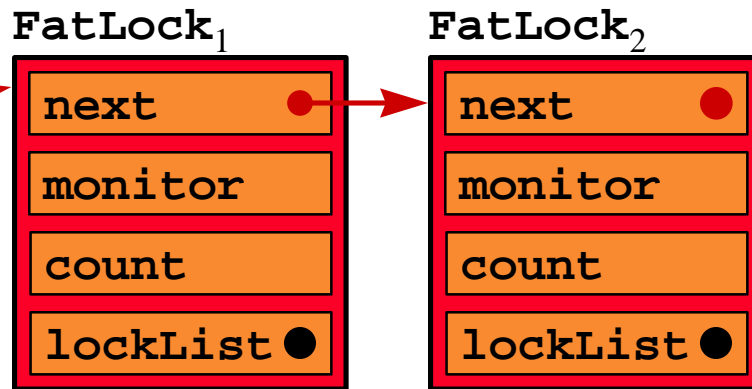
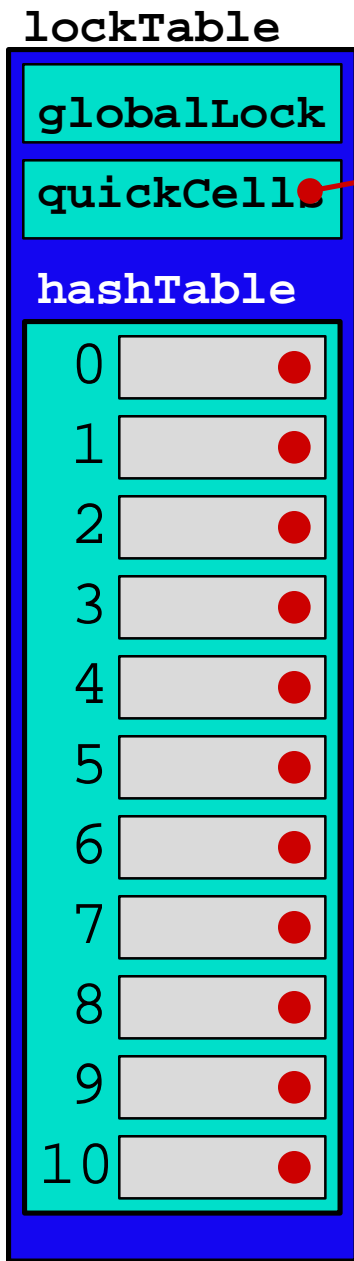
void Monitor::enqueueForLock() {
    lockTable.lock();
    while (true) {
        unsigned temp = lockWord;

        if (temp != 0 &&
            CompareAndSwap(lockWord, temp, temp|BaconBit)) {
            mon = lockTable.inflateMonitor(this);
            mon->addLocker(thread);
            lockTable.unlock();
            thread->suspend();
            return;
        }

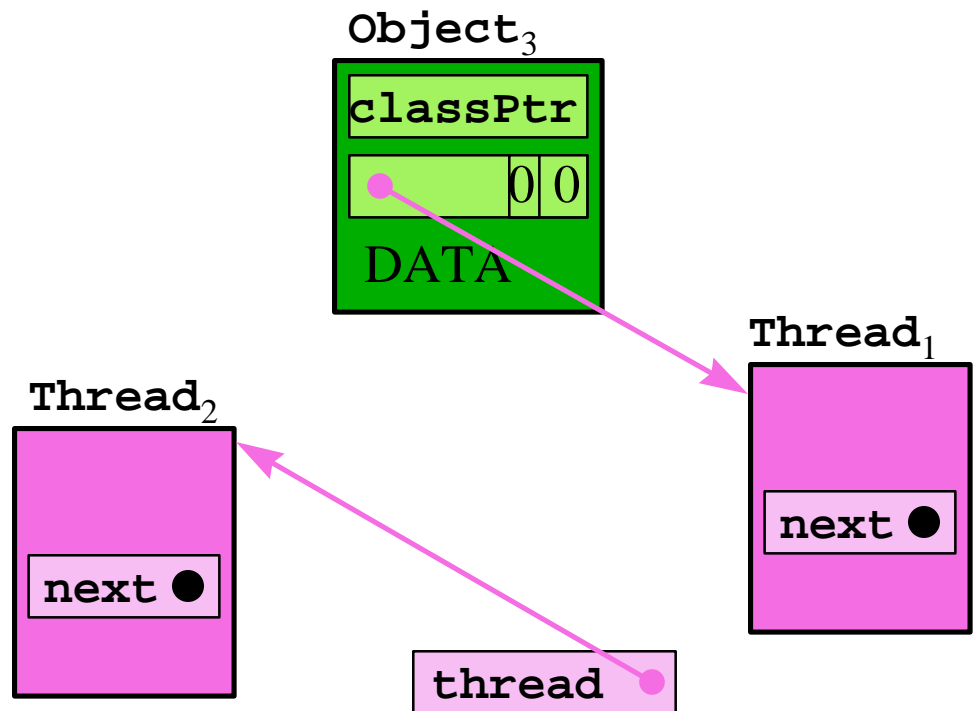
        if (CompareAndSwap(lockWord, 0, thread)) {
            lockTable.unlock();
            return;
        }
    }
}

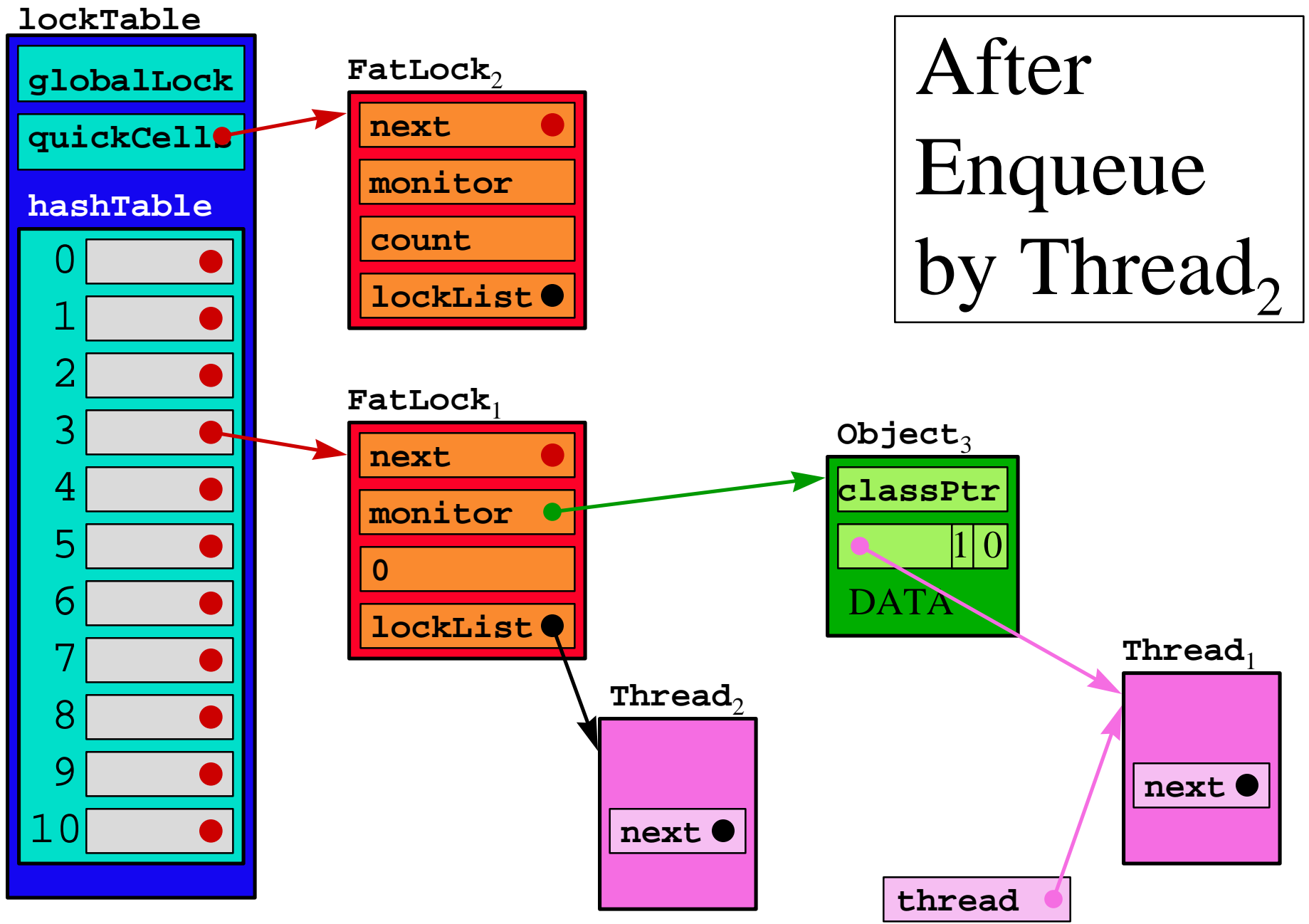
```

# Enqueue Operation



Before  
Enqueue  
by Thread<sub>2</sub>



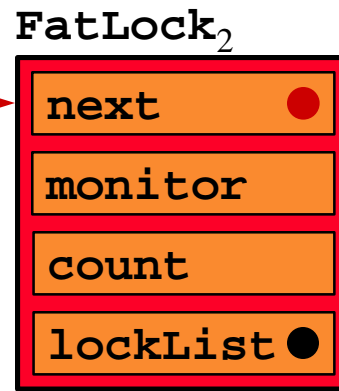
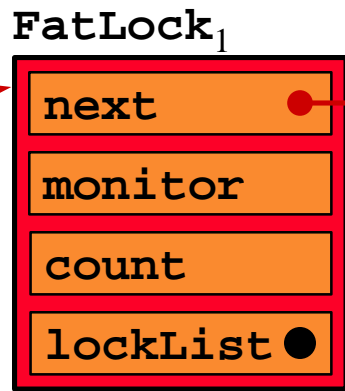
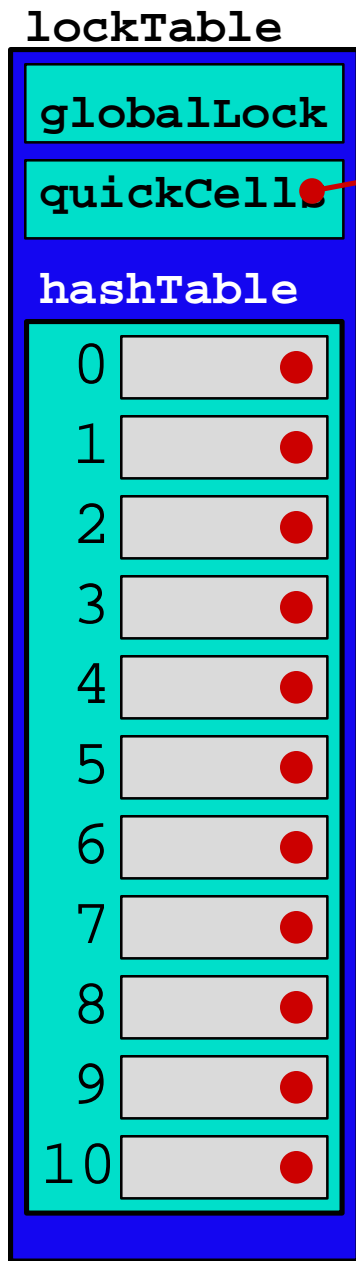


After  
Enqueue  
by Thread<sub>2</sub>

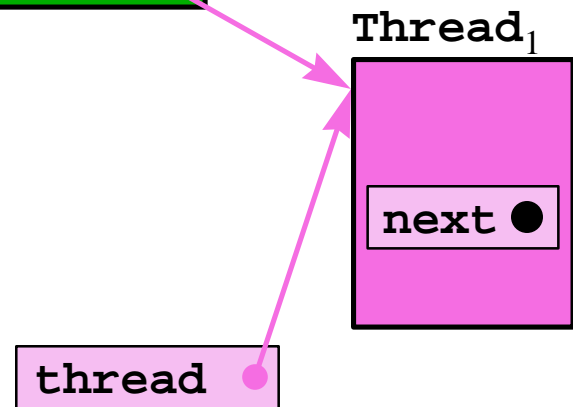
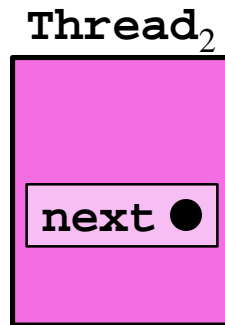
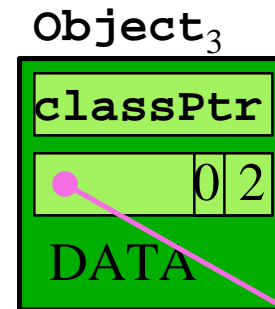
# Deeply Nested Locks

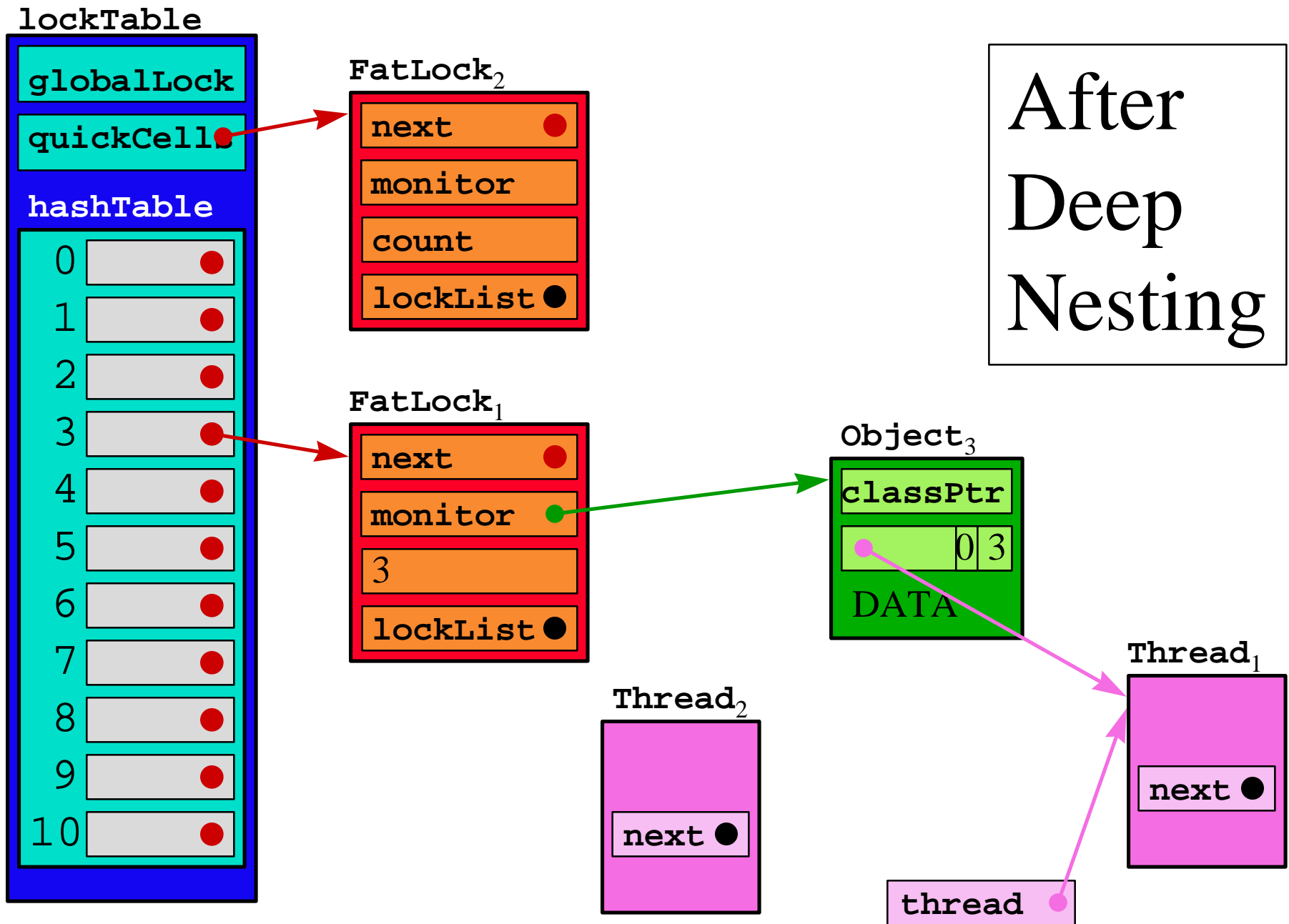
---

- ◆ When count reaches 3
  - lock globalLock
  - inflate monitor
  - set long count to 3
  - don't set Bacon bit

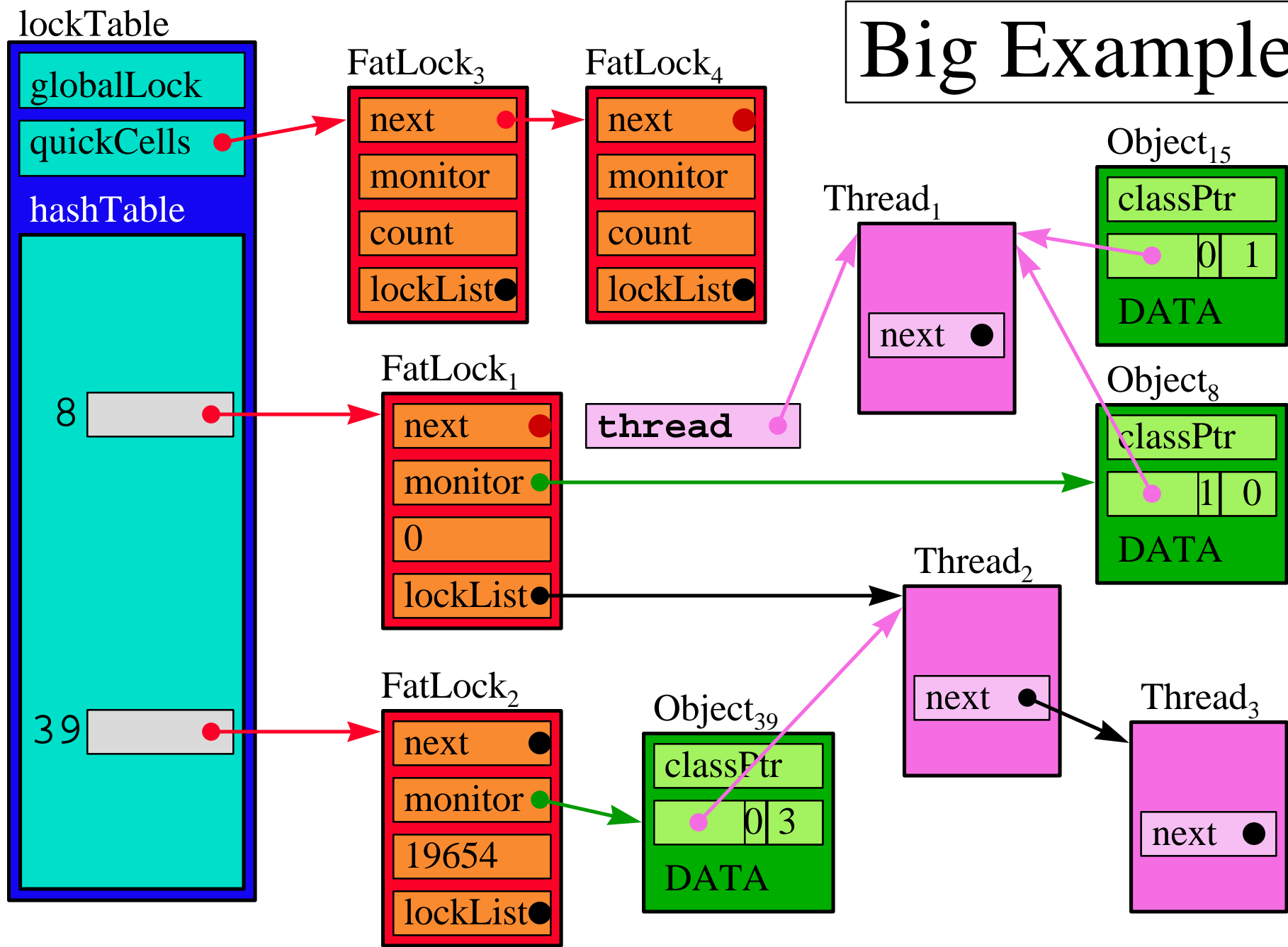


Before  
Deep  
Nesting





# Big Example



# Intel x86 Implementation (486 +)

---

- ◆ 7.5 cycles on Pentium
- ◆ forward jump to stub predicted not taken

```
LOCK: mov      ecx, THREAD      ; ebx is the this pointer
      xor      eax, eax        ; ecx = thread
      cmpxchg  [ebx], ecx      ; eax = 0
      jnz     stub            ; C&S(lockWord, eax, ecx)
lockdone:

stub: call    outOfLineLock    ; swap failed; try slowly
      j      lockdone         ; do it the slow way
                                ; and return
```

# Problem: Deeply Nested Locks

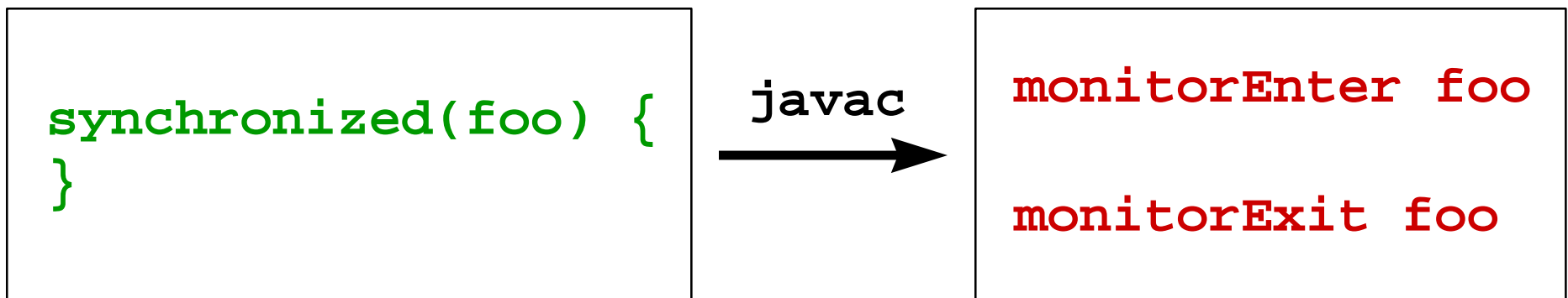
---

- ◆ Each FatLock access locks the global lock
  - decreases performance of recursive methods
  - increases global lock contention
- ◆ Solution: cache deeply nested FatLock
  - test for cache hit before locking global lock
  - If it's a hit, we've already locked the object

# Problem: `synchronized()` blocks

---

- ◆ Synchronized methods are lexically nested
- ◆ `synchronized()` blocks become bytecodes:
  - lexical nesting not checked by verifier
  - throwing exception might not release locks



# Solution for Non-nested Locking

---

- ◆ If possible, verify nesting at “compile-time”
- ◆ Else inflate monitor on non-nested lock
  - set Bacon bit in lockWord
  - search lockTable when thread is killed
  - unlock does not need modification

# Architectural Adaptations

---

- ◆ Uniprocessor with synchronous scheduler
  - don't need C&S **MOV**
- ◆ Uniprocessor with asynchronous scheduler
  - use atomic C&S in cache **CMPXCHG**
- ◆ Strongly ordered multiprocessor (Pentium)
  - use atomic C&S in RAM **LOCK#CMPXCHG**
- ◆ Weakly ordered multiprocessor (Pentium Pro)
  - atomic C&S and cache flush **LOCK#CMPXCHG**  
**CPUID**

# Advantages of Bacon-bit Locks

---

- ◆ Absolutely minimal cost for common case
  - 4 instructions/7.5 cycles on Pentium
  - compare to 6 cycles for no-op **CALL-RET**
- ◆ Next most common case also very fast
  - 10 instructions/17 cycles on Pentium
- ◆ Space overhead only 1 word per object
- ◆ Global lock only used in rare cases
- ◆ Co-exists with locks not lexically-nested

# Advantages of Bacon-bit Locks

---

- ◆ Scalable (can partition global lock)
- ◆ Almost no spin locking required
- ◆ Lock acquisition is fair
- ◆ Same implementation works and is fast on
  - synchronously scheduled uniprocessor
  - asynchronously scheduled uniprocessor
  - strongly ordered multiprocessor
  - weakly ordered multiprocessor