Practical Concurrent and Parallel Programming 12

Peter Sestoft
IT University of Copenhagen

Friday 2014-11-21*

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- Union-find data structure

- Possible parallel programming projects

Q 1

Lock-based queue with sentinel

TestMSqueue.java

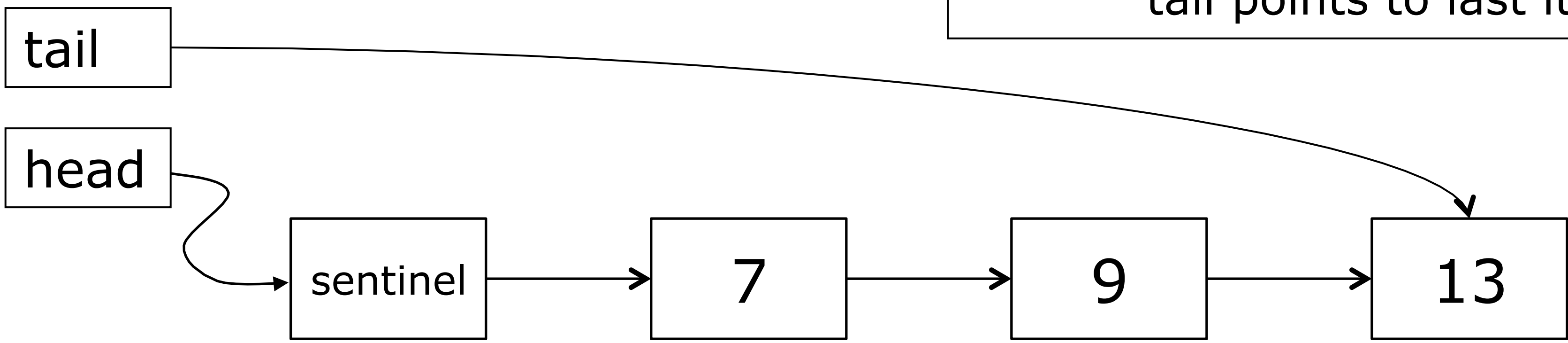
```
class LockingQueue<T> implements UnboundedQueue<T> {
  private Node<T> head, tail;

  public LockingQueue() {
    head = tail = new Node<T>(null, null);
  }
  ...
}
```

Make sentinel node

```
private static class Node<T> {
  final T item;
  Node<T> next;
}
```

Invariants:
tail.next=null
If empty, head=tail
If non-empty: head≠tail,
head.next is first item,
tail points to last item



Q 1

Lock-based queue operations

```
public synchronized void enqueue(T item) {
  Node<T> node = new Node<T>(item, null);
  tail.next = node;
  tail = node;
}
```

Enqueue at tail

TestMSqueue.java

```
public synchronized T dequeue() {
  if (head.next == null)
    return null;
  Node<T> first = head;
  head = first.next;
  return head.item;
}
```

Dequeue from second node, becomes new sentinel

- Important property:
  - Enqueue (**`put`**) updates **`tail`** but not **`head`**
  - Dequeue (**`take`**) updates **`head`** but not **`tail`**

Q 2

Michael-Scott lock-free queue, CAS

```
private static class Node<T> {
  final T item;
  final AtomicReference<Node<T>> next;
}
```

Michael and Scott: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, 1996

```
class MSQueue<T> implements UnboundedQueue<T> {
  private final AtomicReference<Node<T>> head, tail;

  public MSQueue() {
    Node<T> dummy = new Node<T>(null, null);
    head = new AtomicReference<Node<T>>(dummy);
    tail = new AtomicReference<Node<T>>(dummy);
  }
}
```

TestMSqueue.java

- If non-empty:
  - **`head.next`** is first item, **`tail`** points to last item ("quiescent state") or the second-last item ("intermediate state")

Q 2

Intermediate state and "help"

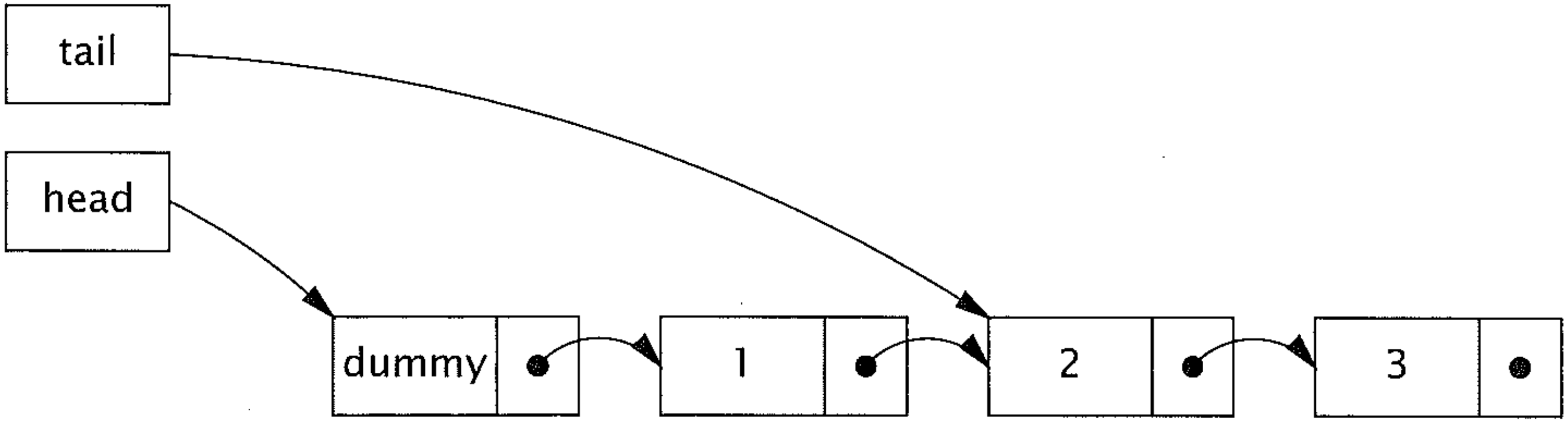


FIGURE 15.4. Queue in intermediate state during insertion.

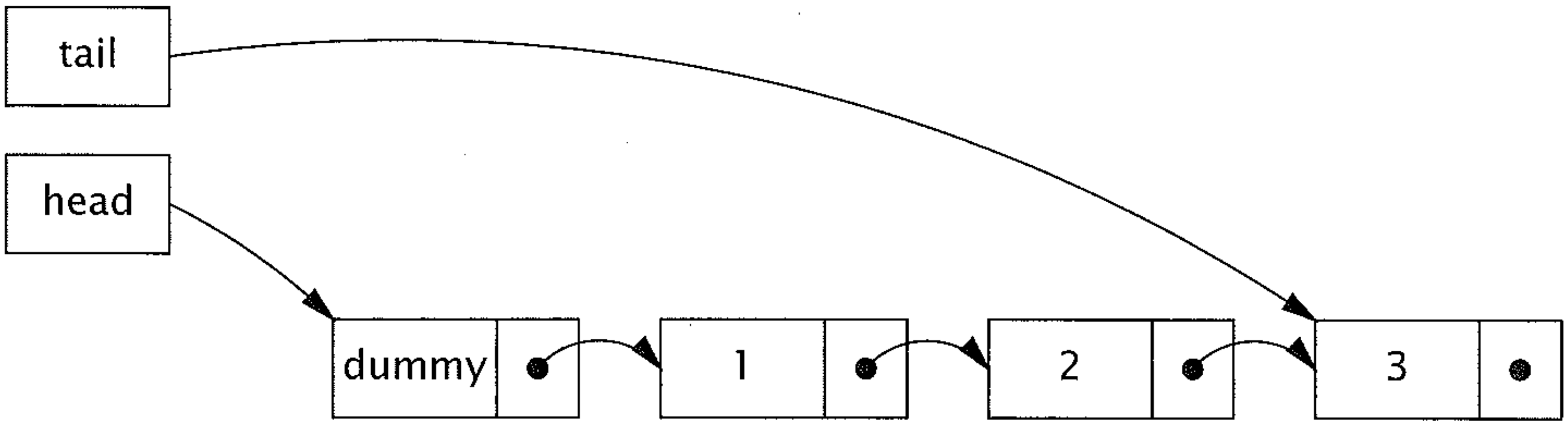
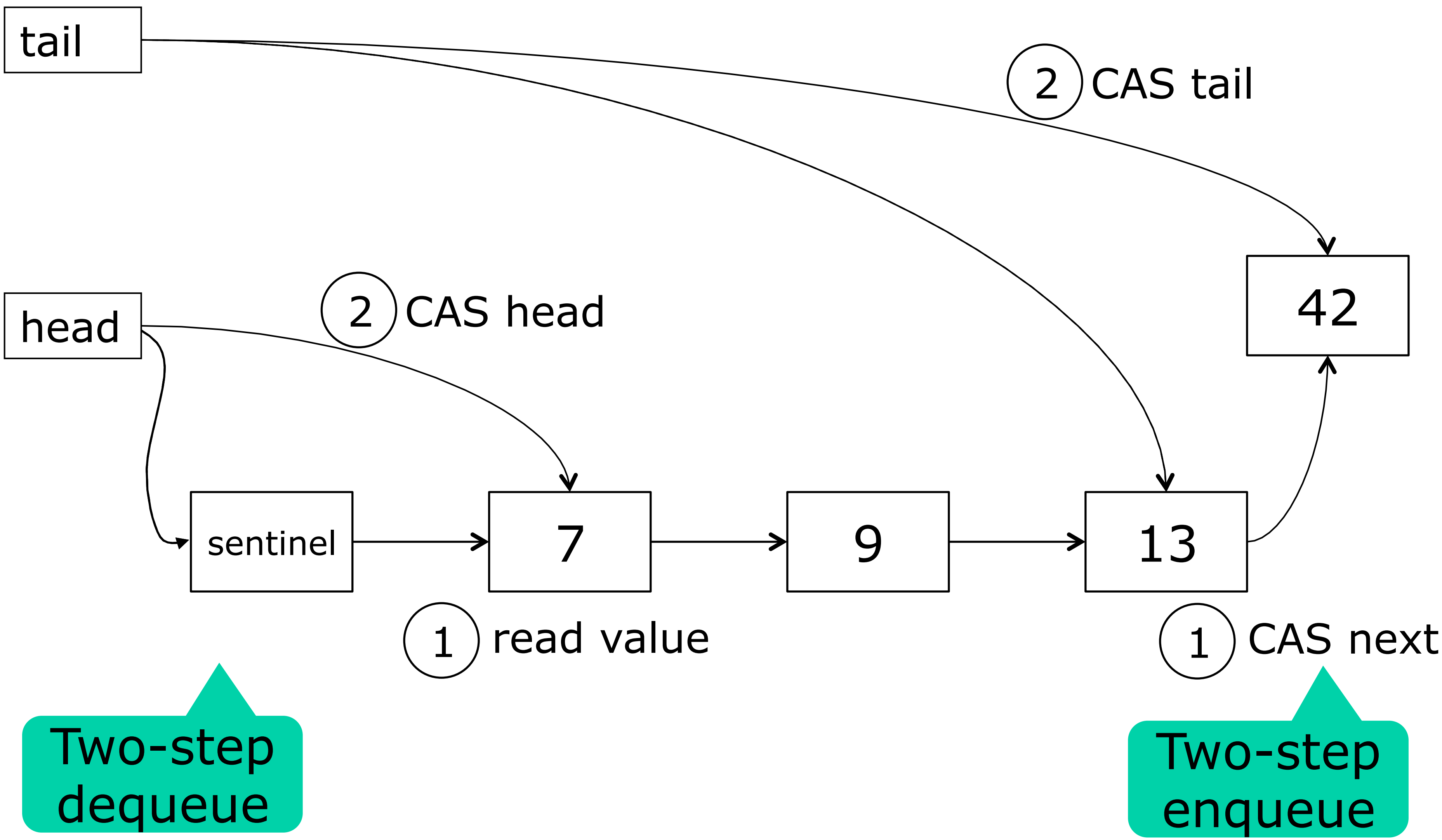


FIGURE 15.5. Queue again in quiescent state after insertion is complete.

Goetz p. 333

Michael & Scott queue operations



After Herlihy & Shavit p. 232

Michael-Scott dequeue (take)

```
public T dequeue() {
  while (true) {
    Node<T> first = head.get(),
            last = tail.get(),
            next = first.next.get();
    if (first == head.get()) {
      if (first == last) {
        if (next == null)
          return null;
        else
          tail.compareAndSet(last, next);
      } else {
        T result = next.item;
        if (head.compareAndSet(first, next)) {
          return result;
        }
      }
    }
  }
}
```

Needed?

Intermediate, try move tail (*)

1

2

Try move head

In Java or C#, but not C/C++, (1) can go after (2)

TestMSqueue.java

Q 2

Michael-Scott enqueue (put)

```
public void enqueue(T item) { // at tail
  Node<T> node = new Node<T>(item, null);
  while (true) {
    Node<T> last = tail.get(),
            next = last.next.get();
    if (last == tail.get()) {
      if (next == null)  {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      } else {
        tail.compareAndSet(last, next);
      }
    }
  }
}
```

Needed?

Quiescent, try add

Success, try move tail (1)

Intermediate, try move tail (2)

"help another enqueuer"

TestMSqueue.java

(*) Why must dequeue mess with the tail?


```
while (true) {
  ...
  if (first == last) {
    if (next == null)
      return null;
    else
      tail.compareAndSet(last, next);
  } else ...
}
```

Intermediate, try move tail

TestMSqueue.java

Queue is empty, head==tail

A: enqueue(7)

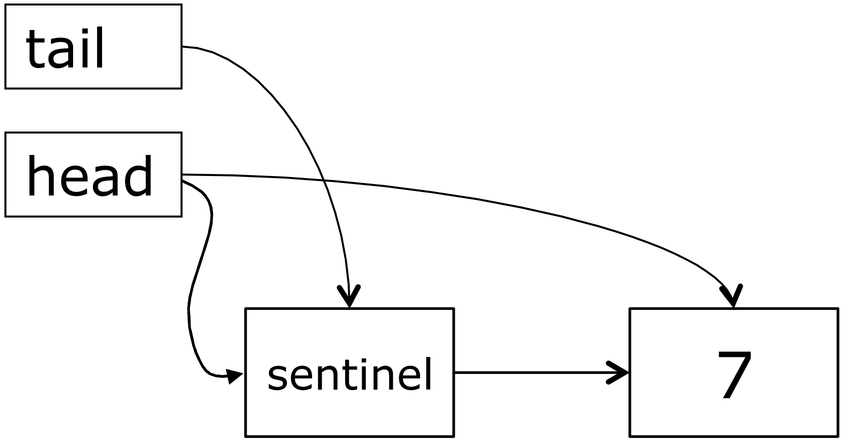
A: update a.next

B: dequeue()

B: update head

Now tail lags behind head, not good

So next dequeue should move tail before moving head



After Herlihy & Shavit p. 233

Understanding Michael-Scott queue

- Linearizable, with linearization points:
  - enqueue: successful CAS at E9
  - dequeue returning null: D3
  - dequeue returning item: successful CAS at D13
- Leariz'n point = where method takes effect

```
public void enqueue(T item) { // at tail
  Node<T> node = new Node<T>(item, null);
  while (true) {
    Node<T> last = tail.get(),
            next = last.next.get();
    if (last == tail.get()) { // E7
      if (next == null)  {
        if (last.next.compareAndSet(next, node)) {   // E9
          tail.compareAndSet(last, node);
          return;
        }
      } else
        tail.compareAndSet(last, next);
    }
  }
}
```

```
public T dequeue() { // from head
  while (true) {
    Node<T> first = head.get(),   // D3
            last = tail.get(),
            next = first.next.get();
    if (first == head.get()) { // D5
      if (first == last) {
        if (next == null)
          return null;
        else
          tail.compareAndSet(last, next);
      } else {
        T result = next.item;
        if (head.compareAndSet(first, next))   // D13
          return result;
      }
    }
  }
}
```

Groves: Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction, 2008

Nice, but ... needs a lot of AtomicReference objects

Q 3

```
private static class Node<T> {
  final T item;
  final AtomicReference<Node<T>> next;

  public Node(T item, Node<T> next) {
    this.item = item;
    this.next = new AtomicReference<Node<T>>(next);
  }
}
```

Q 2

Must be CAS'able

One AR per Node

```
private static class Node<T> {
  final T item;
  volatile Node<T> next;
  ...
}
```

Q 3

Better, no AtomicReference object needed

Instead, make an "updater"

A la Goetz p. 335

```
private final AtomicReferenceFieldUpdater<Node<T>, Node<T>> nextUpdater
  = AtomicReferenceFieldUpdater.newUpdater((Class<Node<T>>)(Class<?>)(Node.class),
                                           (Class<Node<T>>)(Class<?>)(Node.class),
                                           "next");
```

Michael-Scott enqueue, using the "updater" for **last.next**

Q 3

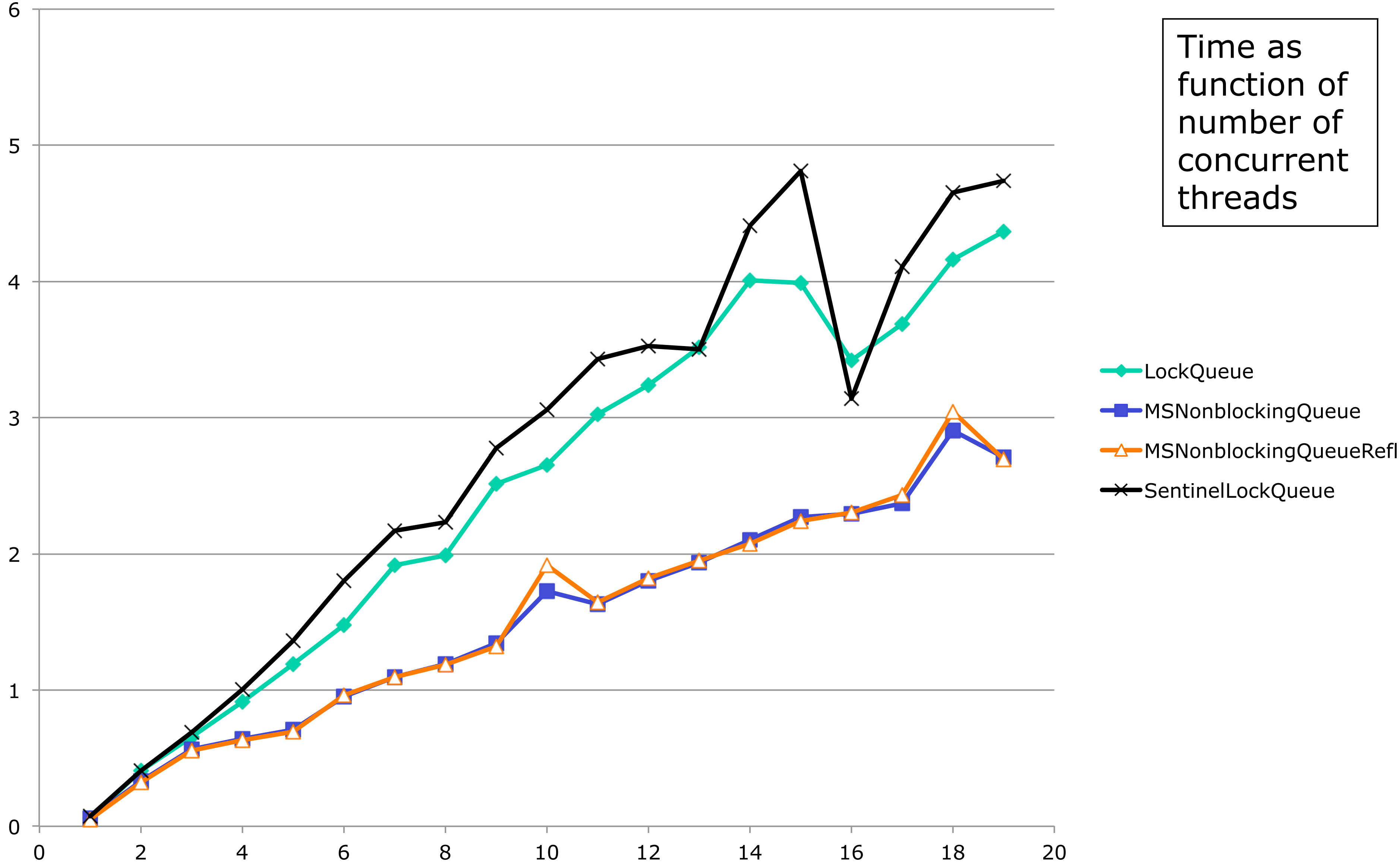
```
public void enqueue(T item) { // at tail
  Node<T> node = new Node<T>(item, null);
  while (true) {
    Node<T> last = tail.get(), next = last.next;
    if (last == tail.get()) {
      if (next == null)  {
        if (nextUpdater.compareAndSet(last, next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      } else {
        tail.compareAndSet(last, next);
      }
    }
  }
}
```

If "next" field of **last** equals **next**, set to **node**

Queue benchmarks

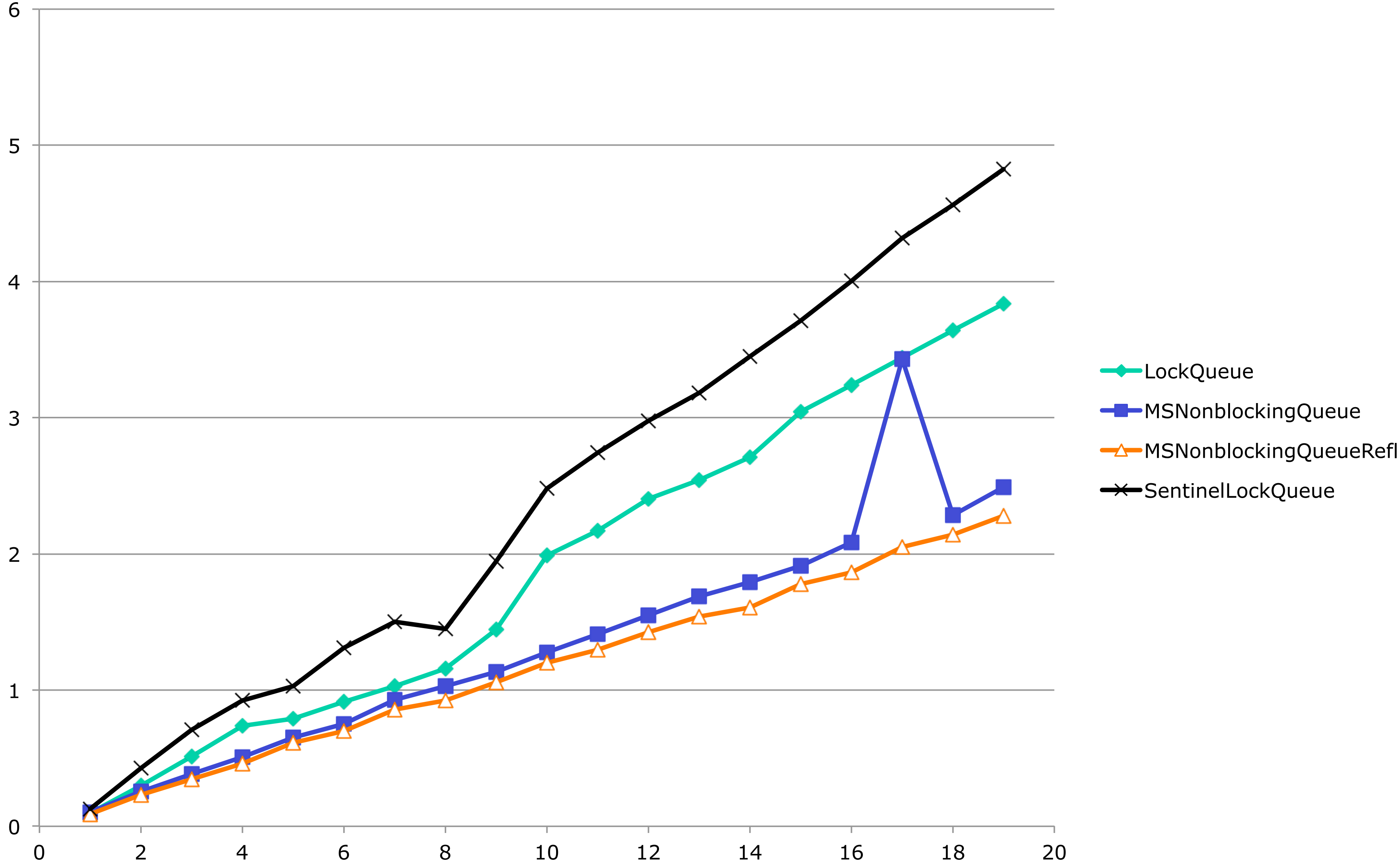
- Queue implementations
  - Lock-based
  - Lock-based, sentinel node
  - Lock-free, sentinel node, AtomicReference
  - Lock-free, sentinel node, AtomicReferenceFieldUpdater
- Platforms
  - Hotspot 64 bit Java 1.7.0_b147, Windows 7, Xeon W3505, 2.53GHz, 2 cores, 2009Q1
  - Hotspot 64 bit Java 1.6.0_37, MacOS, Core 2 Duo, 2.66GHz, 2 cores, 2008Q1
  - Icedtea Java 1.7.0_b21, Linux, Xeon E5320, 1.86GHz, 4/8 cores, 2006Q4
  - Hotspot 64 bit Java 1.7.0_25-b15, Linux, AMD Opteron 6386 SE, 32 cores, 2012Q4
- Measurements probably flawed: the client threads do no useful work, only en/dequeue
- Nevertheless, **big** differences between machines

Java 1.7, Xeon W3505, 2 cores

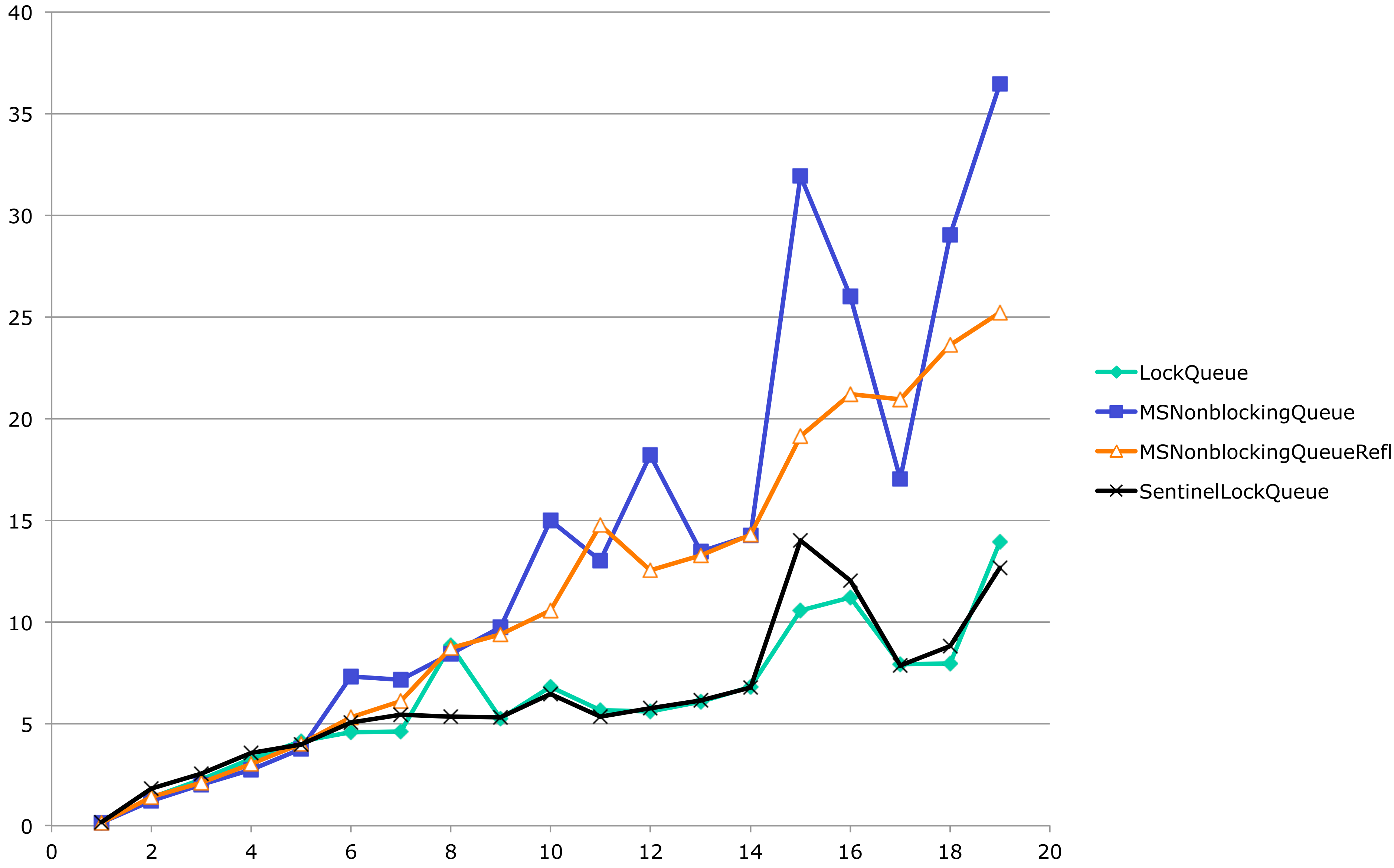


Time as function of number of concurrent threads

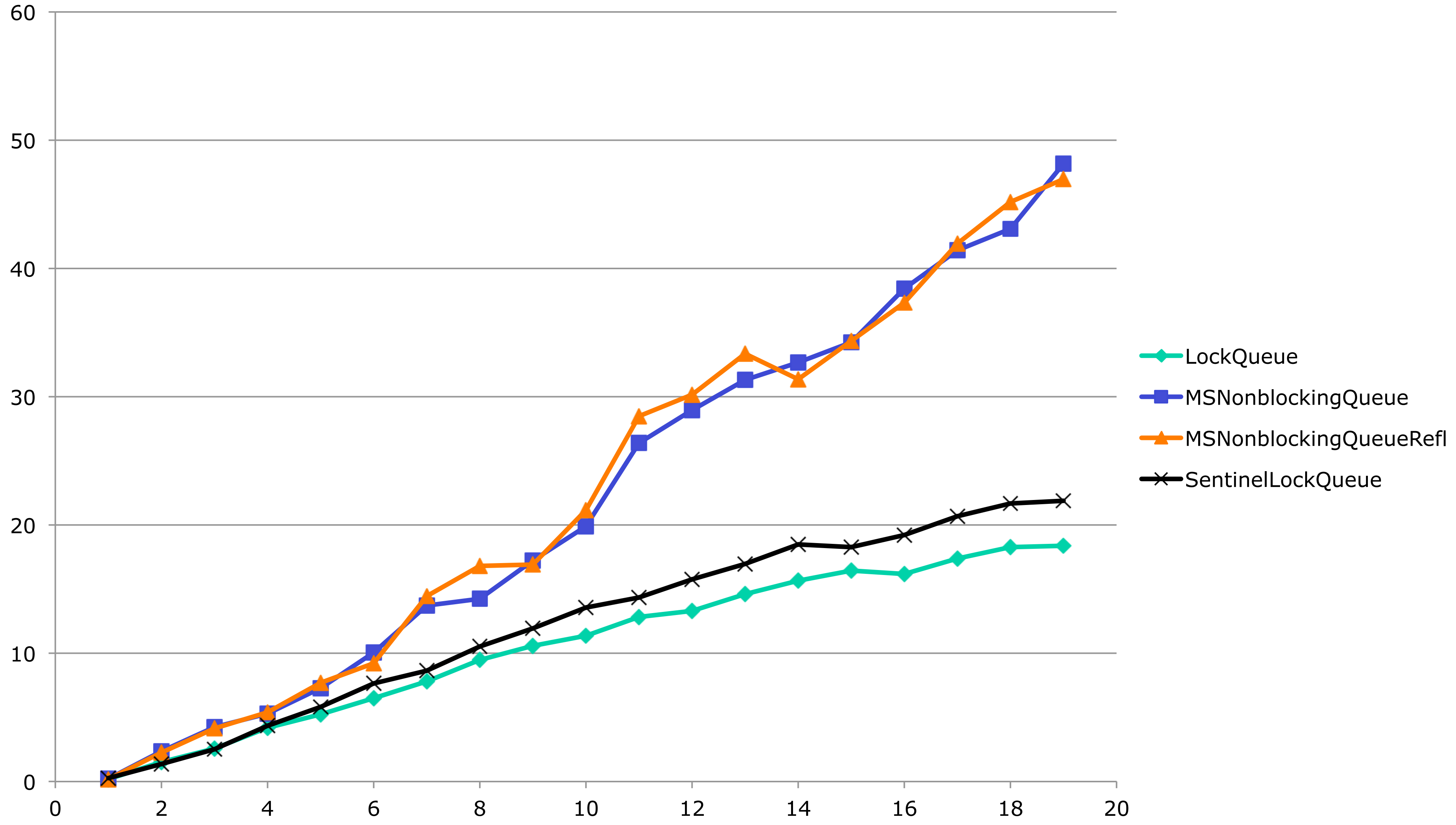
Java 1.6, Core 2 Duo, 2 cores



Java 1.7, Xeon E5320, 4/8 cores



Java 1.7, AMD Opteron, 32 cores



Plan for today

- Michael and Scott unbounded queue
- **Perspective: Work-stealing dequeues**
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- Union-find data structure

- Possible parallel programming projects

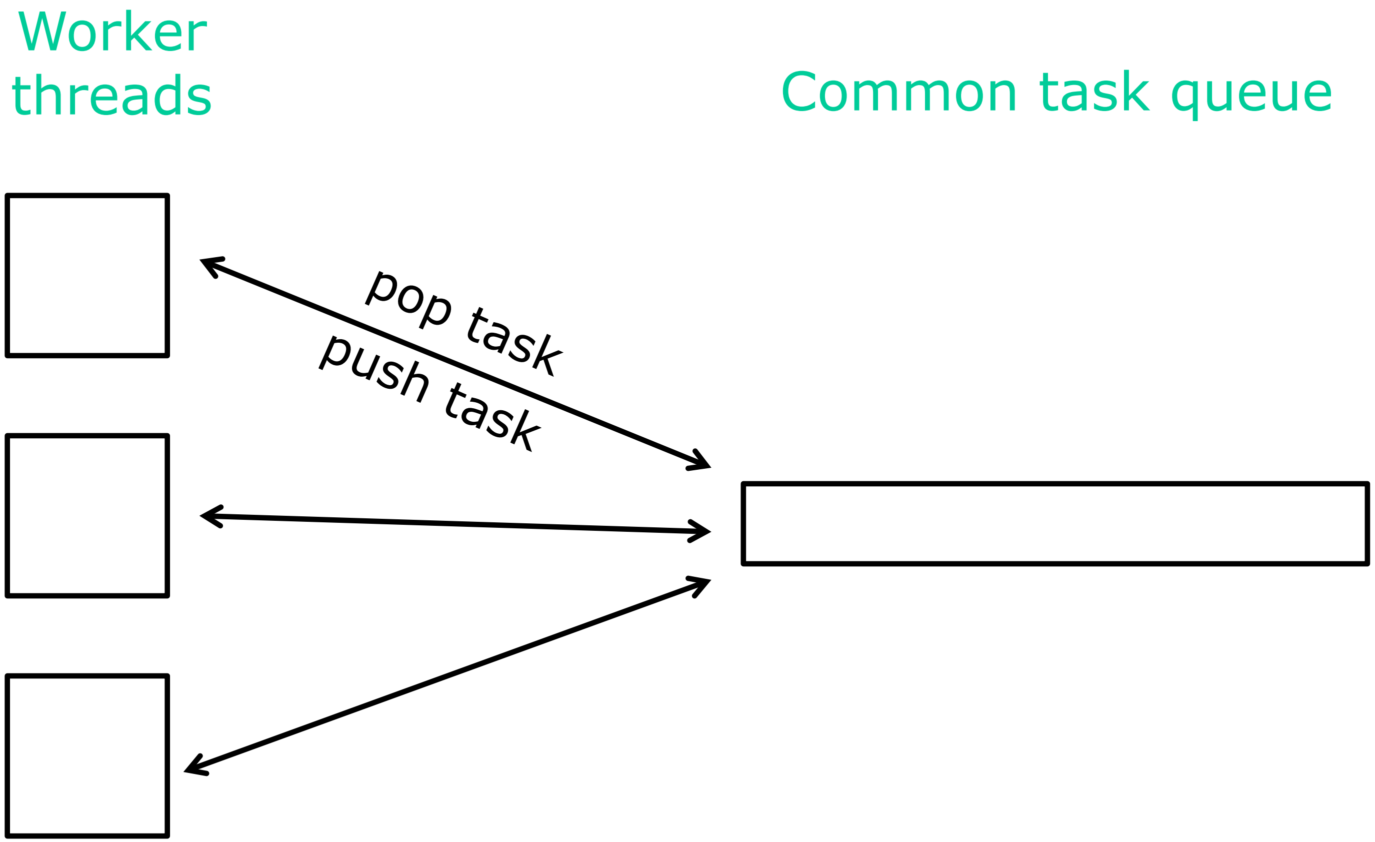
Perspective: Work-stealing dequeues

- Double-ended concurrent queues
- Used to implement
  - Java 7's Fork-Join framework, and Akka (wk 13-14)
  - Java 8's newWorkStealingPool executor
  - .NET 4.0 Task Parallel Library
- Chase and Lev: *Dynamic circular work-stealing queue*, SPAA 2005
- Michael, Vechev, Saraswat: *Idem-potent work stealing*, PPoPP 2009
- Leijen, Schulte, Burckhardt: *The design of a task parallel library*, OOPSLA 2009

Java 8 source

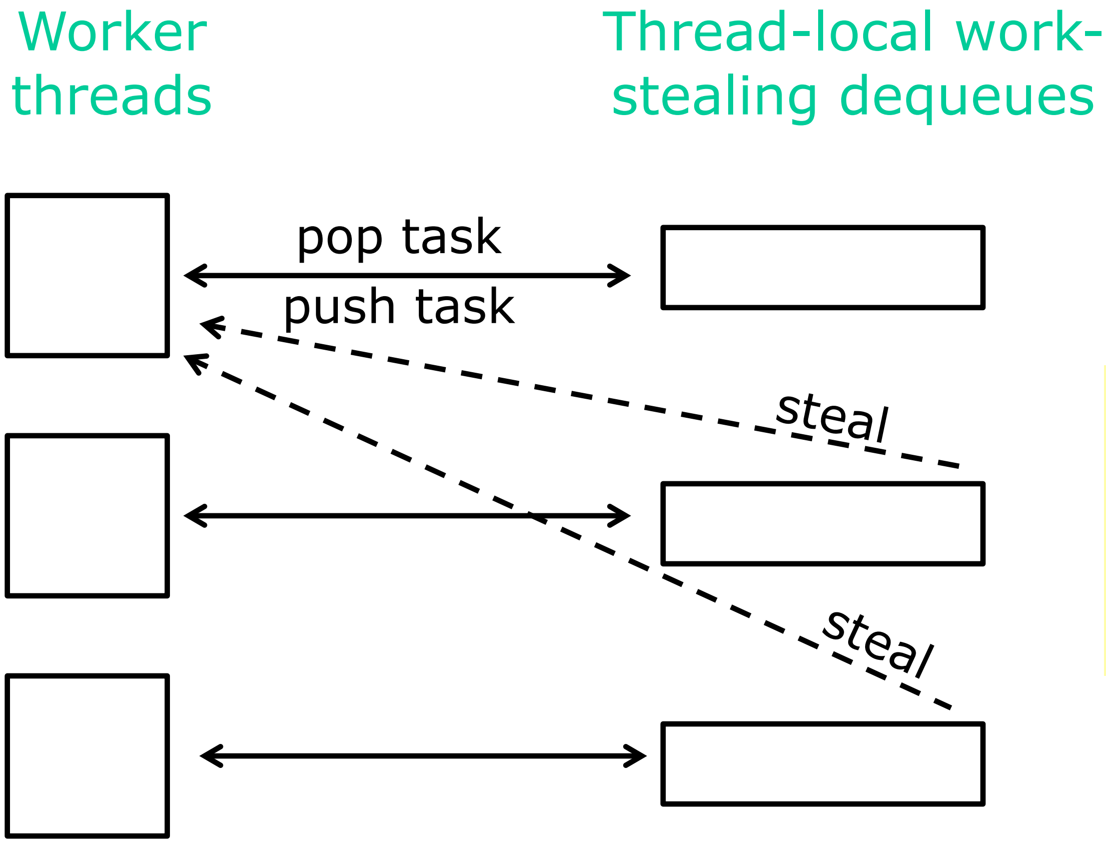
.NET TPL

A worker/task framework



- Worker threads pop and push tasks on queue
- **Not scalable** because single queue is used by many threads

Better worker/task framework




```
interface WSDeque<T> {
  void push(T item);
  T pop();
  T steal();
}
```

- Fewer memory write conflicts:
  - Most queue accesses are from local thread only
  - Pop from bottom, steal from top, conflicts are rare
- **Much better scalability**

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- **Progress concepts**
  - **Wait-free, lock-free, obstruction-free**
- Java Memory Model
- C#/.NET memory model
- Union-find data structure

- Possible parallel programming projects

Progress concepts

- *Non-blocking*: A call by thread A cannot prevent a call from thread B from completing
  - Not true for lock-based queue: A holds lock to **`put()`**, gets descheduled or crashes, while B wants to **`take()`** but cannot get lock
- *Wait-free*: Every call finishes in finite time
  - True for SimpleTryLock's **`tryLock`**
  - Not true for AtomicInteger's **`getAndAdd`**
- *Bounded wait-free*: Every ... in bounded time
- *Lock-free*: Some call finishes in finite time
  - True for AtomicInteger's **`getAndAdd`**
  - Any wait-free method is also lock-free
  - Lock-free is good enough in practice!

Goetz §15.4 and Herlihy & Shavit §3.7

Shavit et al, CACM November 2014, p. 13-15

Obstruction freedom

- *Obstruction-free*: If a method call executes alone, it finishes in finite time
  - Lock-based data structures are not obstruction-free
  - A *lock-free* method is also obstruction-free
  - Obstruction-free sounds rather weak, but in combination with back-off it ensures progress
  - Some people even think it too strong:

> ... we argue that obstruction-freedom is not an important property for software transactional memory, and demonstrate that, if we are prepared to drop the goal of obstruction-freedom, software transactional memory can be made significantly faster

Ennals 2006: STM should not be obstruction-free

Plan for today

- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- **Java Memory Model**
- **C#/.NET memory model**
- Union-find data structure

- Possible parallel programming projects

Why do I need a memory model?

- Threads in Java and C# and C etc *communicate* via mutable shared *memory*
- We need compiler optimizations for speed
  - Compiler optimizations that are harmless in thread A may seem strange from thread B
  - Disallowing strangeness leads to slow software
- We need CPU caches for speed
  - With caches, write-to-RAM order may seem strange
- So we have to live with some strangeness
- A memory model tells *how much* strangeness
- The Java Memory Model is quite well-defined
  - JLS §17.4, Goetz §16, Herlihy & Shavit §3.8

The happens-before relation in Java

- A *program order* of a thread t is some total order of the thread's actions that is consistent with the intra-thread semantics of t

- Action x *synchronizes-with* action y is defined as follows:
  - An unlock action on monitor m *synchronizes-with* all subsequent lock actions on m
  - A write to a volatile variable v *synchronizes-with* all subsequent reads of v by any thread
  - An action that starts a thread *synchronizes-with* the first action in the thread it starts
  - The write of the default value (zero, false, or null) to each variable *synchronizes-with* the first action in every thread
  - The final action in a thread T1 *synchronizes-with* any action in another thread T2 that detects that T1 has terminated
  - If thread T1 interrupts thread T2, the interrupt by T1 *synchronizes-with* any point where any other thread (including T2) determines that T2 has been interrupted

- Action x *happens-before* action y, written hb(x,y), is defined:
  - If x and y are actions of the same thread and x comes before y in *program order*, then hb(x, y)
  - There is a *happens-before* edge from the end of a constructor of an object to the start of a finalizer for that object
  - If an action x *synchronizes-with* a following action y, then we also have hb(x,y)
  - If hb(x, y) and hb(y, z), then hb(x, z) – that is, hb is transitive

Java Language Specification §17.4

Goetz §16.3.1

Strange but legal behavior in Java

- Java Language Specification, sect 17.4:
  - Run these code fragments in two threads
  - Shared fields A, B initially 0; local variables r1, r2

Thread 1

```
r2=A;
B=1;
```

Thread 2

```
r1=B;
A=2;
```

JLS 8 Tables 17.1, 17.5

- What are the possible results?
  - Strangely, r1==1 and r2==2 is possible
  - An ordering consistent with *happens-before* relation

```
B=1;
A=2;
r2=A;
r1=B;
```

Why permit such strange behaviors?

- More comprehensible example from JLS 17.4
  - Assume p, q shared, p==q and p.x==0

Thread 1:

```
r1 = p;
r2 = r1.x;
r3 = q;
r4 = r3.x;
r5 = r1.x;
```

Thread 2:

```
r6 = p;
r6.x = 3;
```

  - Compiler optimization, common subexpr. elimin.:

```
r1 = p;
r2 = r1.x;
r3 = q;
r4 = r3.x;
r5 = r2;
```

NB!

```
r6 = p;
r6.x = 3;
```

(p.x seems to switch from r2=0 to r4=3 and back to r5=0

- Using **`volatile x`** prevents this strangeness

Cost of volatile (week 4 flashback)

```
class IntArrayVolatile {
  private volatile int[] array;
  public IntArray(int length) { array = new int[length]; ... }
  public boolean isSorted() {
    for (int i=1; i<array.length; i++)
      if (array[i-1] > array[i])
        return false;
    return true;
  }
}
```

TestVolatileCost.java

```
IntArray                3.4 us        0.01      131072
IntArrayVolatile       17.2 us        0.14       16384
```

- In Java, volatile read is 5 x slower in this case
- C#/.NET 4.5, volatile read only 1.2 x slower
  - But still 3.7 x slower than Java non-volatile ...
- Mono .NET performs no optim., so no slower

VolatileArray.cs

Volatile prevents JIT optimizations

- For-loop body of **`isSorted`**, JITted x86 code:

```
0xdfff0: mov     0xc(%rsi),%r8d          ; LOAD %r8d = array field
0xdfff4: mov     %r10d,%r9d              ; i NOW IN %r9d
0xdfff7: dec     %r9d                    ; i-1 IN %r9d
0xdfffa: mov     0xc(%r12,%r8,8),%ecx    ; LOAD %ecx = array.length
0xdffff: cmp     %ecx,%r9d               ; INDEX CHECK array.length <= i-1
0xe0002: jae     0xe004b                 ; IF SO, THROW
0xe0004: mov     0xc(%rsi),%ecx          ; LOAD %ecx = array field
0xe0007: lea     (%r12,%r8,8),%r11       ; LOAD %r11 = array base addre
0xe000b: mov     0xc(%r11,%r10,4),%r11d  ; LOAD %r11d = arr[i-1]
0xe0010: mov     0xc(%r12,%rcx,8),%r8d   ; LOAD %r8d = array.length
0xe0015: cmp     %r8d,%r10d              ; INDEX CHECK array.length <= i
0xe0018: jae     0xe006d                 ; IF SO, THROW
0xe001a: lea     (%r12,%rcx,8),%r8       ; LOAD %r8 = array base address
0xe001e: mov     0x10(%r8,%r10,4),%r9d   ; LOAD %r9d = array[i]
0xe0023: cmp     %r9d,%r11d              ; IF arr[i] < array[i-1]
0xe0026: jg      0xe008d                 ; RETURN FALSE
0xe0028: mov     0xc(%rsi),%r8d          ; LOAD %r8d = array field
0xe002c: inc     %r10d                   ; i++
```

array volatile

3 reads of array field

2 index checks

VolatileArray.java

- Non-volatile: read **`arr`** once, unroll loop, ...:

```
0xcb9: mov     0xc(%rdi,%r11,4),%r8d     ; LOAD %rd8d = array[i-1]
0xcbe: mov     0x10(%rdi,%r11,4),%r10d   ; LOAD %rd10d = array[i]
0xcc3: cmp     %r10d,%r8d                ; IF array[i] > array[i-1]
0xcc6: jg      0xd85                     ; RETURN FALSE
```

array not volatile

C#/.NET memory model?

- Quite similar to Java
  - *C# Language Specification,* Ecma-334 standard
- But weaknesses and unclarities
  - C# **`readonly`** has no visibility effect unlike **`final`**
  - C# **`volatile`** is weaker than in Java
  - Allowed to lift variable read out of loop?
  - "Read introduction" seems downright horrible!

- If you write concurrent C# programs, read:
  - Ostrovsky: The C# Memory Model in Theory and Practice, MSDN Magazine, December 2012
  - Even though optional in this course

- Visibility effect of C#/.NET **`readonly`** fields not mentioned in C# Language Specification or Ecma-335 CLI Specification (**`initonly`**)
- In fact, no visibility guarantee is intended...

```
Right.  The CLI doesn't give any special status to initonly fields, from a memory ordering/visibility
perspective.  As with ordinary fields, if they are shared between threads then some sort of fence is needed to
ensure consistency.  This could be in the form of a volatile write, as Carol suggests, or any of the common
synchronization primitives such as releasing a lock, setting an event, etc.

Eric

-----Original Message-----
From: Carol Eidt
Sent: Tuesday, December 4, 2012 10:14 AM
To: Peter Sestoft; Mads Torgersen; Eric Eilebrecht
Cc: Carol Eidt
Subject: RE: Visibility (from other threads) of readonly fields in C#/.NET?

Hi Peter,

I apologize for not responding more quickly to your email.  I am adding Eric Eilebrecht to this thread, since he
is the CLR's memory ordering expert.

I believe that section I.12.6.4 Optimization addresses this, but one has to read between the lines:

"Conforming implementations of the CLI are free to execute programs using any technology that guarantees, within
a single thread of execution, that side-effects and exceptions generated by a thread are visible in the order
specified by the CIL. For this purpose only volatile operations (including volatile reads) constitute visible
side-effects. (Note that while only volatile operations constitute visible side-effects, volatile operations also
affect the visibility of non-volatile references.)"

Where it says " volatile operations also affect the visibility of non-volatile references", this implies (though
vaguely) that volatile reads & writes behave as some form of memory fence, though whether it is bi-directional or
acquire-release is left unstated.  I also believe that the above implies that, in order to achieve the desired
visibility of initonly fields, one would have to declare a volatile field that would be written last, effectively
"publishing" the other fields.

I certainly wouldn't say that the Java memory model make too much fuss over this - it's just fundamentally
tricky!

Carol
```

C#/.NET volatile weaker than Java's

```
class StoreBufferExample {
  volatile bool A = false;
  volatile bool B = false;
  volatile bool A_Won = false;
  volatile bool B_Won = false;
  public void ThreadA() {
    A = true;
    if (!B)
      A_Won = true;
  }
  public void ThreadB() {
    B = true;
    if (!A)
      B_Won = true;
  }
}
```

```
public void ThreadA() {
  A = true;
  Thread.MemoryBarrier();
  if (!B)
    aWon = 1;
}
```

```
public void ThreadB() {
  B = true;
  Thread.MemoryBarrier();
  if (!A)
    B_Won = true;
}
```

TestVolatile.cs

Ostrovsky 2013

- C#: possible to get **`A_won = B_won = true !`**
  - Not JIT compiler, but CPU store buffer delay on A
  - To fix in C#, add MemoryBarrier call (no Java equ.)

C# volatile vs Java volatile

- A read of a volatile field is called a volatile read. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a volatile write. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

C# Language Spec 2006, §17.4.3

- A C# volatile read may move earlier, a volatile write may move later, hence trouble
- Not in Java:

If a programmer protects all accesses to shared data via locks or declares the fields as volatile, she can forget about the Java Memory Model and assume interleaving semantics, that is, Sequential Consistency.

Lochbichler: Making the Java memory model safe, ACM TOPLAS, December 2013

MemoryBarrier() in C#/.NET

> Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.

> MemoryBarrier is required only on multiprocessor systems with weak memory ordering (for example, a system employing multiple Intel Itanium processors).

System.Threading.Thread.MemoryBarrier API docs 4.5

- But seems sometimes to be needed anyway
  - also on x86
- Java does not have such a method, because Java **`volatile`** gives better guarantees

Plan for today

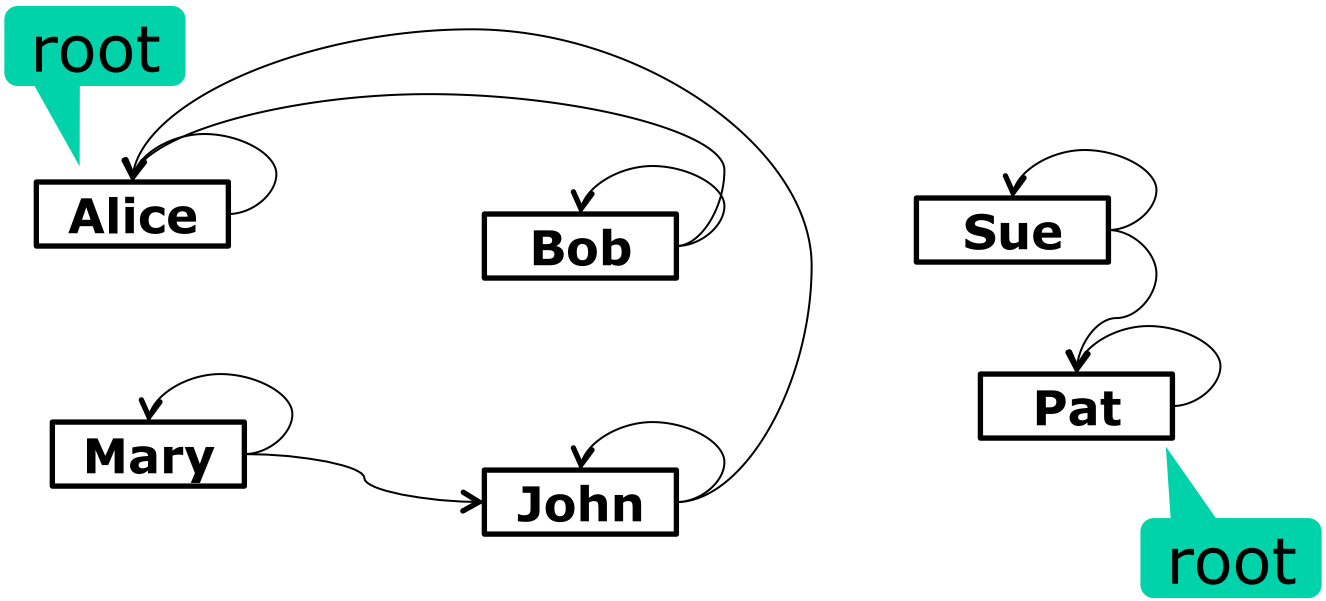
- Michael and Scott unbounded queue
- Perspective: Work-stealing dequeues
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- Java Memory Model
- C#/.NET memory model
- **Union-find data structure**

- Possible parallel programming projects

The union-find data structure

- Efficient way to maintain equivalence classes
- Used in
  - type inference in compilers: F#, Scala, C# ...
  - image segmentation
  - network analysis: chips, WWW, Facebook friends ...
- Example: family relations, who are related?

Tarjan: Data structures and network algorithms, 1983



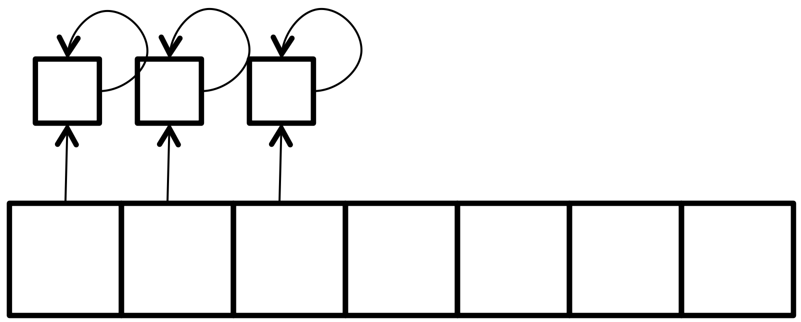
Sue is Pat's sister
Alice is Bob's sister
Mary is John's mother
Mary is Bob's mother

Are Sue and Mary related?

Three union-find implementations

- A: Coarse-locking = Synchronized methods
- B: Fine-locking = Lock on each set partition
- C: Wait-free = Optimistic, CAS-based

```
interface UnionFind {
  int find(int x);
  void union(int x, int y);
  boolean sameSet(int x, int y);
}
```



```
class Node {
  volatile int
    next, rank;
}
```

```
class CoarseUnionFind implements UnionFind {
  private final Node[] nodes;

  public CoarseUnionFind(int count) {
    this.nodes = new Node[count];
    for (int x=0; x<count; x++)
      nodes[x] = new Node(x);
  }
```

TestUnionFind.java

Coarse-locking union-find

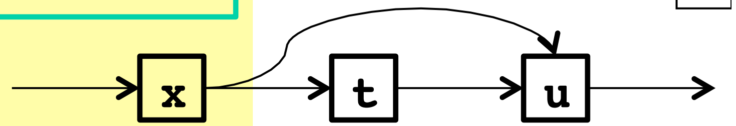
```
class CoarseUnionFind implements UnionFind {
  private final Node[] nodes;
  public synchronized int find(int x) {
    while (nodes[x].next != x) {
      final int t = nodes[x].next, u = nodes[t].next;
      nodes[x].next = u;
      x = u;
    }
    return x;
  }
  public synchronized void union(int x, int y) {
    int rx = find(x), ry = find(y);
    if (rx == ry)
      return;
    if (nodes[rx].rank > nodes[ry].rank) {
      int tmp = rx; rx = ry; ry = tmp;
    }
    nodes[rx].next = ry;
    if (nodes[rx].rank == nodes[ry].rank)
      nodes[ry].rank++;
  }
}
```

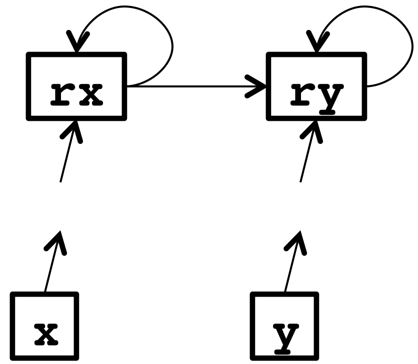
Path halving

Find roots

Union by rank

TestUnionFind.java





Fine-locking union-find

- No locking in find
  - Do path compression separately
  - Ensure visibility by **`volatile next, rank`** in Node

```
class FineUnionFind implements UnionFind {
  public int find(int x) {
    while (nodes[x].next != x)
      x = nodes[x].next;
    return x;
  }

  // Assumes lock is held on nodes[root]
  private void compress(int x, final int root) {
    while (nodes[x].next != x) {
      int next = nodes[x].next;
      nodes[x].next = root;
      x = next;
    }
  }
}
```

No path halving

Path compression

TestUnionFind.java

Fine-locking union-find

```
public void union(final int x, final int y) {
  while (true) {
    int rx = find(x), ry = find(y);
    if (rx == ry)
      return;
    else if (rx > ry) {
      int tmp = rx; rx = ry; ry = tmp;
    }
    synchronized (nodes[rx]) {
      synchronized (nodes[ry]) {
        if (nodes[rx].next != rx || nodes[ry].next != ry)
          continue;
        if (nodes[rx].rank > nodes[ry].rank) {
          int tmp = rx; rx = ry; ry = tmp;
        }
        nodes[rx].next = ry;
        if (nodes[rx].rank == nodes[ry].rank)
          nodes[ry].rank++;
        compress(x, ry);
        compress(y, ry);
    } }
} }
```

Consistent lock order

Restart if updated

Union by rank and path compression

TestUnionFind.java

Wait-free union-find with CAS

UF C

Anderson and Woll: Wait-free parallel algorithms for the union-find problem, 1991

```
class Node {
  private final AtomicInteger next;
  private final int rank;
}
```

```
public int find(int x) {
  while (nodes.get(x).next.get() != x) {
    final int t = nodes.get(x).next.get(),
              u = nodes.get(t).next.get();
    nodes.get(x).next.compareAndSet(t, u);
    x = u;
  }
  return x;
}
```

Path halving with CAS

TestUnionFind.java

Atomic update of root **nodes[x]** to point to fresh Node(y,newRank)

```
boolean updateRoot(int x, int oldRank, int y, int newRank) {
  final Node oldNode = nodes.get(x);
  if (oldNode.next.get() != x || oldNode.rank != oldRank)
    return false;
  Node newNode = new Node(y, newRank);
  return nodes.compareAndSet(x, oldNode, newNode);
}
```

Wait-free union-find: union

```
public void union(int x, int y) {
  int xr, yr;
  do {
    x = find(x);
    y = find(y);
    if (x == y)
      return;
    xr = nodes.get(x).rank;
    yr = nodes.get(y).rank;
    if (xr > yr || xr == yr && x > y) {
      { int tmp = x; x = y; y = tmp; }
      { int tmp = xr; xr = yr; yr = tmp; }
    }
  } while (!updateRoot(x, xr, y, xr));
  if (xr == yr)
    updateRoot(y, yr, y, yr+1);
  setRoot(x);
}
```

Union-by-rank, deterministic

Restart if updated

TestUnionFind.java

Some PCPP-related thesis projects

- Design, implement and test concurrent versions of C5 collection classes for .NET
  - http://www.itu.dk/research/c5/
- The *Popular Parallel Programming (P3)* project
  - Static dataflow partitioning algorithms
  - Dynamic scheduling algorithms on .NET
  - Vector (SSE, AVX) .NET intrinsics for spreadsheets
  - Supercomputing with Excel and .NET
  - http://www.itu.dk/people/sestoft/p3/
- Investigate Java Pathfinder for test and coverage analysis of concurrent software
  - http://babelfish.arc.nasa.gov/trac/jpf

This week

- Reading
  - Michael & Scott 1996: *Simple, fast, and practical non-blocking and blocking concurrent queue ...*
  - Goetz chapter 15 and 16
  - Herlihy & Shavit section 3.8
  - Optional: JLS 8 §17.4
- Exercises
  - Test and experiment with the lock-free Michael & Scott queue
- Read before next week – Claus lectures!
  - Armstrong, Virding, Williams: *Concurrent programming in Erlang*, chapters 1, 2, 5, 11.1