Practical Concurrent and Parallel Programming 11

Thomas Dybdahl Ahle
IT University of Copenhagen

Thursday 2019-11-7

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- Scalability: pessimistic locks vs optimistic CAS
- Treiber lock-free stack
- The ABA problem

- **Course evaluation this week!**

Compare-and-swap (CAS)

- **Atomic** check-then-set, IBM 1970, Intel 80486 ...
- Java AtomicReference<T>
  - **`var.compareAndSet(T oldVal, T newVal)`**
    If **`var`** holds **`oldVal`**, set it to **`newVal`** and return true
- .NET/CLI System.Threading.Interlocked
  - **`CompareExchange<T>(ref T var, T newVal, T oldVal)`**
    If **`var`** holds **`oldVal`**, set it to **`newVal`** and return old value
- Used in optimistic concurrency
  - Try to update; if it fails, maybe restart
- Similar to transactional memory (STM, week 9)
  - but only one variable at a time
  - and under programmer control, not automatic
  - hardware machine primitive, where STM is high-level

CAS versus mutual exclusion (locks)

- Optimistic versus pessimistic concurrency
- Pro CAS
  - Almost all modern hardware implements CAS
  - Modern CAS is quite fast
  - CAS is used to implement locks
  - A failed CAS, unlike failed lock acquisition, requires no context switch, see Java Precisely p. 81
  - Therefore fast when contention is low
- Con CAS
  - Restart may fail arbitrarily many times
  - Therefore slow when contention is high
  - CAS slow on some manycore machines (32 c AMD)

Pseudo-implementation of CAS

```
class MyAtomicInteger {
  private int value;// Visibility ensured by locking
  synchronized boolean compareAndSet(int oldValue, int newValue){
    if (this.value == oldValue) {
      this.value = newValue;
      return true;
    } else
      return false;
  }

  public synchronized int get() {
    return this.value;
  }
  ...
}
```

TestCasAtomicInteger.java

- Only to *illustrate* CAS semantics
  - In reality **`synchronized`** is implemented by CAS
  - Not the other way around

AtomicInteger operations via CAS

```
public int addAndGet(int delta) {
  int oldValue, newValue;
  do {
    oldValue = get();
    newValue = oldValue + delta;
  } while (!compareAndSet(oldValue, newValue));
  return newValue;
}
public int getAndSet(int newValue) {
  int oldValue;
  do {
    oldValue = get();
  } while (!compareAndSet(oldValue, newValue));
  return oldValue;
}
```

TestCasAtomicInteger.java

- Optimistic concurrency approach
  - read **oldValue** from variable without locking
  - do computation, giving **newValue**
  - update variable if **oldValue** still valid

CAS and multivariable invariants: Unsafe number range [lower,upper]

```
public class NumberRange {
  // INVARIANT: lower <= upper
  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public void setLower(int i) {
    if (i > upper.get())
      throw new IllegalArgumentException("can't set lower");
    lower.set(i);
  }

  public void setUpper(int i) {
    if (i < lower.get())
      throw new IllegalArgumentException("can't set upper");
    upper.set(i);
  }
}
```

Goetz p. 67

Non-atomic test-then-set, may break *invariant*

Bad

Immutable integer pairs

- Use same technique as for factor cache (wk 2)
  - Make *immutable* pair of fields
  - Atomic assignment of reference to immutable pair
- Here, immutable pair of lower & upper bound:

```
private class IntPair {
  // INVARIANT: lower <= upper
  final int lower, upper;

  public IntPair(int lower, int upper) {
    this.lower = lower;
    this.upper = upper;
  }
}
```

Immutable, and safely publishable

Goetz p. 326

Using CAS to set the pair reference

```
public class CasNumberRange {
  private final AtomicReference<IntPair> values
    = new AtomicReference<IntPair>(new IntPair(0, 0));

  public int getLower() { return values.get().lower; }

  public void setLower(int i) {
    while (true) {
      IntPair oldv = values.get();
      if (i > oldv.upper)
        throw new IllegalArgumentException("Can't set lower");
      IntPair newv = new IntPair(i, oldv.upper);
      if (values.compareAndSet(oldv, newv))
        return;
    }
  }
```

Set if nobody else changed it

Goetz p. 326

- Atomic replacement of one pair by another
  - But may create many pairs before success ...
  - (And loop should be written using **do-while**)

CAS has visibility effects

- Java's AtomicReference.compareAndSet etc have the same visibility effects as **`volatile`**: "The memory effects for accesses and updates of atomics generally follow the rules for volatiles" (java.util.concurrent.atomic package documentation)

- Also in C#/.NET/CLI, Ecma-335, §I.12.6.5: "... atomic operations in the System.Threading.Interlocked class ... perform implicit acquire/release operations"

CAS in Java versus .NET

- .NET has static CAS methods in Interlocked
  - One can CAS to any variable or array element, good
  - But can easily forget to use CAS for update, bad
- Java's AtomicReference<T> seems safer
  - Because *must* access the field through that class
- But, for efficiency, Java allows standard field access through AtomicReferenceFieldUpdater
  - Uses reflection, see next week
  - This is at least as bad as the .NET design
  - And gives poor tool support: IDE, refactoring, ...

Why compare-and-swap (CAS)?

- *Consensus number* CN of a read-modify-write operation: the maximum number of parallel processes for which it can solve *consensus*, ie. make them agree on the value of a variable

- Atomically read a variable: CN = 1
- Atomically write a variable: CN = 1
- Test-and-set: atomically write a variable and return its old value: CN = 2
- Compare-and-swap: atomically check that variable has value oldVal and if so set it to newVal, returning true; else false: CN = ∞

Herlihy: Wait-free synchronization, 1991

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- **How to implement a lock using CAS**
- Scalability: pessimistic locks vs optimistic CAS
- Treiber lock-free stack
- The ABA problem

How to implement a lock using CAS

- Let’s make a lock class in four steps:
- A: Simple TryLock
  - non-blocking tryLock and unlock, once per thread
- B: Reentrant TryLock
  - non-blocking tryLock and unlock, multiple times
- C: Simple Lock
  - blocking lock and unlock, once per thread
- D: Reentrant Lock = j.u.c.locks.ReentrantLock
  - blocking lock and unlock, multiple times per thread

Lock A

Simple TryLock, no blocking

TestCasLocks.java

```
class SimpleTryLock {
  private final AtomicReference<Thread> holder
    = new AtomicReference<Thread>();
  public boolean tryLock() {
    final Thread current = Thread.currentThread();
    return holder.compareAndSet(null, current);
  }
  public void unlock() {
    final Thread current = Thread.currentThread();
    if (!holder.compareAndSet(current, null))
      throw new RuntimeException("Not lock holder");
  }
}
```

Try to take unheld lock

Release, if holder

- If lock is free, **`holder`** is **`null`**
  - Thread can take lock only if **`holder`** is **`null`**
- If lock is held, **`holder`** is the holding thread
  - Only the holding thread can unlock

A philosopher using SimpleTryLock

```
while (true) {
  int left = place, right = (place+1) % forks.length;
  if (forks[left].tryLock()) {
    try {
      if (forks[right].tryLock()) {
        try {
          System.out.print(place + " ");     // Eat
        } finally { forks[right].unlock(); }
      }
    } finally { forks[left].unlock(); }
  }
  try { Thread.sleep(10); }                  // Think
  catch (InterruptedException exn) { }
}
```

A fork is a SimpleTryLock

TestCasLocks.java

- Never deadlocks, may livelock
- Must unlock inside **`finally`**, else an exception may cause the thread to never release lock

Lock B

Reentrant TryLock, no blocking

```
class ReentrantTryLock {
  private final AtomicReference<Thread> holder = new Atomic...;
  private volatile int holdCount = 0;  // valid if holder!=null
  public boolean tryLock() {
    final Thread current = Thread.currentThread();
    if (holder.get() == current) {
      holdCount++;
      return true;
    } else if (holder.compareAndSet(null, current)) {
      holdCount = 1;
      return true;
    }
    return false;
  }
  public void unlock() {
    final Thread current = Thread.currentThread();
    if (holder.get() == current) {
      holdCount--;
      if (holdCount == 0)
        holder.compareAndSet(current, null))
      return;
    }
    throw new RuntimeException("Not lock holder");
  }
}
```

Already held by current thread

Unheld and we got it

Held by other

We hold it, reduce count

If count is 0, release

TestCasLocks.java

Lock C

Simple Lock, with blocking

```
class SimpleLock {
  private final AtomicReference<Thread> holder = new Atomic...;
  final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

  public void lock() {
    final Thread current = Thread.currentThread();
    waiters.add(current);
    while (waiters.peek() != current
      || !holder.compareAndSet(null, current))
    {
      LockSupport.park(this);
    }
    waiters.remove();
  }

  public void unlock() {
    final Thread current = Thread.currentThread();
    if (holder.compareAndSet(current, null))
      LockSupport.unpark(waiters.peek());
    else
      throw new RuntimeException("Not lock holder");
  }
}
```

Enter queue waiting for lock

If first, & lock free, take it ...

...else park

Got lock, leave queue

Unpark first parked thread

TestCasLock...

Based on example in java.util.concurrent.LockSupport documentation

Parking a thread

- Static methods in j.u.c.locks.LockSupport:
  - **`park()`**, deschedule current thread until permit becomes available; do nothing if already available
  - **`unpark(thread)`**, makes permit available for **`thread`**, allowing it to be scheduled again
- A thread can call **`park`** to wait for a resource without consuming any CPU time
- Another thread can **`unpark`** it when the resource appears to be available again

- Similar to **`wait/notifyAll`**, but those work only for intrinsic locks

Taking care of thread interrupts

- Parking will *block* the thread
  - may be interrupted by **`t.interrupt()`** while parked
  - should preserve interrupted status till unparked

TestCasLocks.java

```
class SimpleLock {
  ...
 public void lock() {
    final Thread current = Thread.currentThread();
    boolean wasInterrupted = false;
    waiters.add(current);
    while (waiters.peek() != current
        || !holder.compareAndSet(null, current)) {
      LockSupport.park(this);
      if (Thread.interrupted())
        wasInterrupted = true;
    }
    waiters.remove();
    if (wasInterrupted)
      current.interrupt();
  }
}
```

If interrupted while parked ...

... note that & clear interrupt

... & set interrupt when unparked

Based on example in java.util.concurrent.LockSupport documentation

Reentrant Lock, with blocking

```
class MyReentrantLock {
  private final AtomicReference<Thread> holder = new AtomicRef...;
  final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();
  private volatile int holdCount = 0;   // Valid if holder!=null
  public void lock() {
    final Thread current = Thread.currentThread();
    if (holder.get() == current)
      holdCount++;
    else {
      waiters.add(current);
      while (waiters.peek() != current
          || !holder.compareAndSet(null, current)) {
        LockSupport.park(this);
      }
      holdCount = 1;
      waiters.remove();
    }
  }
  public void unlock() { ... }
}
```

Already held by current thread

Enter queue waiting for lock

If first, & lock free, take it ...

...else park

Got lock, leave queue

- A cross between ReentrantTryLock and SimpleLock: both **`holdCount`** and **`waiters`**

Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- **Scalability: locks vs optimistic CAS**
- Treiber lock-free stack
- The ABA problem

A CAS is machine instruction

- Java

```
ai.compareAndSet(65, y)
```

- Bytecode

```
bipush        65
invokevirtual AtomicInteger.compareAndSet
```

- x86 code

second

first

```
mov   $0x41,%eax
lock cmpxchg %esi,(%rbx)
```

TestCas.java

- Intel x86 Instruction Reference CMPXCHG:

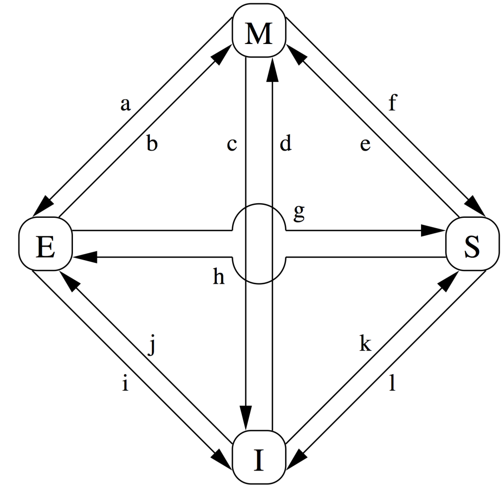
Compares the value in the EAX register with the first operand. If the two values are equal, the second operand is loaded into the first operand.
This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. [...] the first operand receives a write cycle without regard to the result of the comparison. The first operand is written back if the comparison fails; otherwise, the second operand is written into the first one.

Intel® 64 and IA-32 Architectures Software Developer's Manual, vol 2A p. 3-153

So CAS must be very fast?

- YES, it is fast
  - A successful CAS is faster than taking a lock
  - An unsuccessful CAS does not cause thread descheduling
- NO, it is slow
  - If many CPU cores try to CAS the same variable, then memory overhead may be very large
- Performancewise, like transactional memory
  - if mostly reads, then high concurrency
  - if many conflicting writes, then many restarts

Week 8 flashback: MESI cache coherence protocol



A write in a non-exclusive state requires acknowledge ack* from *all other* cores

CAS: many messages when other cores write same variable

		Cause	I send	I rec	
M	a	(Send update to RAM)	writeback	-	
E	b	Write	-	-	
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack

Scalability of locks and CAS: Pseudorandom number generation

```
class LockingRandom implements MyRandom {
  private long seed;
  public synchronized int nextInt() {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> 16);
  }
}
```

Lock-based

```
class CasRandom implements MyRandom {
  private final AtomicLong seed;
  public int nextInt() {
    long oldSeed, newSeed;
    do {
      oldSeed = seed.get();
      newSeed = (oldSeed * 0x5DEECE66DL + 0xBL) & ((1L << 48)-1);
    } while (!seed.compareAndSet(oldSeed, newSeed));
    return (int)(newSeed >>> 16);
  }
}
```

TestPseudoRandom.java

A la Goetz p. 327

CAS-based

- (Q: Could one use **volatile** instead?)

Thread-locality is (more) important for scalability

```
class TLLockingRandom implements MyRandom {
  private final ThreadLocal<MyRandom> myRandomGenerator;
  public TLLockingRandom(final long seed) {
    this.myRandomGenerator =
      new ThreadLocal<MyRandom>() {
        public MyRandom initialValue() {
          return new LockingRandom(seed);
      }};
  }
  public int nextInt() {
    return myRandomGenerator.get().nextInt();
  }
}
```

Create this thread's generator

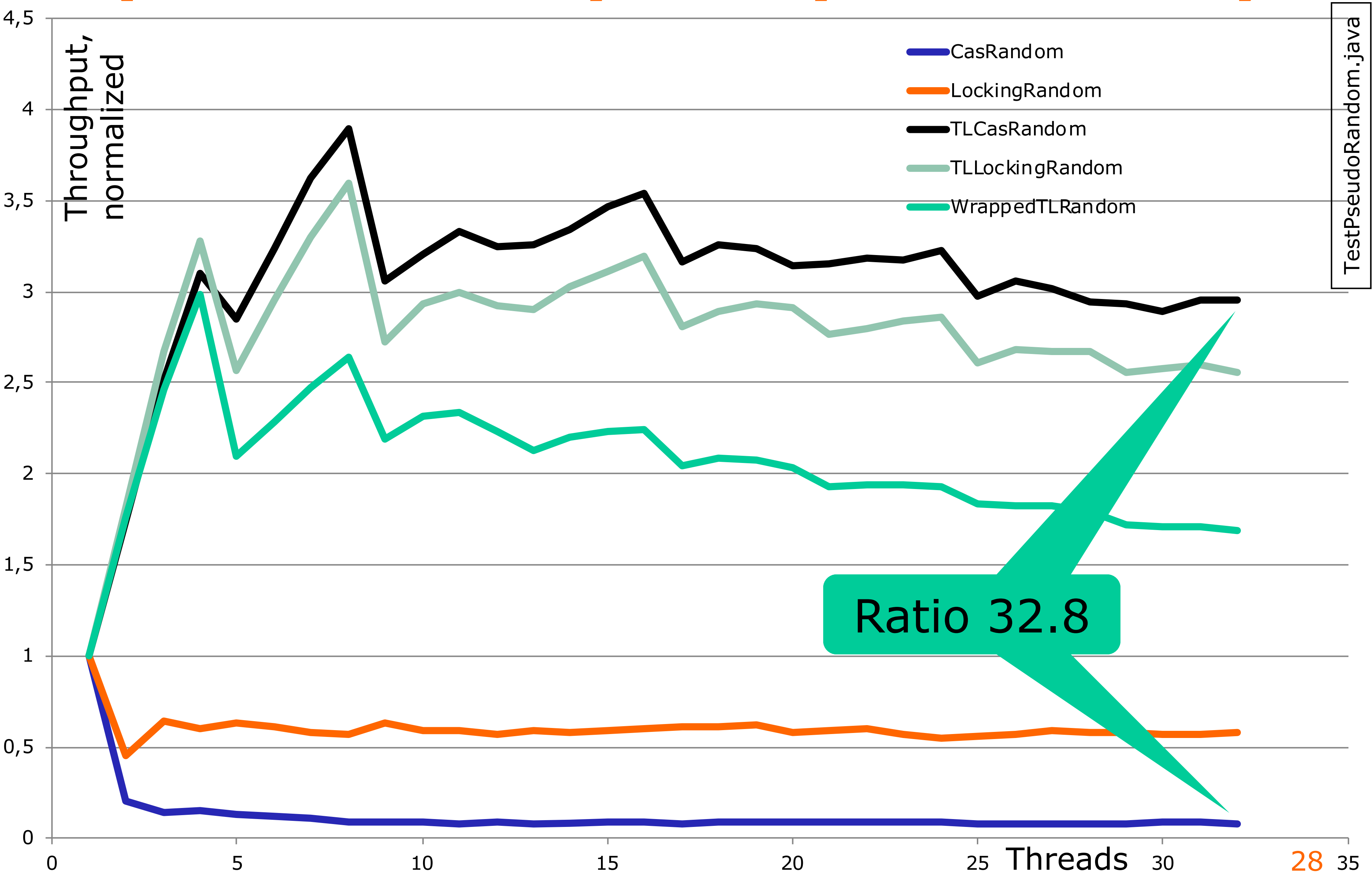
Get this thread's generator

TestPseudoRandom.java

- A LockingRandom instance for each thread
- A thread's first call to **`.get()`** causes a call to **`initialValue()`** to create the instance
- Never any access conflicts between threads

Goetz § 3.3.3

Random number generator scalability (unrealistically heavy contention)



Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- Scalability: pessimistic locks vs optimistic CAS
- **Treiber lock-free stack**
- The ABA problem

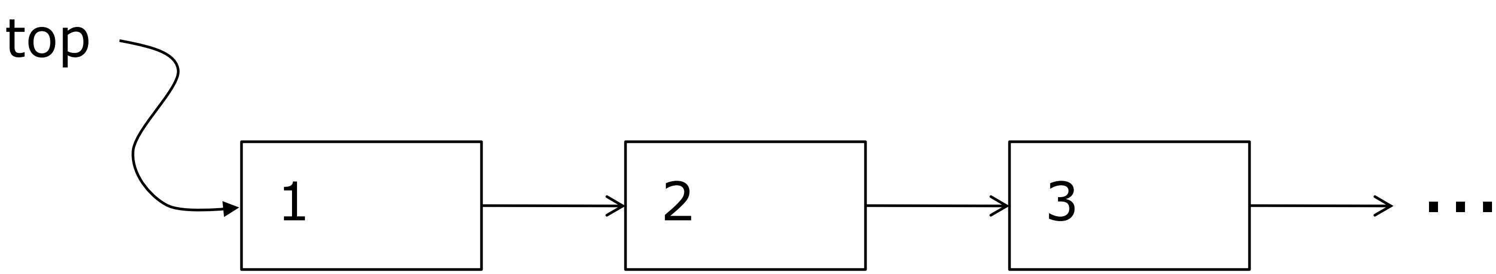
Treiber's lock-free stack (1986)

```
class ConcurrentStack <E> {
  private static class Node <E> {
    public final E item;
    public Node<E> next;

    public Node(E item) {
      this.item = item;
    }
  }

  AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();
  ...
}
```

Goetz Listing 15.6



Treiber's stack operations

```
public void push(E item) {
  Node<E> newHead = new Node<E>(item);
  Node<E> oldHead;
  do {
    oldHead = top.get();
    newHead.next = oldHead;
  } while (!top.compareAndSet(oldHead, newHead));
}
```

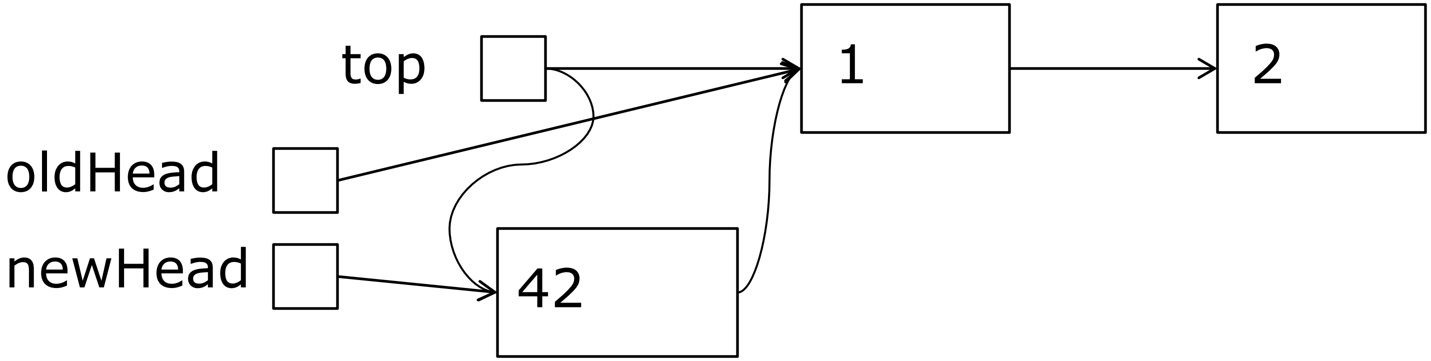
Set **top** to **new** if not changed

```
public E pop() {
  Node<E> oldHead, newHead;
  do {
    oldHead = top.get();
    if (oldHead == null)
      return null;
    newHead = oldHead.next;
  } while (!top.compareAndSet(oldHead, newHead));
  return oldHead.item;
}
```

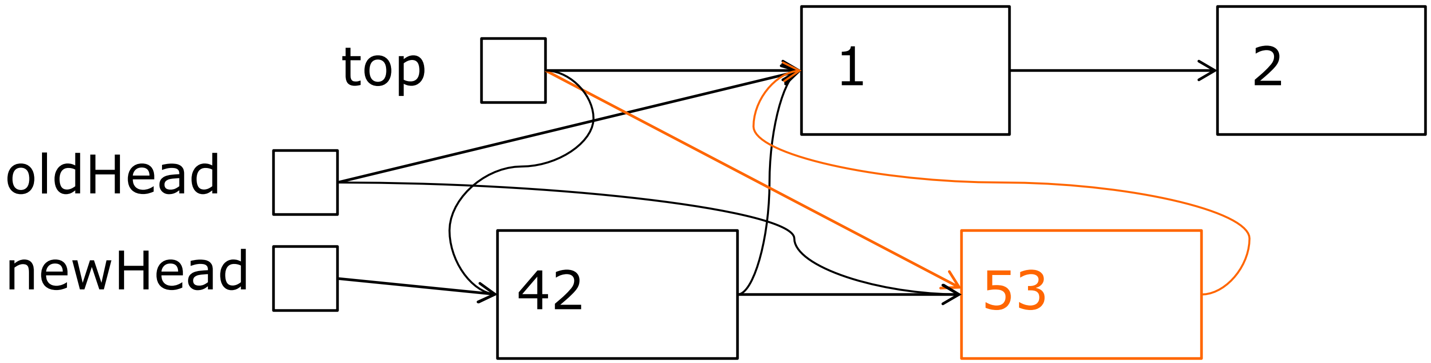
Set **top** to **next** if not changed

Treiber stack push(42)

Success on first try



Success on second try



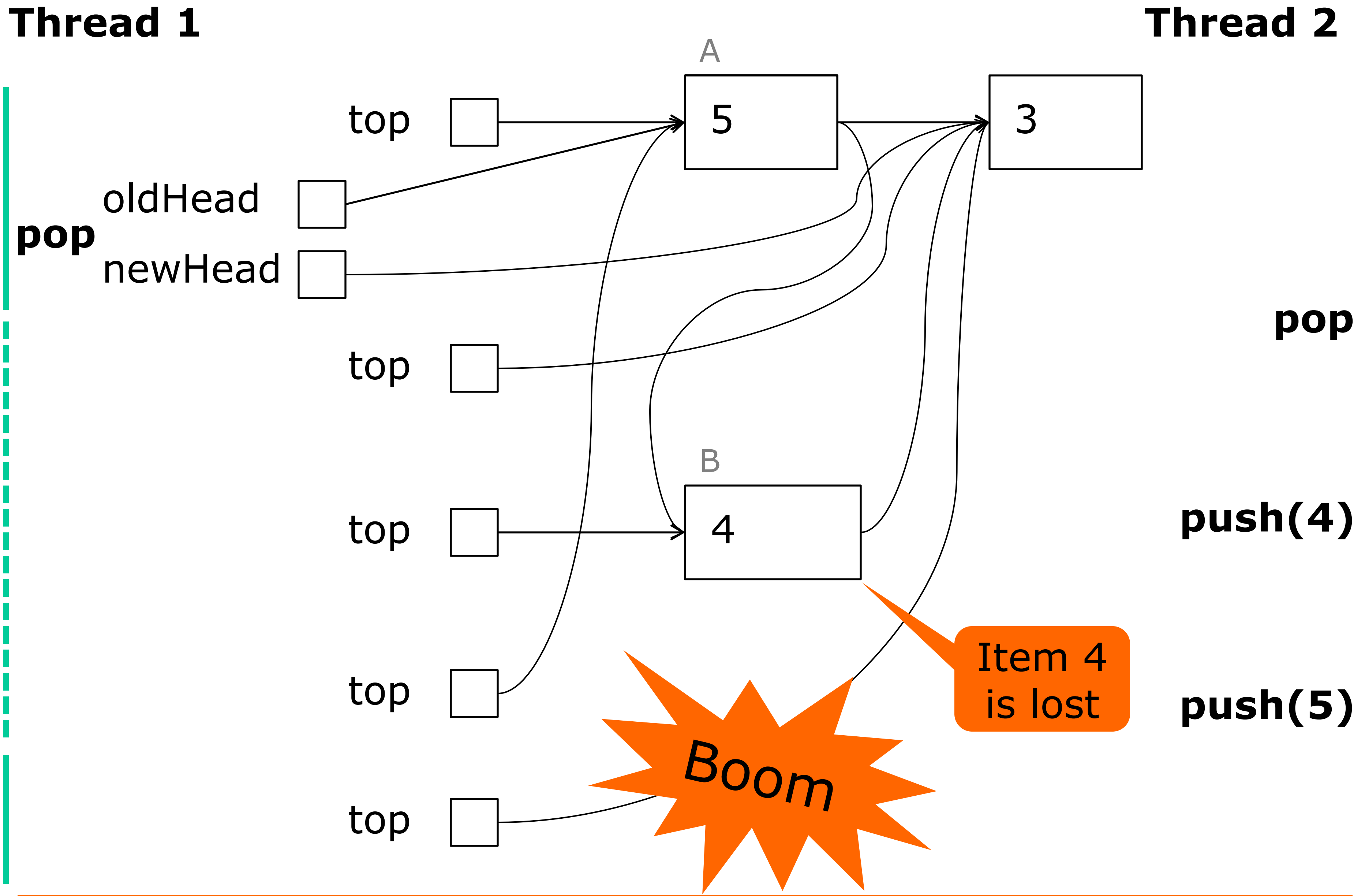
Plan for today

- Compare and swap (CAS) low-level atomicity
- Examples: AtomicInteger and NumberRange
- How to implement a lock using CAS
- Scalability: pessimistic locks vs optimistic CAS
- Treiber lock-free stack
- **The ABA problem**

The ABA problem

- CAS variable has value A, then B, then A
  - Hence variable changed, but CAS does not see it
- Eg AtomicInteger was A, then add +b, add –b
  - Not a problem in MyAtomicInteger
- Typically a problem with pointers in C, C++
  - Reference p points at a struct; then free(p); then malloc() returns p, but now a different struct ...
- Standard solution: make pair (p,i) of pointer and integer counter; probabilistically correct
- Rarely an ABA-problem in Java, C#
  - Automatic memory management, garbage collector
  - So objects are not reused while referred to

ABA in Treiber stack à la C



This week

- Reading
  - Goetz et al section 3.3.3 and chapter 15
- Exercises
  - Show that you can implement a concurrent Histogram and a ReadWriteLock using CAS