Practical Concurrent and Parallel Programming 11

Peter Sestoft
IT University of Copenhagen

Friday 2015-11-13*
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 feedback
- Bonus: Measuring memory hierarchy latency
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

- Optimistic concurrency
  - Try to update; if it fails, maybe restart

- Similar to transactional memory (STM, week 10)
  - 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. 67
  – 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

```java
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;
   }
   ...
}
```

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

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

```java
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;
}
```
CAS and multivariable invariants: Unsafe number range \([\text{lower}, \text{upper}]\)

```java
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:

```java
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;
        }
    }
}

• Atomic replacement of one pair by another
  – But may create many pairs before success ...
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 = ∞
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 feedback
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
Simple TryLock, no blocking

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");
    }
}

- 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

Try to take unheld lock
Release, if holder
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) { } 
}

• Very similar to Exercise 9.2.5
• Must unlock in **finally**, else an exception may cause the thread to never release lock
Reentrant TryLock, no blocking

class ReentrantTryLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    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");
    }
}
Simple Lock, with blocking

```java
class SimpleLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    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");
    }
}
```

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 resources

• 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

```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();
    }
    ...
}
```
class MyReentrantLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    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() { ... }
}

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 stack
• The ABA problem
• Progress concepts
  – Lock-free, wait-free, obstruction-free
• Course evaluation feedback
A CAS is machine instruction

- Java
  ```java
  ai.compareAndSet(65, y)
  ```

- Bytecode
  ```
  bipush 65
  invokevirtual AtomicInteger.compareAndSet
  ```

- x86 code
  ```
  mov $0x41,%eax
  lock cmpxchg %esi,(%rbx)
  ```

- 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 retries
Week 8 flashback: MESI cache coherence protocol

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

<table>
<thead>
<tr>
<th>Cause</th>
<th>I send</th>
<th>I receive</th>
<th>CAS: many messages when other cores write same variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>M a (Send update to RAM)</td>
<td>writeback</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>E b Write</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>M c Other wants to write</td>
<td>-</td>
<td>read inv</td>
<td></td>
</tr>
<tr>
<td>I d Atomic read-mod-write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td></td>
</tr>
<tr>
<td>S e Atomic read-mod-write</td>
<td>read inv</td>
<td>inv ack*</td>
<td></td>
</tr>
<tr>
<td>M f Other wants to read</td>
<td>-</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>E g Other wants to read</td>
<td>-</td>
<td>read</td>
<td></td>
</tr>
<tr>
<td>S h Will soon write</td>
<td>inv</td>
<td>inv ack*</td>
<td></td>
</tr>
<tr>
<td>E i Other wants atomic rw</td>
<td>-</td>
<td>read inv</td>
<td></td>
</tr>
<tr>
<td>I j Want to write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td></td>
</tr>
<tr>
<td>I k Want to read</td>
<td>read</td>
<td>read resp</td>
<td></td>
</tr>
<tr>
<td>S l Other wants to write</td>
<td>-</td>
<td>inv</td>
<td>inv ack 25</td>
</tr>
</tbody>
</table>
Scalability of locks and CAS: Pseudorandom number generation

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

```java
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);
    }
}
```

- (Could one use `volatile` instead?)
Thread-locality is (more) important for scalability

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

```java
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();
  }
}
```
Random number generator scalability (unrealistically heavy contention)

Throughput, normalized

Threads

Ratio 32.8

CasRandom
LockingRandom
TLCasRandom
TLLockingRandom
WrappedTLRandom

TestPseudoRandom.java
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
- Progress concepts
  - Lock-free, wait-free, obstruction-free
- Course evaluation feedback
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

```java
class Node<E> {
    Node<E> next;
    E item;
}

class Stack<E> {
    private Node<E> top;

    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));
    }

    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 new if not changed

Set top to next if not changed
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

Thread 1

pop

oldHead
newHead

top

Thread 2

pop

push(4)
push(5)

Item 4 is lost

Boom
Course evaluation

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

• However, contents overlap with ITU Software Development BSc program

• Possible actions, fall 2016
  – Compress the Threads & Locks stuff even more
  – Spend more time (> 5 weeks) on
    • transactional memory (week 10)?
    • lock-free data structures (week 11-12)?
    • message passing and actors (week 13-14)?
    • other languages than Java – if so, which ones?
# Numerical results (n=38) 2015

<table>
<thead>
<tr>
<th>Question (6 = agree completely, 1 = disagree completely)</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall: I am happy about this course</td>
<td>5.29</td>
</tr>
<tr>
<td>I see a close correlation between the course topics and the exam requirements</td>
<td>5.47</td>
</tr>
<tr>
<td>I sense a close correlation between the exam requirements and the exam form</td>
<td>5.50</td>
</tr>
<tr>
<td>I think the course is relevant for my future job profile</td>
<td>5.16</td>
</tr>
<tr>
<td>My time consumption for this course is too high [...]</td>
<td>3.63</td>
</tr>
<tr>
<td>I am satisfied with my effort on this course</td>
<td>4.71</td>
</tr>
</tbody>
</table>
### Numerical results (n=32) 2014

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

• Jumping through an array of indexes:

```java
int k = 0;
for (int j=0; j<arr.length; j++)
    k = arr[k];
```

arr.length = 16,777,216

• Constant number of “operations”
• But speed heavily depends on array contents
  – Sequential or random travel, length of cycle
  – Reveals speed of L1, L2, L3 caches and RAM
Speed depends on memory touched

i7 (Mac)

ns/iter

memory touched (KB), random access
Speed depends on memory touched

Q: How fast is a register access?

L1: 1.6 ns
L2: 3.6 ns
L3: 16 ns
RAM: 80 ns

i7 (Mac)

memory touched (KB), random access
Memory latency in three machines

- i7 (Mac)
- Xeon (mala)
- AMD (mtlab)

Memory touched (KB), random access
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
  – Goetz et al section 3.3.3 and chapter 15
  – Herlihy & Shavit section 3.8

• Exercises
  – Show that you can implement a concurrent Histogram and a ReadWriteLock using CAS

• Read before next week
  – 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