Atomic variables and non-blocking synchronization

Yang Xu

Outline

- Disadvantages of locking
- Hardware support for concurrency
- Atomic variable classes
- Non-blocking algorithms

Disadvantages of locking

- A lot of overhead
- Especially, under contention
- Delay
- High-priority thread waits for low-priority thread
- …

Conclusion: Locking is a heavyweight mechanism, but modern processors offer a finer-grained technique.

Compare and swap

- Locking – pessimistic
- CAS – optimistic

- “I think V should have the value A;
- If it does, put B there,
- Otherwise don’t change it but tell me I was wrong.”

A non-blocking counter

```
@ThreadSafe
public class CasCounter {
  private SimulatedCAS value;

  public int getValue() {
    return Value.get();
  }

  public int increment() {
    int v;
    do {
      v = value.get();
    }
    while (v != value.compareAndSwap(v,v+1));
    return v+1;
  }
}
```

CAS support in the JVM:

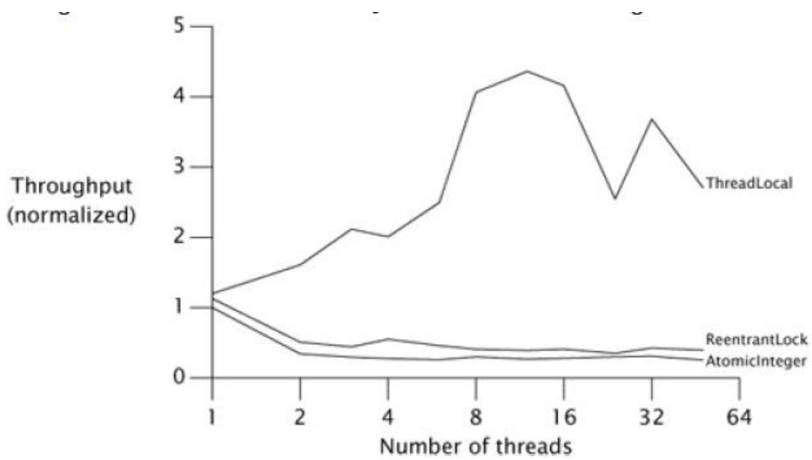
`AtomicXXX` in `java.util.concurrent.atomic`

Atomics as "better volatiles"

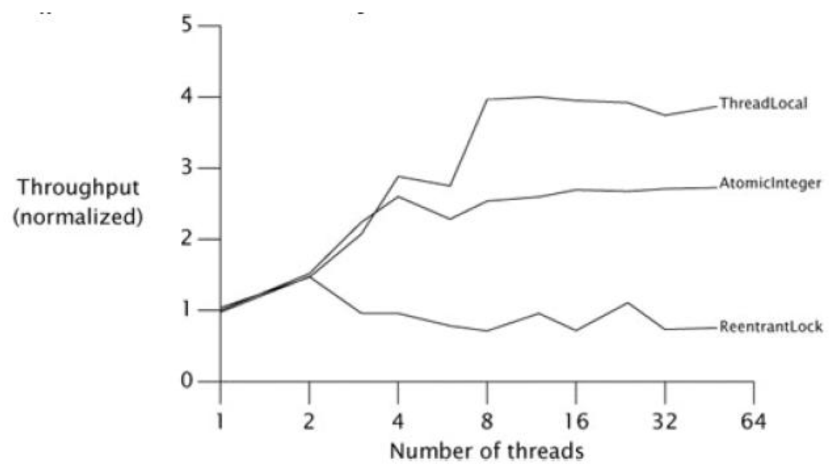
```java
public class CasNumberRange {
  @Immutable
  private static class IntPair {
    final int lower;  // Invariant: lower <= upper
    final int upper;
    ...
  }

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

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

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

A pseudorandom number generator

High contention



Moderate contention



A non-blocking stack

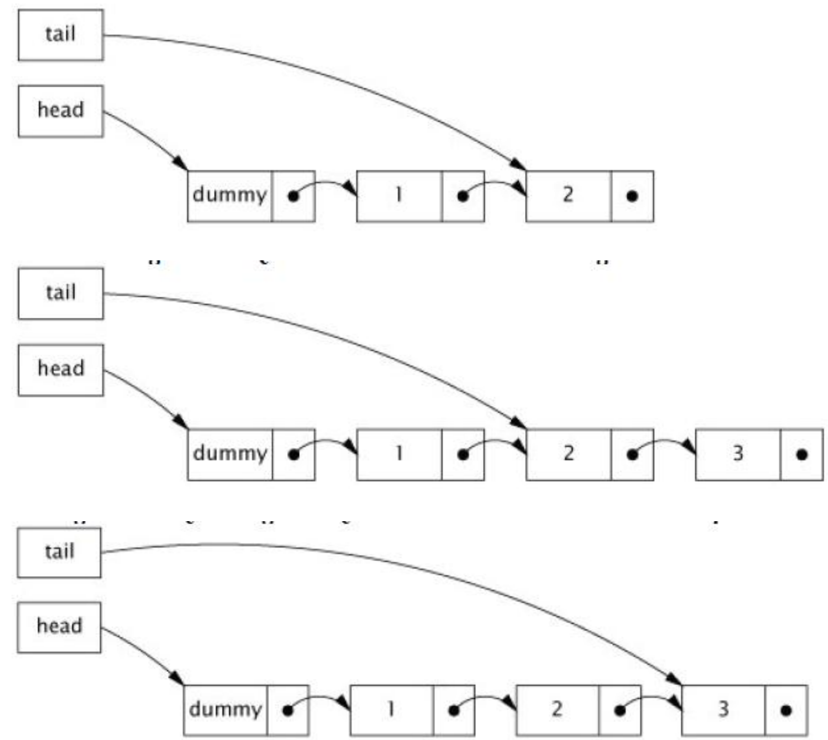
- node = a value + a link to the next node

- push method:
  install a new node on the top of stack
  - succeed
  - fail -> try again

```
@ThreadSafe
public class ConcurrentStack <E> {
  AtomicReference<Node <E>> top = new AtomicReference<Node<E>>();

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

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

    public Node<E item> {
      this.item = item;
    }
  }
}
```

A non-blocking linked list

- 2 pointers refer to the tail node:
  - the next pointer of the current last element
  - the tail pointer
- Should be updated atomically
- `compareAndSet`
- `tail.next` is null or non-null



Atomic field updater

```
private class Node<E> {
  private final E item;
  private volatile Node<E> next;

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

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater
  = AtomicReferenceFieldUpdater.newUpdater(
    Node.class, Node.class, "next");
```

- Use a volatile reference
- Weaker than regular atomic class

The ABA problem

- “Is the value of V still A?”
  -> “Has the value of V changed since I last observed it to be A?”
- Solutions:
  - let the garbage collector mange link nodes
  - a reference -> a reference + a version number

Summary

Non-blocking algorithms:

- Better scalability and liveness
- Difficult to design and implement