# Practical Concurrent and Parallel Programming 11

Riko Jacob
IT University of Copenhagen

Friday 2017-11-24

# Plan for today

- **Michael and Scott unbounded queue 1996**
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- Union-find data structure
- Work-stealing dequeues
  - Chase-Lev dequeue 2005

Based on slides by
Peter Sestoft

# Bonus: More on volatile and CAS speed

TestTimeCas.java

- Int field increment:

```
data.x = data.x + 1;
```

  - Single thread; and non-volatile or volatile
- AtomicInteger "incr":

```
int old = data.get();
data.compareAndSet(old, old+1);
```

  - Single thread
  - Single thread, one other interfering thread
  - Single thread, one other non-interfering thread
- Results

| Activity | Time/ns |
|---|---|
| Non-volatile field `x` | 0.9 |
| Volatile field `x` | 8.8 |
| CAS alone | 11.4 |
| CAS with interfering thread | 74.5 |
| CAS with non-interfering thread | 11.7 |

# Lock-based queue with sentinel

Q 1

TestMSqueue.java

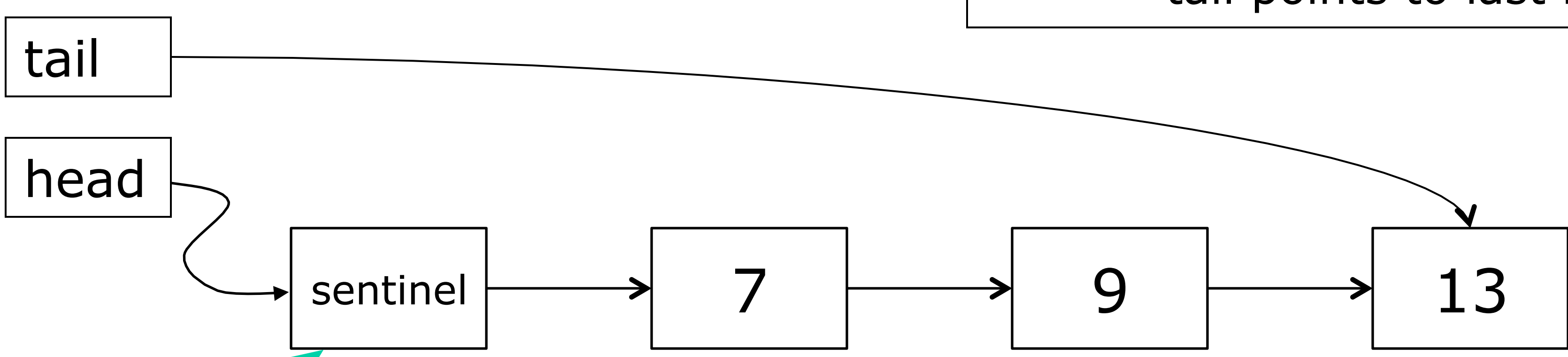
```
class LockingQueue<T> implements UnboundedQueue<T> {
  private Node<T> head, tail;

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

Make sentinel node

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

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




Purpose: Avoid special case for empty queue

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;
}
```

Atomic

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;
}
```

Atomic

Dequeue from second node, second becomes new sentinel

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

# Michael-Scott lock-free queue, CAS

Q 2

```
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);
  }
}
```

Make sentinel node

TestMSQueue.java

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

Q 2

# Intermediate state and "help"

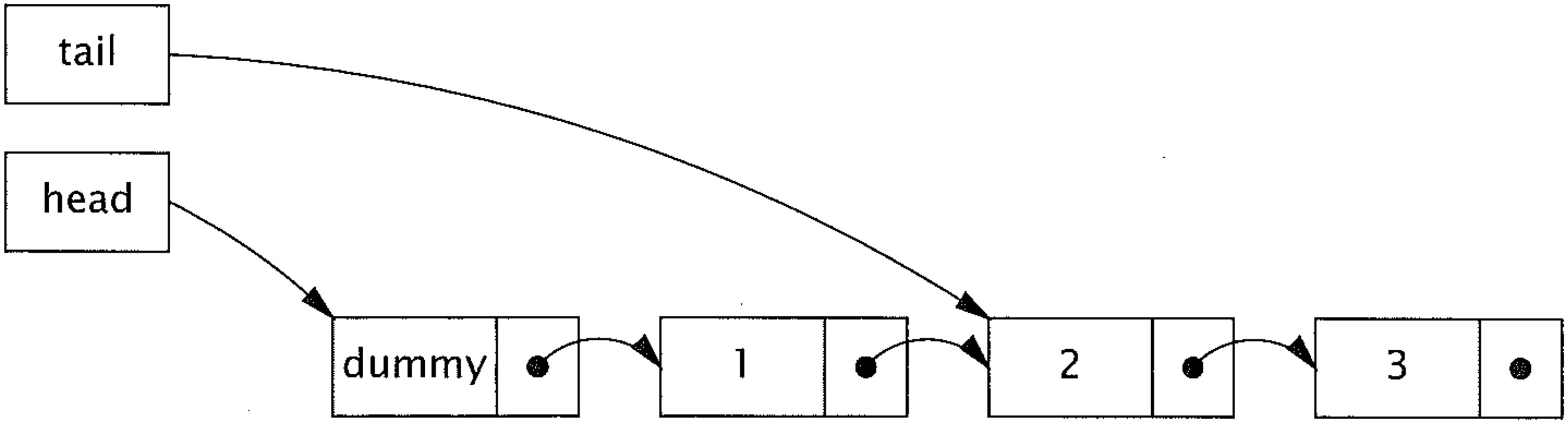



FIGURE 15.4. Queue in intermediate state during insertion.

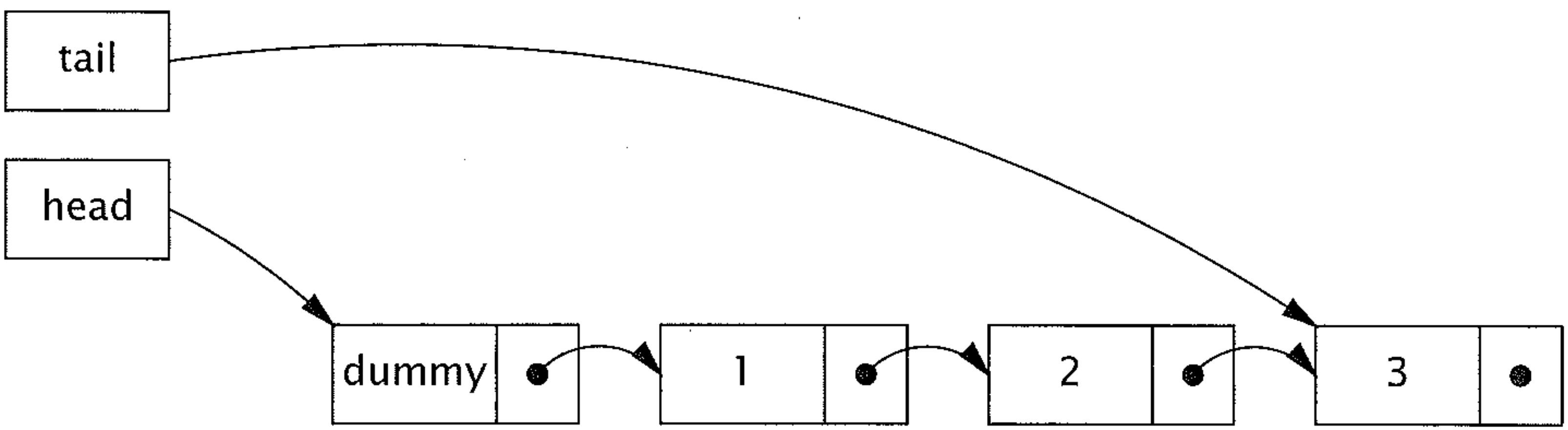
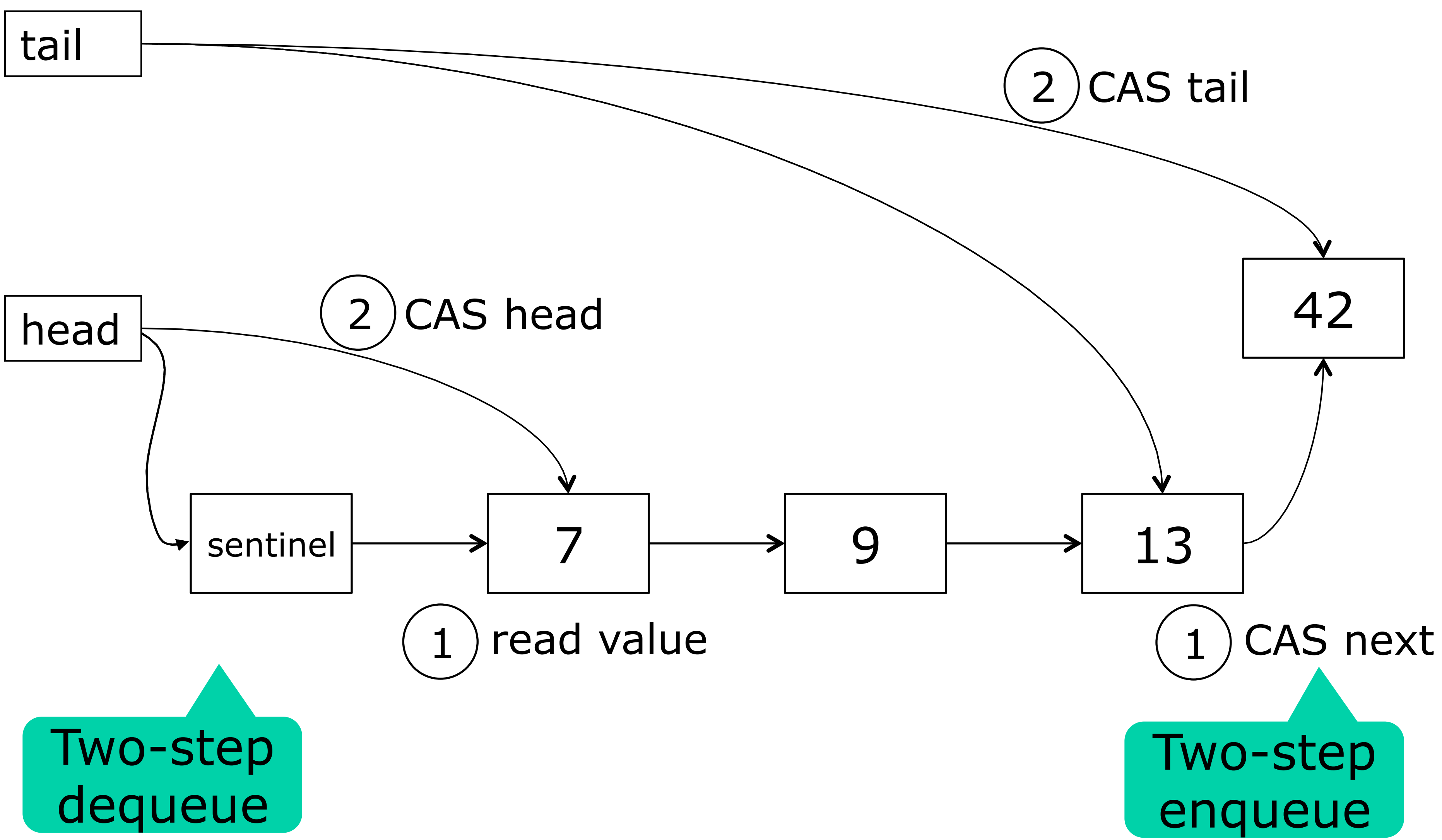



FIGURE 15.5. Queue again in quiescent state after insertion is complete.

Goetz p. 333

# Michael & Scott queue operations




After Herlihy & Shavit p. 232

Q 2

# 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?

May be empty

Is empty

Intermediate, try move tail

Try move head

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

Intermediate, try move tail

"help another enqueuer"

TestMSqueue.java

# Why must dequeue mess with the tail?

Q 2

```
public T dequeue() {
  ...
  if (first == last) {
    if (next == null)
      return null;
    else
      tail.compareAndSet(last, next);
  } else ...
}
```

Intermediate, try move tail

TestMSqueue.java

Scenario without it:
If queue empty, head==tail

A: enqueue(7)

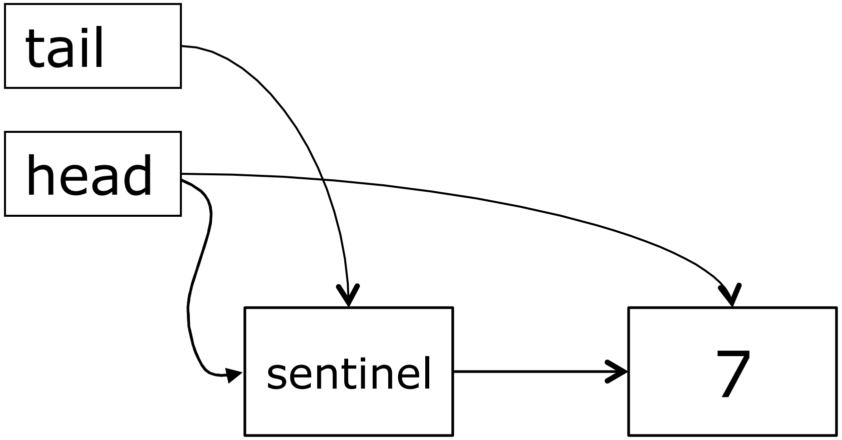
A: update a.next

B: dequeue()

B: update head

Now tail lags behind head, not good

So B: dequeue() must move tail before moving head




After Herlihy & Shavit p. 233

# Understanding Michael-Scott queue

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

```
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))
          return result;                 D13
      }
    }
  }
}
```

```
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       E9
      if (next == null)  {
        if (last.next.compareAndSet(next, node)) {
          tail.compareAndSet(last, node);
          return;
        }
      } else
        tail.compareAndSet(last, next);
    }
  }
}
```

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

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

```
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);
  }
}
```

Must be CAS'able

One AR per Node

Q 2

```
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`

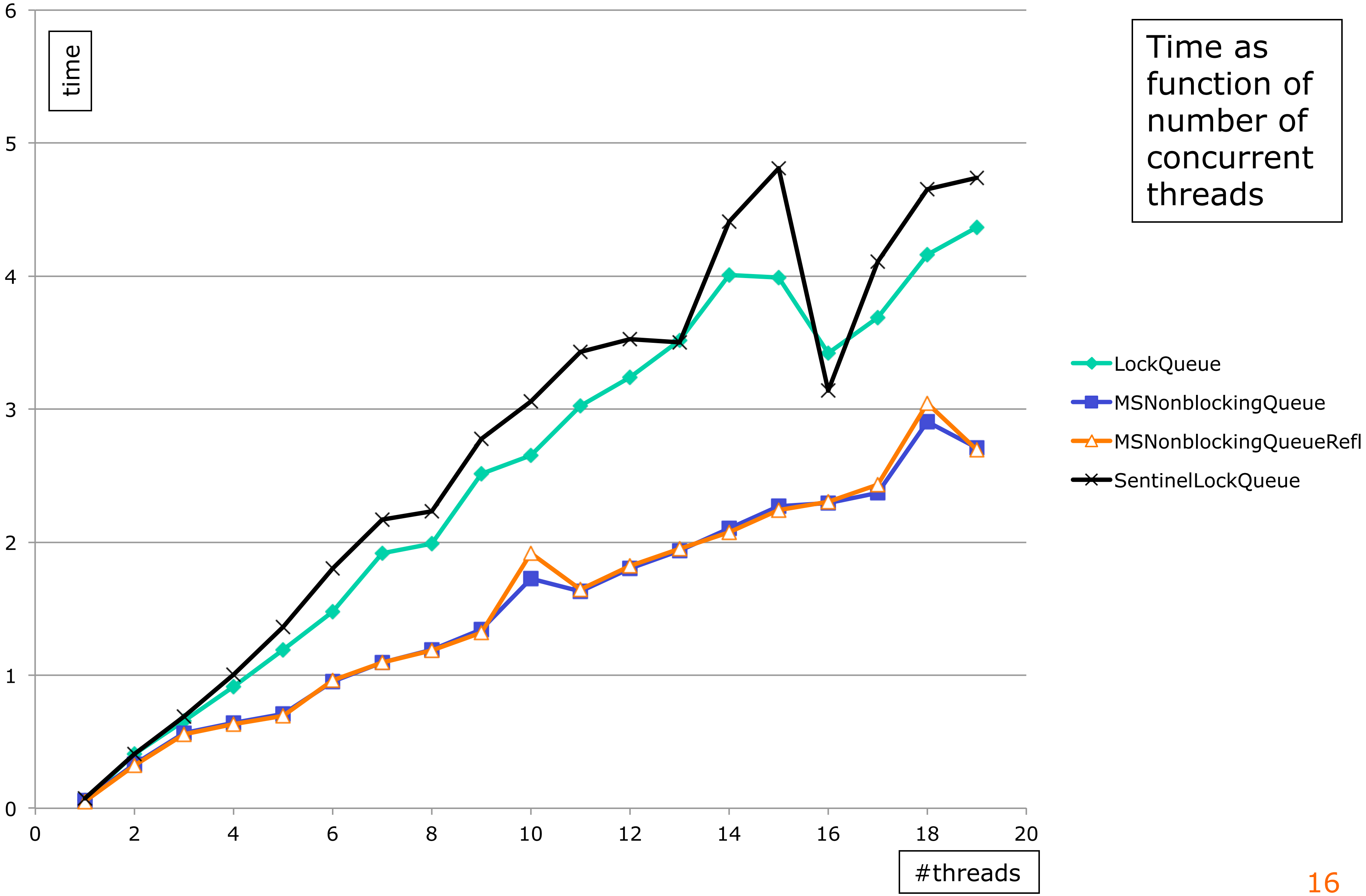
```
public void enqueue(T item) { // at tail
  Node<T> node = new Node<T>(item, null);
  while (true) {
    Node<T> last = tail.get(), next = last.next;
    if (last == tail.get()) {
      if (next == null)  {
        if (nextUpdater.compareAndSet(last, next, node)) {
          tail.compareAndSet(last,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

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

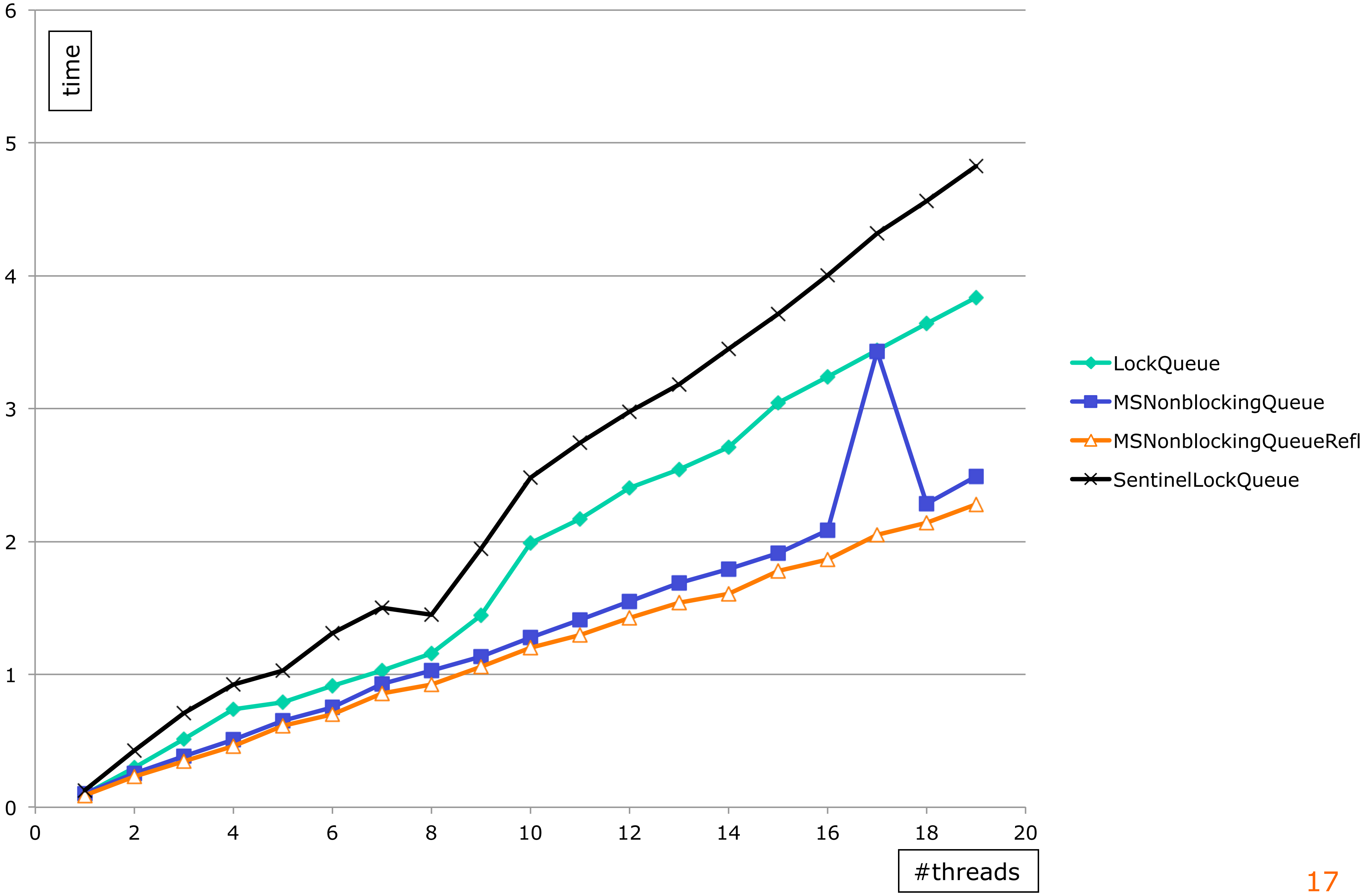
# Queue benchmarks

- 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

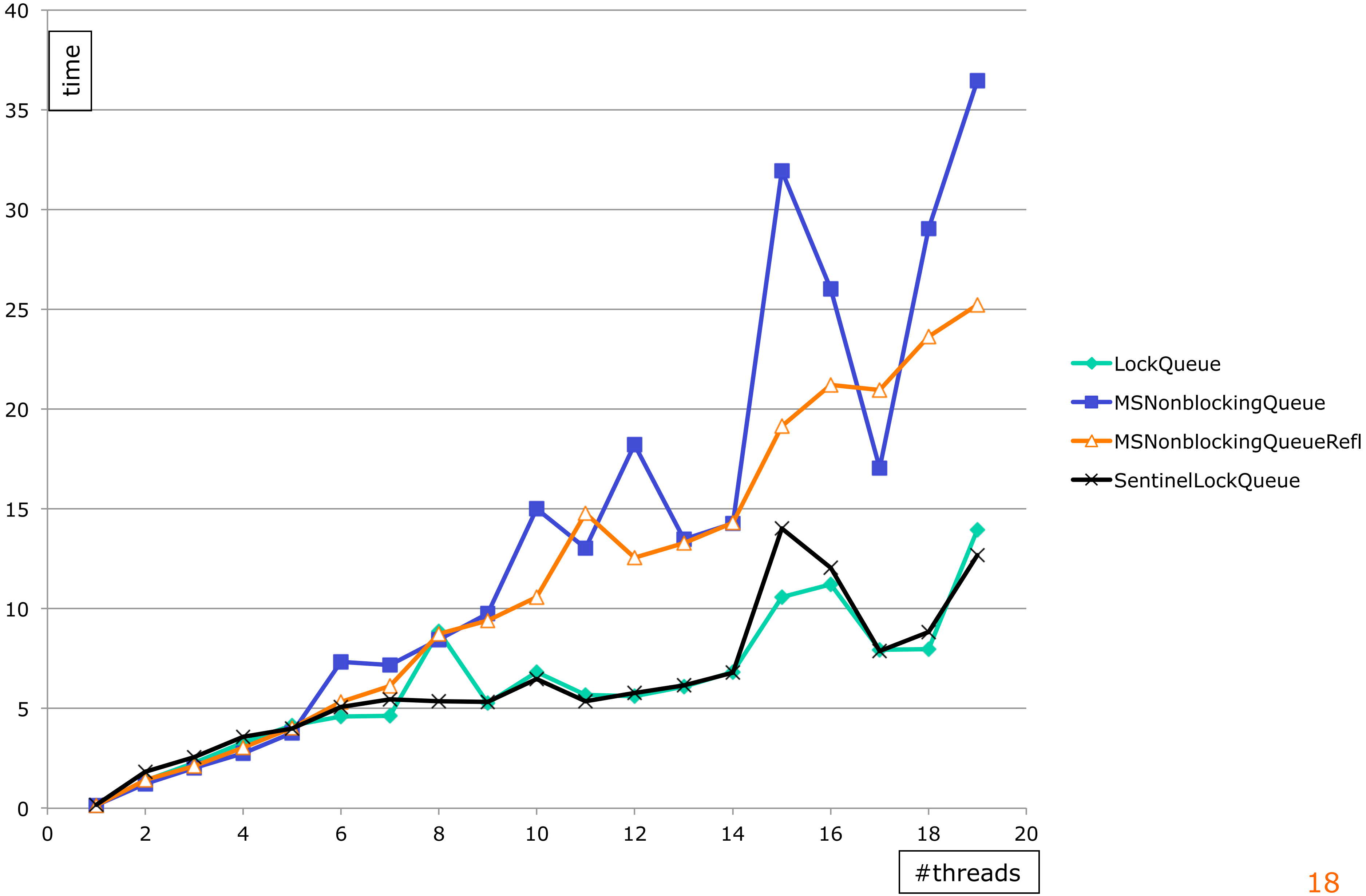
# Java 1.7, Xeon W3505, 2 cores



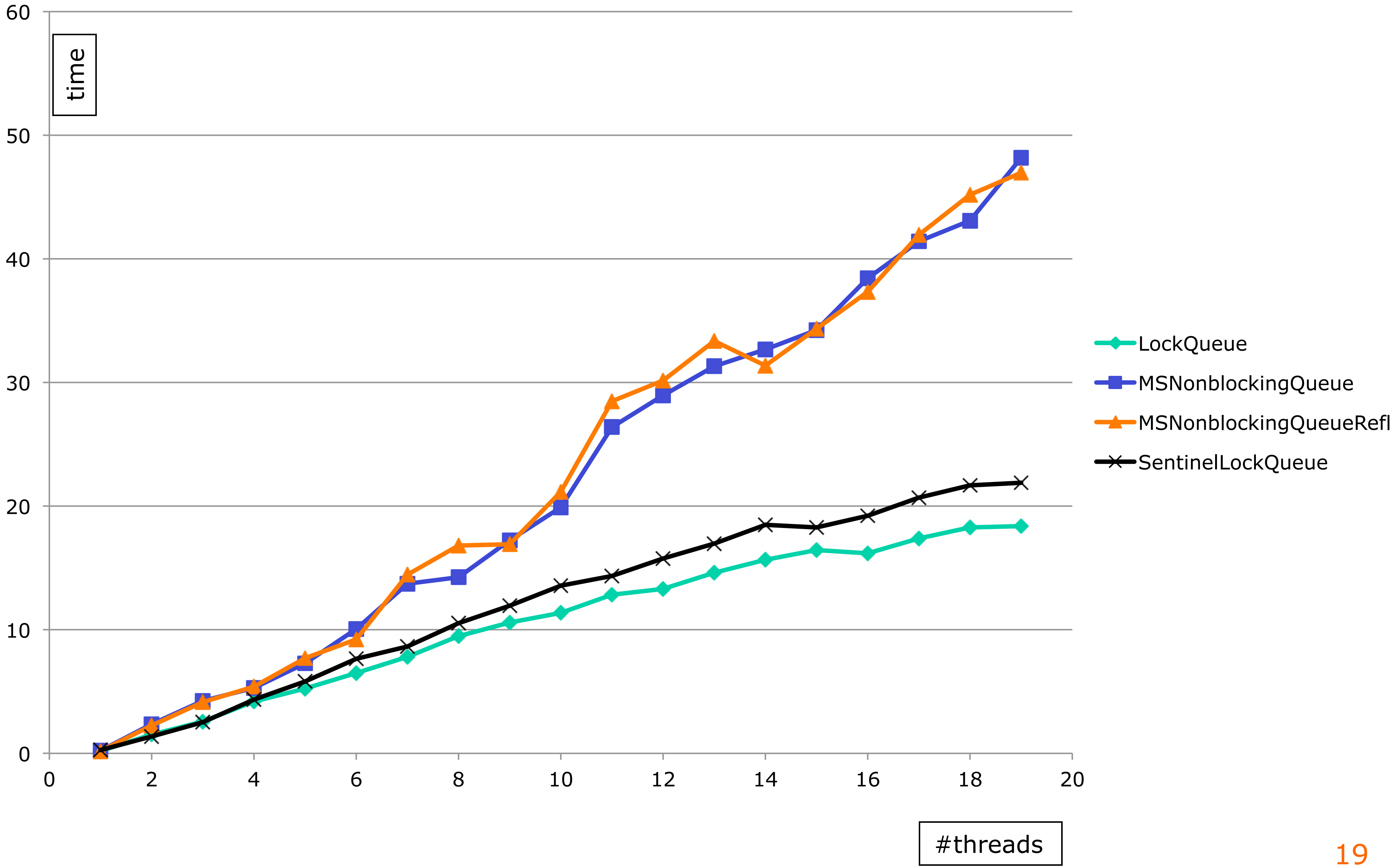
# Java 1.6, Core 2 Duo, 2 cores



# Java 1.7, Xeon E5320, 4x2 cores



# Java 1.7, AMD Opteron, 32 cores



# Plan for today

- Michael and Scott unbounded queue 1996
- **Progress concepts**
  - **Wait-free, lock-free, obstruction-free**
- Work-stealing dequeues
  - Chase-Lev dequeue 2005
- Union-find data structure

# Progress concepts

- *Non-blocking*: A call by thread A cannot prevent a call by 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

Not same as lock-less

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

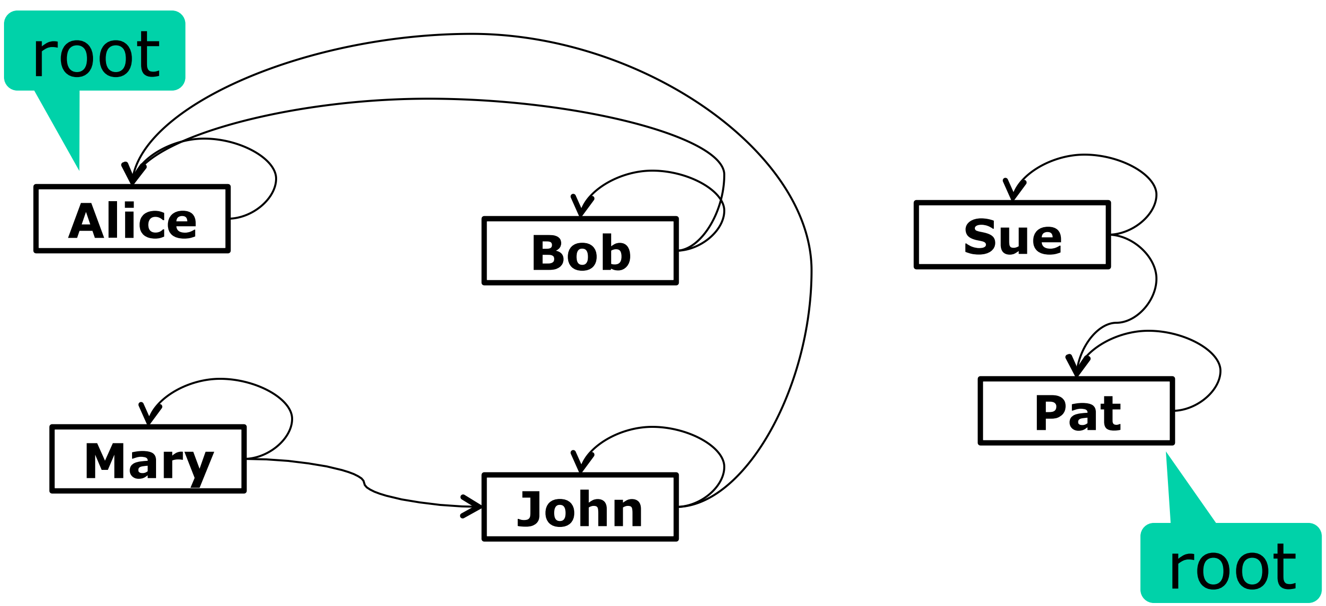
# Plan for today

- Michael and Scott unbounded queue 1996
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- **Union-find data structure**
- Work-stealing dequeues
  - Chase-Lev dequeue 2005

# 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



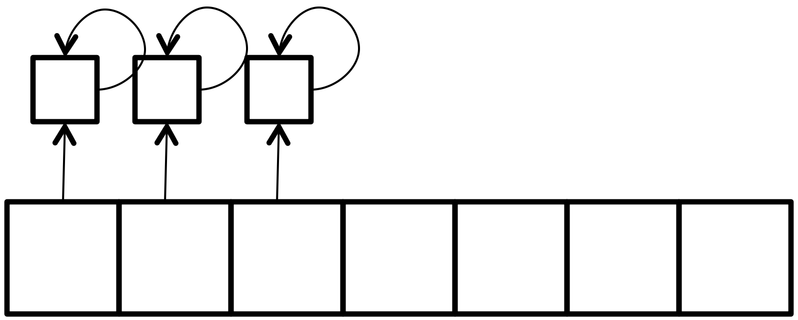

Sue is Pat's sister
Alice is Bob's sister
Mary is John's mother
Mary is Bob's mother

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

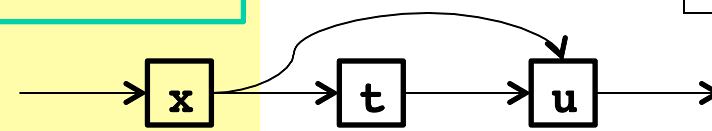
```
class CoarseUnionFind implements UnionFind {
  private final Node[] nodes;
  public synchronized int find(int x) {
    while (nodes[x].next != x) {
      final int t = nodes[x].next, u = nodes[t].next;
      nodes[x].next = u;
      x = u;
    }
    return x;
  }
  public synchronized void union(int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

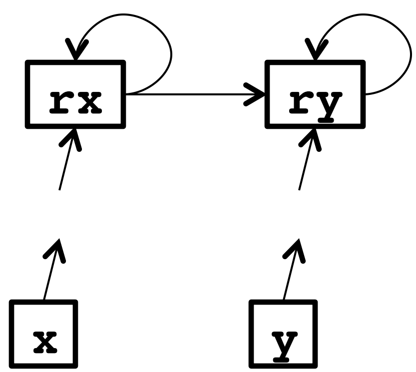
Path halving

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



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

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);
}
```

TestUnionFind.java

# 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

# Plan for today

- Michael and Scott unbounded queue 1996
- Progress concepts
  - Wait-free, lock-free, obstruction-free
- Union-find data structure
- **Work-stealing dequeues**
  - **Chase-Lev dequeue 2005**

# Perspective: Work-stealing dequeues

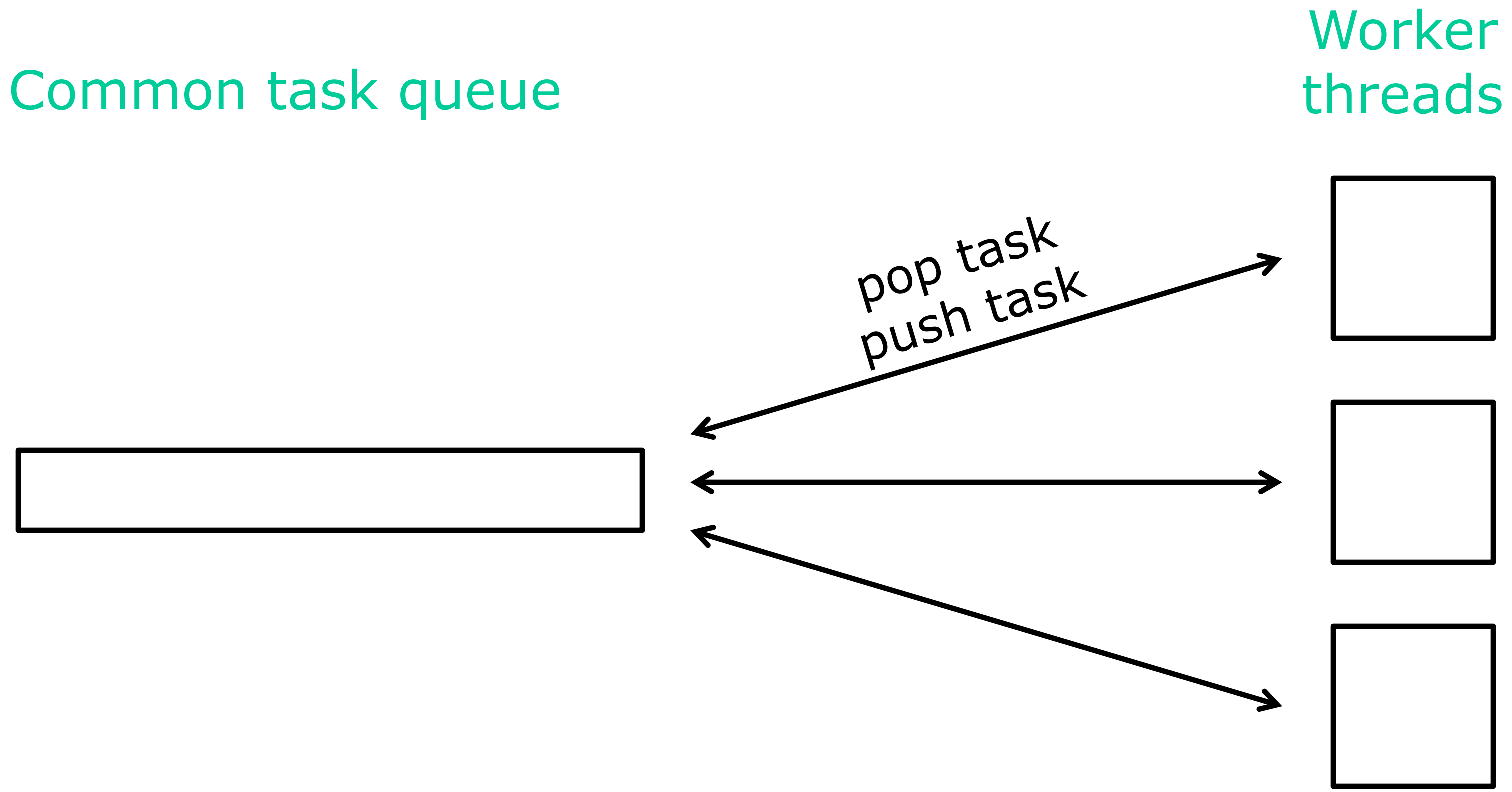
- Double-ended concurrent queues
- Used to implement
  - Java 7's Fork-Join framework, and Akka (wk 13-14)
  - Java 8's newWorkStealingPool executor
  - .NET 4.0 Task Parallel Library
- Chase and Lev: *Dynamic circular work-stealing queue*, SPAA 2005
- Michael, Vechev, Saraswat: *Idem-potent work stealing*,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

PCPP exam Jan 2015

Java 8 source

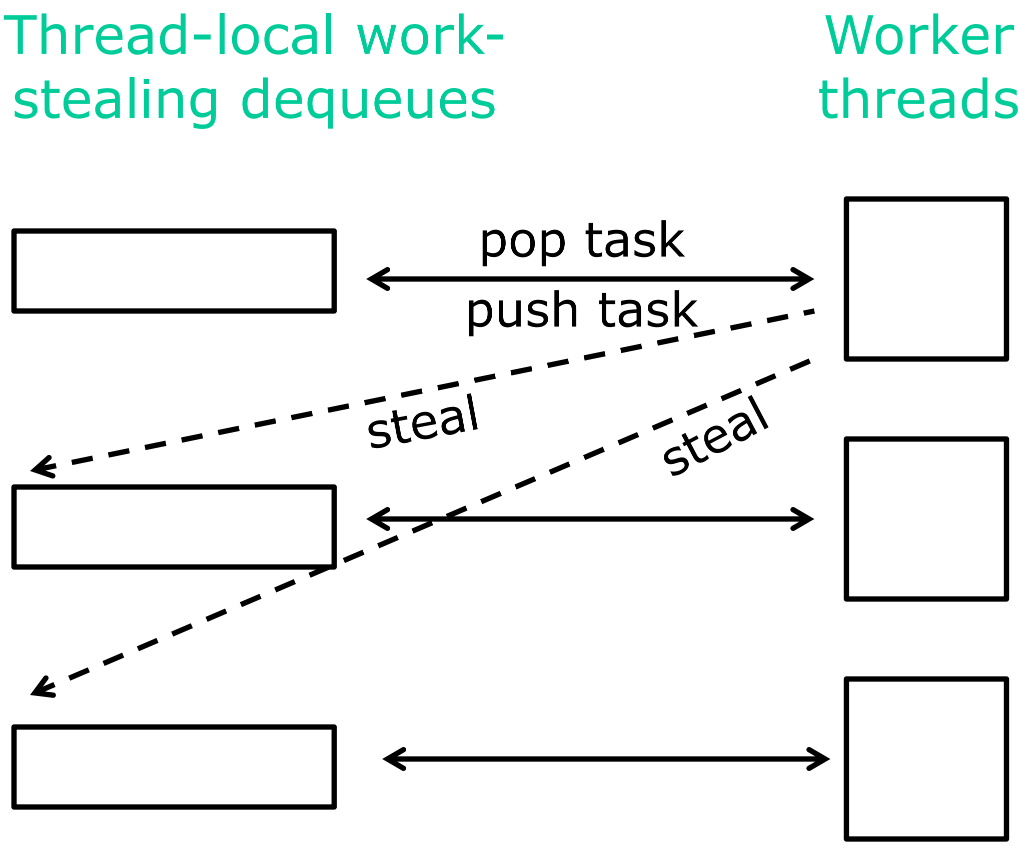
.NET TPL

# A worker/task framework




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

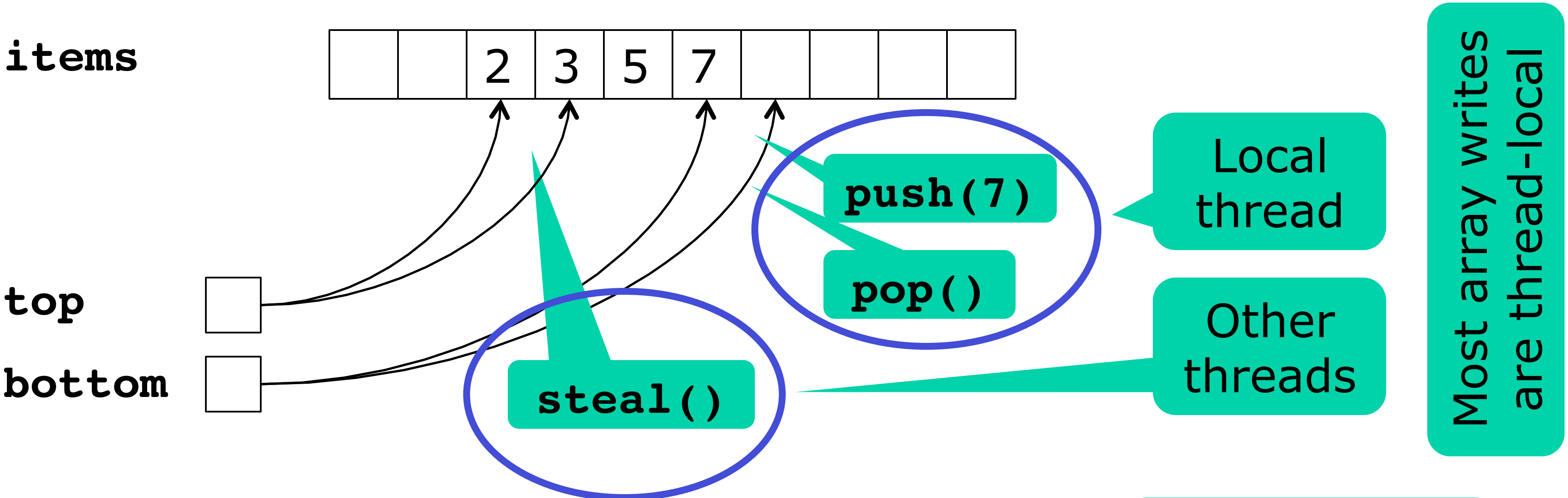
# Better worker/task framework




```
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**

# Chase-Lev workstealing queue (2005)




```
class ChaseLevDeque<T> {
  final T[] items;
  volatile long bottom = 0;
  final AtomicLong top = new AtomicLong();
  ...
}
```

Fixed size, for simplicity

Only the local thread writes

- **push** and **pop** at bottom: stack for local thread
- **steal** at top: queue for other threads

# Chase-Lev push at bottom

```
public void push(T item) {
  final long b = bottom, t = top.get(), size = b - t;
  if (size == items.length)
    throw new RuntimeException("queue overflow");
  items[index(b, items.length)] = item;
  bottom = b+1;
}
```

TestChaseLevQueue.java

- This is thread-safe, even without locks or CAS
  - Only one thread calls **`push`**
  - So only one thread *updates* the **`bottom`** field
  - Other threads *read* it, so it must be volatile

# Chase-Lev steal at top

```
public T steal() {
  final long t = top.get(), b = bottom, size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

Empty before call

Somebody else stole top item

TestChaseLevQueue.java

- Several threads may call **steal**
  - And try to increment **top**, hence an AtomicLong
  - So **steal** may fail (with null) due to interference
    - even if queue is non-empty
  - OK because callers keep stealing until success

# Chase-Lev pop at bottom

```
public T pop() {
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(), afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
        result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

Empty before call

Non-empty after call

Became empty

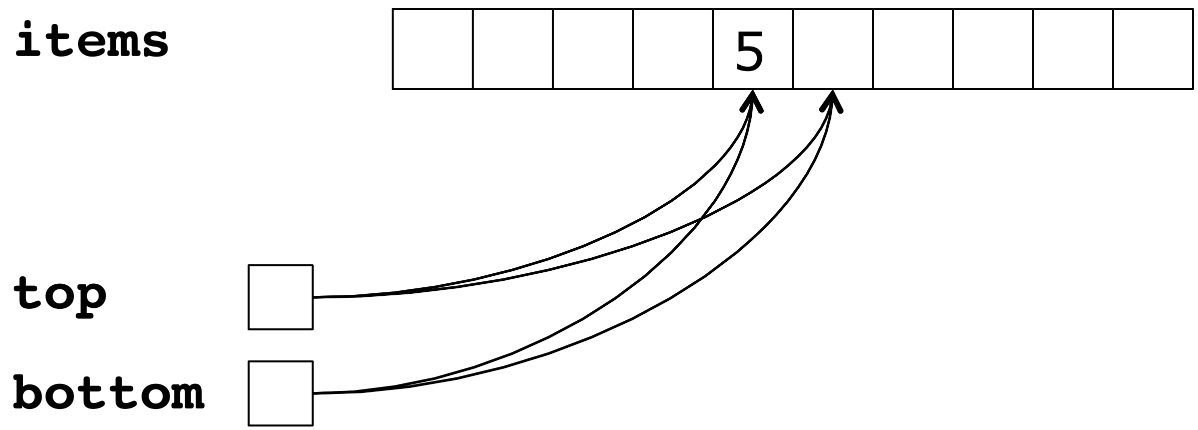
… so write **top** then set **bottom**

Oops, somebody stole last item

TestChaseLevQueue.java

# Why does pop update top?

- If **pop** takes the last item, it may clash with a concurrent **steal** operation
  - Because then **size == 0** and so **bottom == top**




- Hence **pop** must
  - check **top** is unchanged (nobody stole item yet)
  - if so, update **top** so stealers know item is taken
  - both done by **top.compareAndSet(t, t+1)**
  - no ABA problem because **top** always increases

# Linearization points

- When does **`steal`** take effect?
- When does **`push`** take effect?
- When does **`pop`** take effect?

# This week

- Reading
  - Michael & Scott 1996: *Simple, fast, and practical non-blocking and blocking concurrent queue ...*
  - Chase & Lev 2005: *Dynamic circular work-stealing deque*, sections 1, 2, 5
- Exercises
  - Test and experiment with the lock-free Michael & Scott queue
  - Test and experiment with the Chase-Lev work-stealing dequeue
- Read before next week – Claus lectures!
  - Armstrong, Virding, Williams: *Concurrent programming in Erlang*, chapters 1, 2, 5, 11.1

# Course evaluation

- General satisfaction with course, teachers, teaching assistants, exercises, ...

- However, contents overlaps somewhat with ITU BSc Software Development program
- Possible actions, fall 2017
  - Compress the Threads & Locks stuff even more
  - Spend more time (> 5 weeks) on
    - transactional memory (week 9)?
    - lock-free data structures (week 10-11)?
    - message passing and actors (week 12-13)?
    - other languages than Java (week 14) – but which ones?

# Numerical results (n=40) 2016

| **Question (6 = agree completely, 1 = disagree completely)** | **average** |
|---|---|
| Overall: I am happy about this course | 5.12 |
| I see a close correlation between the course topics and the exam requirements | 5.54 |
| I sense a close correlation between the exam requirements and the exam form | 5.41 |
| I think the course is relevant for my future job profile | 5.08 |
| My time consumption for this course is too high [...] | 3.63 |
| I am satisfied with my effort on this course | 4.85 |

# Numerical results (n=38) 2015

| **Question (6 = agree completely, 1 = disagree completely)** | **average** |
|---|---|
| Overall: I am happy about this course | 5.29 |
| I see a close correlation between the course topics and the exam requirements | 5.47 |
| I sense a close correlation between the exam requirements and the exam form | 5.50 |
| I think the course is relevant for my future job profile | 5.16 |
| My time consumption for this course is too high [...] | 3.63 |
| I am satisfied with my effort on this course | 4.71 |

# Numerical results (n=32) 2014

| Question (6 = agree completely, 1 = disagree completely) | average |
|---|---|
| Overall: I am happy about this course | 5.06 |
| I see a close correlation between the course topics and the exam requirements | 5.58 |
| I sense a close correlation between the exam requirements and the exam form | 5.61 |
| I think the course is relevant for my future job profile | 5.34 |
| My time consumption for this course is too high [...] | 3.44 |
| I am satisfied with my effort on this course | 4.84 |