Lock Free Programming

- **Locking and CAS**
- **Atomics**
- **Non-Blocking Algorithms**
- **ABA Problem**

Much of the recent research on concurrent algorithms has focused on non-blocking algorithms.

Applications:
- Inside VM (thread scheduling)
- Inside VM (GC implementation)
- Implementation of locks
- Implementation of concurrent data structures

Locking

- **Disadvantages of Locks** ***synchronized*** ***java.util.concurrent.locks.Lock***
  - Under contention, performance suffers because of context-switch overhead
  - When a thread is waiting for a lock, it cannot do anything else
    - Variant: Lock.tryLock (acquires lock only if it is free)
  - If a thread which holds a lock is delayed, then no thread that needs the lock can make progress
    - Priority inversion:
      High-priority thread waits for a lock held by a low-priority thread
      => high-priority thread is downgraded to priority of low-priority thread
  - If a thread which holds a lock is permanently blocked (infinite loop, live lock, deadlock), then any thread waiting for that lock cannot make progress
  - Amdahl's law

Example: Synchronized Counter

```
public final class Counter {
   private long value = 0;

   public synchronized long getValue() {
      return value;
   }

   public synchronized long increment() {
      return ++value;
   }
}
```

- How can the use of locks be avoided?

Example: Synchronized Counter

```
public final class Counter {
   private volatile long value = 0;

   public long getValue() {
      return value;
   }

   public synchronized long increment() {
      return ++value;
   }
}
```

- But volatiles do not support read-modify-write sequences

=> *It would be nice to have something like volatile, providing atomic updates*

CAS: Compare and Set/Swap Instruction

- **CAS(memory_location, expected_old_value, new_value)**
  - Atomically compares the content of a memory location to a given value and, if they are the same, modifies the content of that memory location to a given new value
  - The result must indicate whether the substitution was performed
    - Boolean result (=> compare-and-set)
    - Returning the value read from the memory location (=> compare-and-swap)

```
long value;
boolean compareAndSet(long oldValue, long newValue) {
   if(value == oldValue) { value = newValue; return true; }
   else return false;
}
```

Atomic check and set

- **CPU Instructions**
  - CMPXCHG x86 / Itanium
  - LL/SC Alpha / PowerPC / MIPS / ARM (load-link/store-conditional)

CAS: Compare and Swap Instruction

```
package sun.misc;
public final class Unsafe {
  /**
   * Compares the value of the integer field at the specified
   * offset in the supplied object obj with the given expected
   * value, and atomically updates it if they match.
   * @param obj    the object containing the field to modify.
   * @param offset the offset of the integer field within obj.
   * @param expect the expected value of the field.
   * @param update the new value of the field if it equals expect.
   * @return true if the field was changed.
   */
  public final native boolean compareAndSwapLong(
         Object obj, long offset, long expect, long update);
  ...

}
```

Example: CAS Counter

```
public final class CASCounter {

   private volatile long value = 0;

   public long getValue() {
      return value;
   }

   public long increment() {
      while(true) {
         long current = getValue();
         long next = current + 1;
         if (compareAndSwap(current, next)) return next;
      }
   }

   ...
}
```

Example: CAS Counter

```
public final class CASCounter {
   ...
   private static final Unsafe unsafe = Unsafe.getUnsafe();
   private static final long valueOffset;

   static {
      try {
        valueOffset = unsafe.objectFieldOffset(
            CASCounter.class.getDeclaredField("value"));
      } catch (Exception ex) { throw new Error(ex); }
   }

   private boolean compareAndSwap(long expectedVal, long newVal) {
      return unsafe.compareAndSwapLong(this, valueOffset,
                                       expectedVal, newVal);
   }
}
```

Lock Free Programming

- Locking and CAS
- Atomics
- Non-Blocking Algorithms
- ABA Problem

Atomics: java.util.concurrent.atomic

- **Atomic Scalars and field updaters**
  - AtomicInteger
  - AtomicLong
  - AtomicReference<V>
  - AtomicBoolean
    - All support CAS + arithmetic operations for int/long
    - JMM: Atomics guarantee the same visibility behavior as volatile variables! => same happens-before rules
- **Atomic Arrays**
  - AtomicIntegerArray
  - AtomicLongArray
  - AtomicReferenceArray<V>
    - JMM: Atomic arrays provide volatile access semantics to the elements of the array, a feature not available for ordinary arrays

AtomicInteger

- **Methods provided by AtomicInteger**

```
class AtomicInteger extends Number {
   AtomicInteger()
   AtomicInteger(int initialValue)

   boolean compareAndSet(int expect, int update)

   int incrementAndGet()      int decrementAndGet()
   int getAndIncrement()      int getAndDecrement()
   int addAndGet(int delta)   int getAndAdd(int delta)
   int getAndSet(int newValue)

   int intValue()             double doubleValue()
   long longValue()           float floatValue()
   int get()                  void set(int newValue)
}
```

AtomicInteger

- **Example: Implementation of incrementAndGet**

```
private volatile int value;

public final int get() { return value; }
public final void set(int newValue) { value = newValue; }

public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Characteristic of optimistic algorithms

- **Pessimistic execution**
  - First acquire all locks and then perform the computation
- **Optimistic execution**
  - Some work is done speculatively and may have to be redone

```
while(true) {
  oldp = p.get();
  newp = ...
  if( p.compareAndSet(oldp, newp) ) return;
}
```

  - Thread-safety derived from
    - Atomicity of CAS operation
    - Visibility guarantees
  - Simple case: atomic change is limited to a single variable
    - Invariants across several variables are difficult to control

Example

```
public class NumberRange {
  // INVARIANT: lower <= upper
  private int lower, upper = 0;

  public void setLower(int newLower) {
    if (newLower > upper) throw new IllegalArgumentException();
    lower = i;
  }

  public void setUpper(int newUpper) {
    if (newUpper < lower) throw new IllegalArgumentException();
    upper = i;
  }

  public boolean contains(int i) {
    return i >= lower && i <= upper;
  }
}
```

Example: Lock-Free NumberRange

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

   public int getLower() { return lower.get(); }
   public void setLower(int newLower) {
      while (true) {
         int l = lower.get(), u = upper.get();
         if (newLower > u) throw new IllegalArgumentException();
         if (lower.compareAndSet(l, newLower)) return;
      }
   }
   // same for getUpper/setUpper

   public boolean contains(int x) {
      return lower.get() <= x && x <= upper.get();
} }
```

Example: Lock-Free NumberRange

```
public class NumberRange {
   private static class Pair {
      final int lower, upper; // lower <= upper
      Pair(int l, int u) { lower = l; upper = u; }
   }

   private final AtomicReference<Pair> values =
      new AtomicReference<>( new Pair(0,0) );

   public int getLower(){ return values.get().lower; }
   public void setLower(int newLower){
      while(true) {
         Pair oldp = values.get();
         if(newLower > oldp.upper)
                        throw new IllegalArgumentException();
         Pair newp = new Pair(newLower, oldp.upper);
         if( values.compareAndSet(oldp, newp) ) return;
      }}}
```

Lock Free Programming

- **Locking and CAS**
- **Atomics**
- **Non-blocking Algorithms**
  - Non-blocking Stack
  - Non-blocking Queue
- **ABA Problem**

Non-blocking Algorithms

- **Non-blocking**
  - An algorithm is called *non-blocking* if failure or suspension of any thread cannot cause failure or suspension of another thread
    - => no deadlock
    - => no priority inversion
    - => BUT live-lock & starvation still possible
- **Non-blocking guarantees**
  - Lock-free (guaranteed system-wide progress)
    - An algorithm is called *lock-free* if some thread always makes progress
    - => starvation possible, but system-wide throughput is guaranteed
  - Wait-free (guaranteed per-thread progress)
    - An algorithm is called *wait-free* if every thread makes progress in the face of arbitrary delay (or even failure) of other threads (difficult to achieve)
    - => no starvation, per-thread progress guaranteed

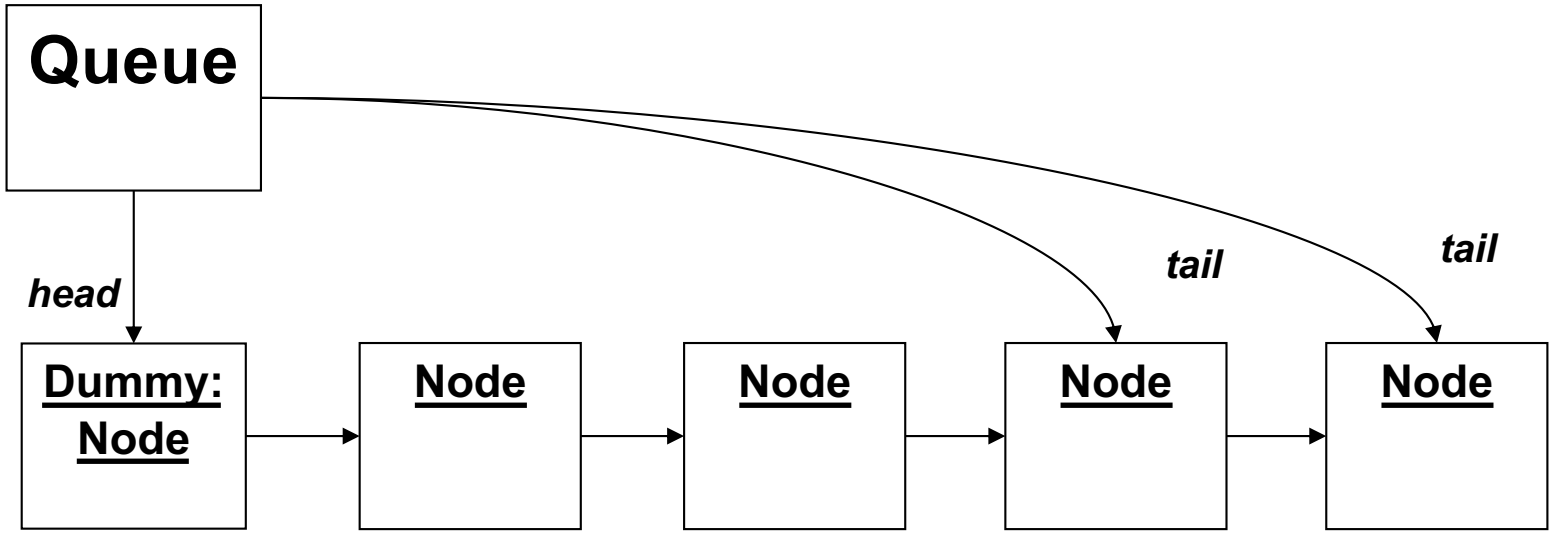
Non-blocking Stack

```
public class ConcurrentStack<E> {
   private static class Node<E> {
      public final E item;
      public Node<E> next;
      public Node(E item) { this.item = item; }
   }

   final AtomicReference<Node<E>> top  = new AtomicReference<>();

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

Non-blocking Stack

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

Non-blocking Queue



- Enqueue: new elements are added at the tail
  - 1) new node is referenced by tail.next
  - 2) tail reference is advanced

  => Two pointers have to be updated
- Dequeue: elements are taken from the head

Non-blocking Queue

- **Problems**
  - Enqueue operations may run in parallel
  - Enqueue may be delayed after it has set tail.next
    - Failure of one thread does not prevent other threads from making progress!
  - While an enqueue is active, a dequeue may be executed

- **Invariant**
  - `tail` refers to dummy (i.e. to the same node as head) OR
    `tail` refers to the last element OR
    `tail` refers to second-last element (in the middle of an update)

In this situation the queue is empty

- **Reference**
  - Maged Michael & Michael Scott, *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, PODC 1996*

Non-blocking Queue

```
public class LinkedQueue <E> {

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

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

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head =
                        new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail =
                        new AtomicReference<Node<E>>(dummy);
```

Non-blocking Queue

```
public boolean put(E item) {
    Node<E> newNode = new Node<E>(item, null);
    while (true) {
        Node<E> curTail = tail.get();
        Node<E> tailNext = curTail.next.get();
        if (tailNext != null) {
            // Queue in intermediate state, advance tail
            tail.compareAndSet(curTail, tailNext);
        } else {
            // In consistent state, try inserting new node
            if (curTail.next.compareAndSet(null, newNode)) {
                // Insertion succeeded, try advancing tail
                tail.compareAndSet(curTail, newNode);
                return true;
            }
        }
    }
}
```

Non-blocking Algorithms

- **Characteristics of all non-blocking algorithms**
  - Some work is done speculatively and may have to be redone
- **Thread-Safety**
  - CAS provides atomicity
  - CAS provides visibility (like volatiles)

Lock Free Programming

- Locking and CAS
- Atomics
- Non-Blocking Algorithms
- ABA Problem

ABA Problem

- **ABA problem occurs in lock-free algorithms when a variable which was read has been changed by another thread**
  - A->B->A
- **The CAS operation will compare its A with A and is thinking that "nothing has changed" even though the second thread did work that violates that assumption**
  - load-link/store-conditional does not have this problem
- **Example**
  - Thread T1 reads value A from shared memory
  - T1 is preempted, allowing thread T2 to run
  - T2 modifies the shared memory value A to value B and back to A before preemption
  - T1 begins execution again, sees that the shared memory value has not changed and continues.

AtomicStampedReference<V>

- **Problem can be solved with a stamped reference**

```
public class AtomicStampedReference<V>  {
   public AtomicStampedReference(V ref, int stamp) { ... }

   public V   getReference() { ... }  // returns reference
   public int getStamp() { ... }      // returns stamp
   public V   get(int[] stampHolder) { ... } // returns both

   public void set(V newReference, int newStamp) { ... }

   public boolean compareAndSet(V    expectedReference,
                                V    newReference,
                                int  expectedStamp,
                                int  newStamp) {    }

   public boolean attemptStamp(
         V expectedReference, int newStamp) { ... }
}
```

AtomicStampedReference<V>

- **Implementation**
  - Pair class is defined and referenced over an AtomicReference<V>

```
private static class ReferenceIntegerPair<T> {
   private final T reference;
   private final int integer;
   ReferenceIntegerPair(T r, int i) {
      reference = r; integer = i;
   }
}

private final AtomicReference<ReferenceIntegerPair<V>> atomicRef;
```

AtomicStampedReference<V>

```
public boolean compareAndSet(V      expectedReference,
                             V      newReference,
                             int    expectedStamp,
                             int    newStamp) {
   ReferenceIntegerPair<V> current = atomicRef.get();
   return
          expectedReference == current.reference &&
          expectedStamp == current.integer
      &&
         ((newReference == current.reference &&
            newStamp == current.integer)
         ||
            atomicRef.compareAndSet(
               current,
               new ReferenceIntegerPair<V>(newReference, newStamp)
            )
         );
    }
```

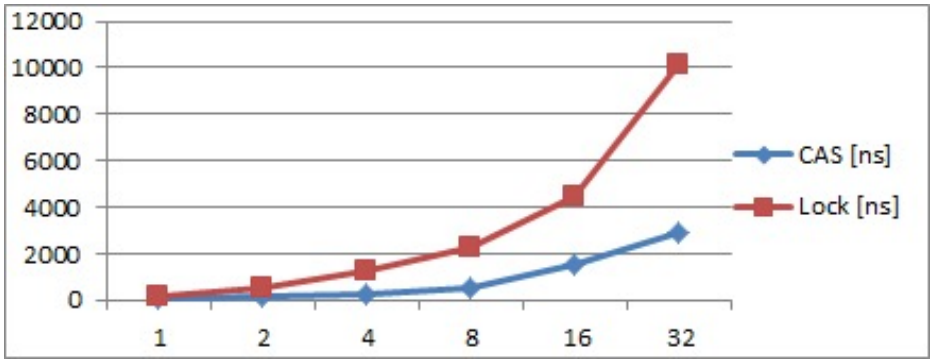
Performance Characteristics

- **Optimistic strategy**
  - When multiple threads update the same variable simultaneously, one wins, the others lose
    - Losers are not punished by suspension
    - Loser can decide whether it wants to try again (or to do nothing)
  - More efficient at low contention
  - At high contention threads block each other
    - Locks suspend threads which reduces CPU usage and (synchronization) traffic on the shared memory bus

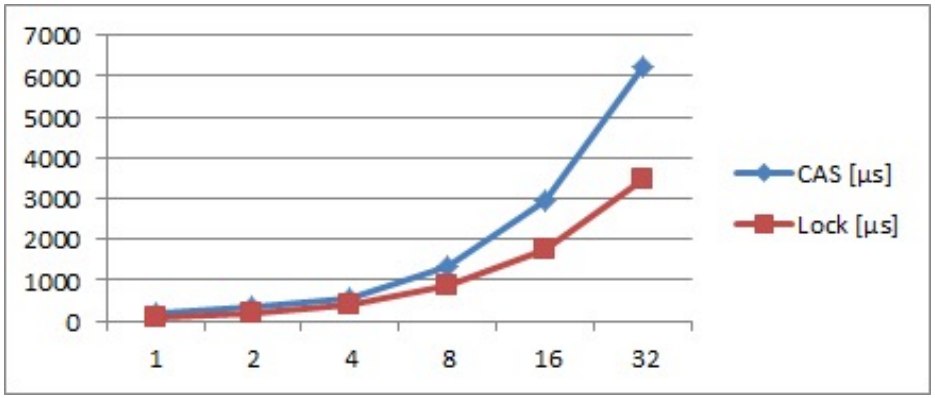
CAS vs Locks

- **Rule of thumb:**
  - With high contention locking tends to outperform atomic variables, but with moderate contention atomic variables outperform locks

Low contention [ns]



High contention [µs]



  - Thus, a cross intersection with traffic light is a better choice than a roundabout if the average arrival rate per each lane exceeds about 0.64 vps which means an arrival rate of nearly 16 vehicles per each 25 seconds.