Intro to Parallel and Concurrent Programming

The Pursuit of Atomos

WashU CSE 2301
Prof. Dennis Cosgrove
#21: Tue, Nov 11, 2025

Sequential: Yes; Atomic: No

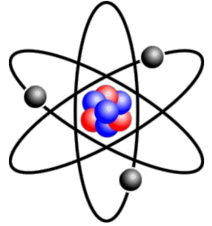
Why is it that read and write within an fork don't operate sequentially?

- they do, just not atomically

https://en.wikipedia.org/wiki/Linearizability

In concurrent programming, an operation (or set of operations) is atomic, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from interrupts, signals, concurrent processes and threads.

Atom



from the Greek “atomos” meaning “uncuttable”

PERCY JACKSON

THE PURSUIT OF ATOMOS

source: https://www.yorinarpati.com/

Atomicity in Computing

"In a concurrent system, processes can access a shared object at the same time. Because multiple processes are accessing a single object, there may arise a situation in which while one process is accessing the object, another process changes its contents. This example demonstrates the need for linearizability. In a linearizable system ***although operations overlap on a shared object, each operation appears to take place instantaneously.****"*

uncuttable works well

S&Q: Are these atomics just objects that utilize locks, or is there something more complex being done in these libraries?

fast hardware-level support for simple thread-safe operations

S&Q: Is the idea of atomic variables something we have to implement, or is it built into most languages?

it is built in. this worksheet is only to try to help you understand how the methods work.

S&Q: Are atomic variables specific to java? What other languages have built-in atomic variables?

c++ std::atomic

S&Q: is there a java interface for atomic variables?

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

Worksheet

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html

If `incrementAndGet` didn't exist

- read
- slam

a = 231

fork

    a = a + 1000

fork

    a = a + 100

- read
- change_from_expected_to_desired_or_do_nothing

a = 231

fork

    expected = a // 231

    change a from 231 to 1000+231

fork

    expected = a // 231

    change a from 231 to 100+231

If `incrementAndGet` didn’t exist

```
while true
    expected = atom.get()
    desired = expected + 1
    if atom.cmpNset(expected, desired)
        break
```

broken terrible: atom.set(`atomic.get()+1`)

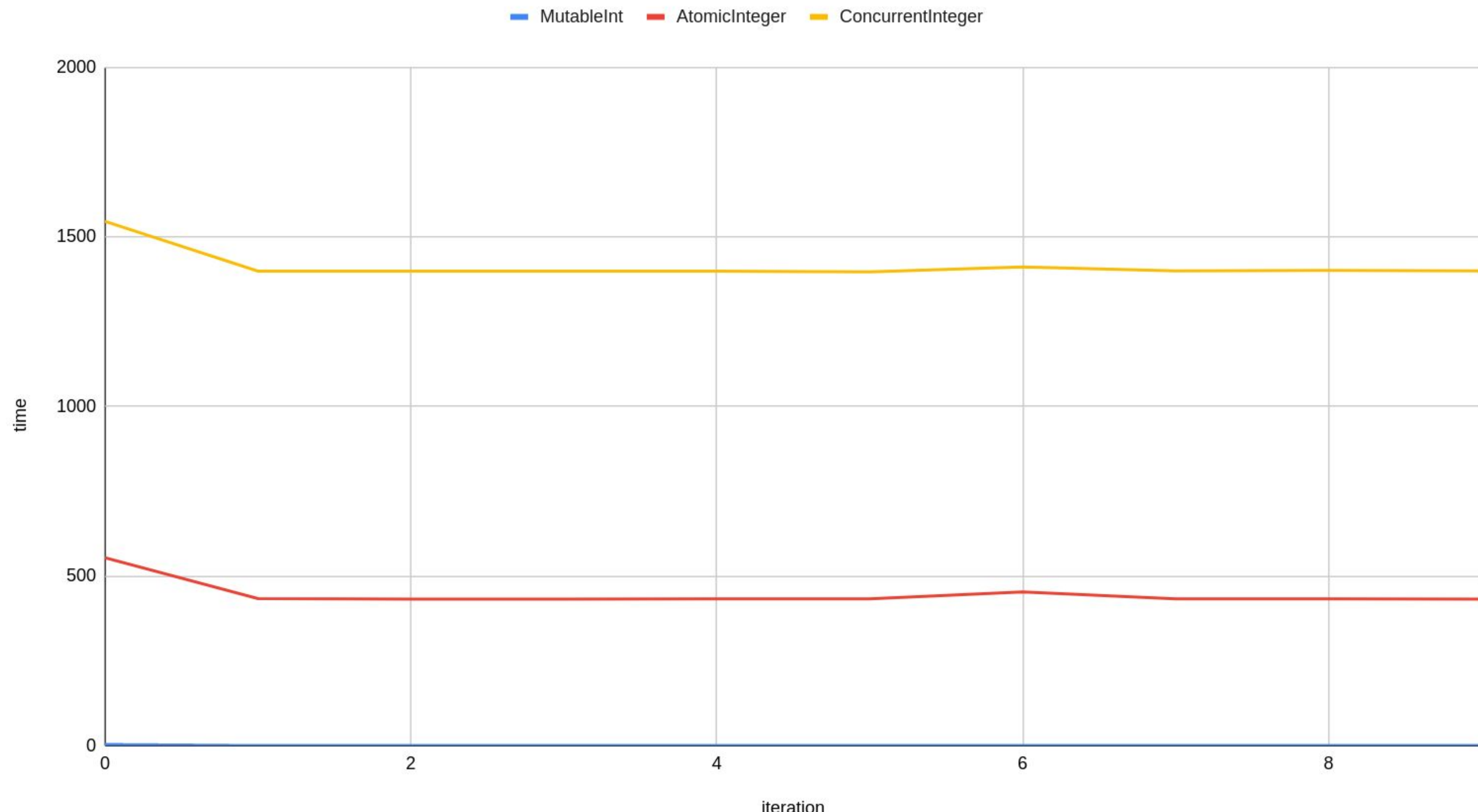
ΛΛΛ

S&Q: Atomics in Sequential

Are atomics useful in sequential code?

- **no**
  (they work, but no need for them)

*** What is the contention level for sequential code?



No Contention. Single Task.
MutableInt
AtomicInteger
ConcurrentInteger
2000
1500
1000
500
0
time
0
2
4
6
8
iteration

```
function increment(atomic)
    while true
        prev = atomic.get()
        next = prev + 1
        if atomic.compareAndSet(prev, next)
            yay
        else
            pass
```

S&Q: I'm still confused as to what atomic variables are and how to use them. For example, how do atomic variables work so that multiple isolations can call them safely without atomicity races? Is there an instance where it's not preferable to use atomic variables over locks?

atomics are blazingly fast but have limited applicability

S&Q: Atomics vs. Locks

say you have a critical section,
it is narrow, really fast, and **well-suited for atomics**
(say: sum += value),
and should **rarely be in contention**?

- use atomics
  - lock overhead is high and not necessary (by definition “well-suited for atomics”)
  - AtomicInteger addAndGet() works

S&Q: Does atomicity only apply to simpler functions like += (ie very close read/write statements) or can it handle more complex situations?

+= is getting to the upper bound of support

Note on “well-suited”

- Locks is a general mechanism
- Atomics are special cases for a common pattern

java.util.concurrent.atomic.AtomicInteger
  addAndGet(delta)
  get()
java.util.concurrent.atomic.AtomicIntegerArray
  addAndGet(i, delta)
  get(i)
java.util.concurrent.atomic.AtomicReference
  compareAndSet(expect,update)
  get()
java.util.concurrent.atomic.AtomicReferenceArray
  compareAndSet(i,expect,update)
  get(i)

How to use Atomics (compareAndSet edition)

```
desiredValue = something
while( true ):
    expectedValue = atomic.get()
    if atomic.compareAndSet( expectedValue, desiredValue ):
        break
```

If `incrementAndGet` didn't exist

```
while( true ):
    expectedValue = atomic.get()
    desiredValue = expectedValue + 1
    if atomic.compareAndSet( expectedValue, desiredValue ):
        break
return desiredValue
```

^ ^ ^

Atomics vs. Locks

say you have a critical section,
and **it is NOT well-suited to directly use atomics**
(say: map.put(key, value)),
and should rarely be in contention?

- use locks

Atomics vs. Locks

say you have a critical section,
and **it is NOT well-suited to directly use atomics**
(say: map.put(key, value)),
~~and should rarely be in contention?~~
and is going to be in **high contention**?

- find a way to lower contention
  - https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

Real World

So, thinking real world applications, you would probably almost always use the locks, correct?

- sadly, data races and non-deterministic bugs are the strategy of choice for many :(
- use thread safe data structures
  - implementation detail
  - see: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
- atomics are very relevant
  - for example, the thread safe data structures above (ConcurrentHashMap for one)
- deal-with-conflicts-when-they-come approaches
  - https://en.wikipedia.org/wiki/Transactional_memory

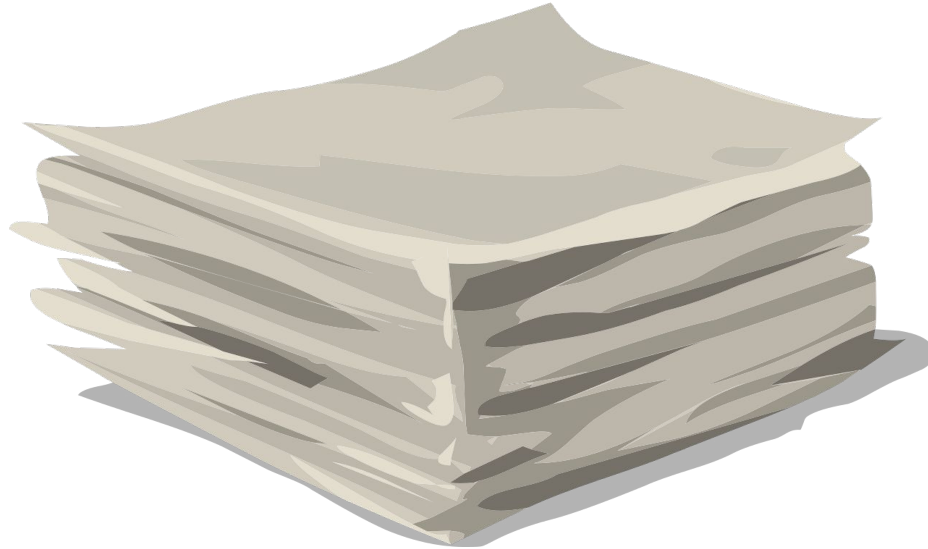
North–South railway aka Reunification Express

How Could You Do Better?

Reflect, Synthesize, Ask, and Turn In Worksheets

Note: you are responsible for getting the worksheets to the us.

Do not leave a pile on the end of your table and hope.



Atomic Stack Exercise

Atomicity Exercise

https://classes.engineering.wustl.edu/cse231/core/index.php/Atomicity

fix the atomicity race and build a class with tell-don't-ask encapsulation

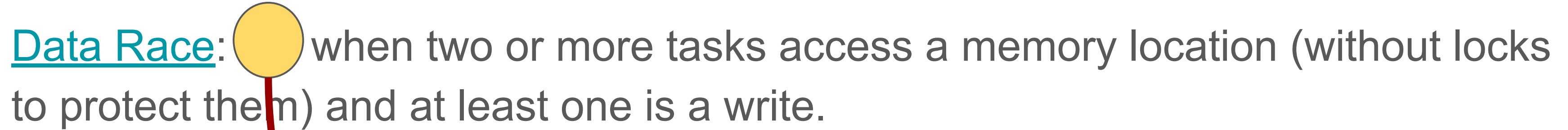
yuck: CheckThenActCourse

addIfSpace returns boolean

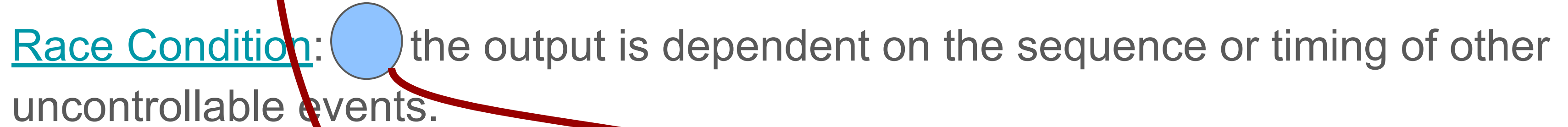
https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

note: tests have flexibility for both compute and merge fans

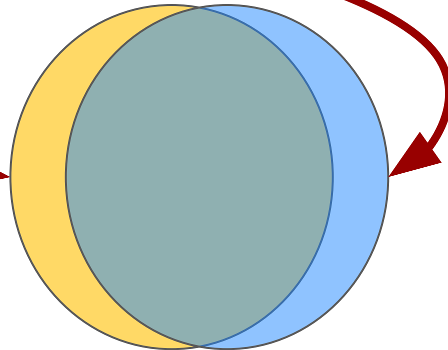
Race Condition



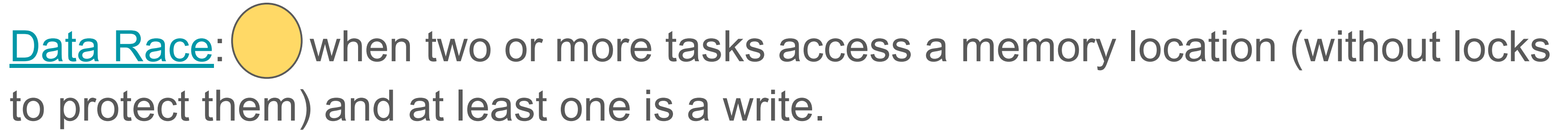
Data Race: when two or more tasks access a memory location (without locks to protect them) and at least one is a write.



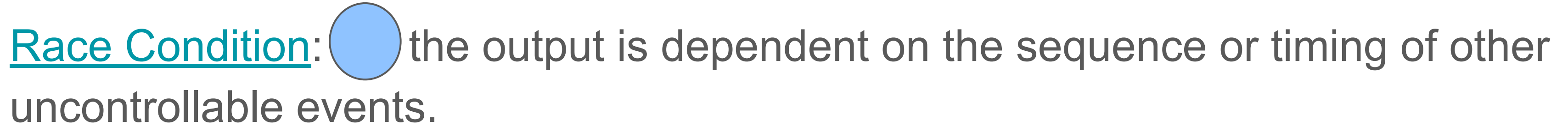
Race Condition: the output is dependent on the sequence or timing of other uncontrollable events.



Race Condition



Data Race: when two or more tasks access a memory location (without locks to protect them) and at least one is a write.



Race Condition: the output is dependent on the sequence or timing of other uncontrollable events.

different output data races

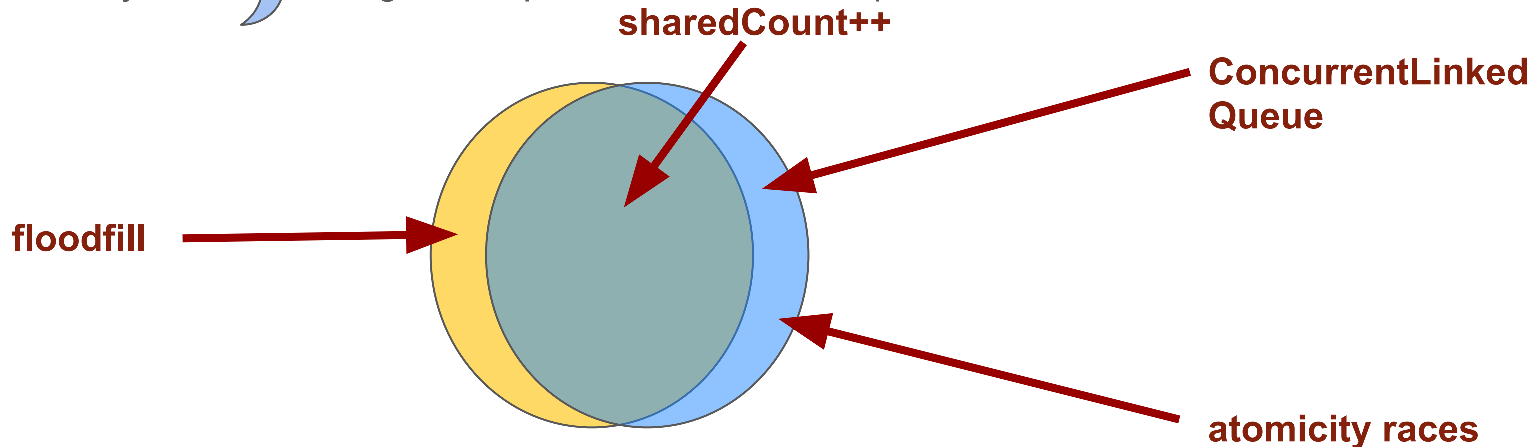
determinacy races

benign data races

atomicity races

Race Terminology

- Benign Data Race: think Floodfill
- Data Race, Race Condition: think sharedCount++
- Non-Data Race, Race Condition:
  - Determinacy Race: think ConcurrentLinkedQueue
  - Atomicity Race: think get then put on ConcurrentMap



```
queue = new ConcurrentLinkedQueue()

Parallel.for(names, name -> {
       queue.add(name)
}
```

Check Then Act, Read Modify Write

```
class MySet
     if list.contains(o):
          pass
     else:
          list.add(o)
```

```
value = map.getOrDefault(key, 0)
value = value + 1
map.put(key, value)
```

Encapsulation: Tell Don’t Ask

Good programming style nicely also lends itself well to parallelism

Parallel/Concurrent Forces Good Practices

All Good Things:

- immutability
- encapsulation (not sharing)
- tell don’t ask

S&Q: Is there any ever reason to not use a thread safe hash map as opposed to a normal one

- if you are going to possibly have more than one task access the map with at least one being a write and you want your code to be correct/not-crash.

S&Q: Whats the benefit of the ConcurrentHashMap if it doesn't solve all of the problems? It seems like it is just as dangerous as using the normal HashMap. I guess the coder should always read the documentation but it doesn't seem to fix all the problems.

- ConcurrentHashMap + compute solves a ton of problems.

Atomicity Races

Frère Jacques, Frère Jacques,
Dormez-vous? Dormez-vous?
Sonnez les matines! Sonnez les matines!
Ding, dang, dong. Ding, dang, dong.

Get then put is not atomic
Call compute. Call compute.
Use ConcurrentHashMap! Use ConcurrentHashMap!
Or say shoot. Or say shoot.

S&Q: Is there a reason java uses concurrent hashmaps instead of making normal hashmaps thread safe?

- performance
- there is overhead in making code thread safe.

Q&S: "Under the hood", how is compute working to ensure all entries are present in the ConcurrentHashMap?

- implementation detail, one should expect some combination of locks and/or atomics

note: you are building your own ConcurrentMap with explicit ReadWriteLocks

S&Q: Can we call the ConcurrentHashHap's method within the compute lambda? Also, can we use forks and joins within the compute lambda, that use ConcurrentHashHap's method?

- NO. the remapping function you pass to compute should be “pure”.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#compute-K-java.util.function.BiFunction-

the computation should be short and simple, and must not attempt to update any other mappings of this Map.

If incrementAndGet didn't exist

```
while(true)
    expected = atomic.get()
    desired = expected + 1
    if(atomic.compareAndSet(expected, desired))
        return desired
```

atom.set(`atomic.get()+1`)

Read Modify Write

```
private static void updatePassport(Map<Bear, String> passportMap, Bear bear, String location) {
    String prevLocations = passportMap.get(bear);
    String nextLocations;
    if (prevLocations != null) {
        nextLocations = prevLocations + "; " + location;
    } else {
        nextLocations = location;
    }
    passportMap.put(bear, nextLocations);
}
```

SuspectWordCount

```
public class SuspectWordCount {
    public static Map<String, Integer> countWords(Iterable<String> words) {
        Map<String, Integer> map = new HashMap<>();
        join_void_fork_loop(words, (String word) -> {
            Integer count = map.get(word);
            if (count != null) {
                count = count + 1;
            } else {
                count = 1;
            }
            map.put(word, count);
        });
        return map;
    }
}
```

StockPortfolio

```java
private int transfer(String listingSymbol, int deltaShareCount) {
    Integer oldValue = this.map.get(listingSymbol);
    Integer newValue;
    if (oldValue != null) {
        newValue = oldValue + deltaShareCount;
    } else {
        newValue = deltaShareCount;
    }
    this.map.put(listingSymbol, newValue);
    return newValue;
}
```

CheckThenActCourse (Asking For Trouble)

```
public class CheckThenActCourse {
   private final Set<Student> students;
   private final int limit;
   public CheckThenActCourse(int limit) {
      this.students = new HashSet<>();
      this.limit = limit;
   }
   public boolean isSpaceRemaining() {
      synchronized (students) {
         return students.size() < limit;
      }
   }
   public void add(Student student) {
      synchronized (students) {
         students.add(student);
      }
   }
   public boolean drop(Student student) {
      synchronized (students) {
         return students.remove(student);
      }
   }
}
```

```
fork everybody {
      sync(checkAC)

    if (checkThenActCourse.isSpaceRemaining()) {
       checkThenActCourse.add(student);
    }
}

fork
      sync(checkAC)

            checkThenActCount.add(busStu)
```

Course (don’t lead your clients into atomicity races)

```
public class Course {
    private final Set<Student> students;
    private final int limit;
    public Course(int limit) {
        throw new NotYetImplementedException();
    }
    public int limit() {
        throw new NotYetImplementedException();
    }
    public boolean addIfSpace(Student student) {
        throw new NotYetImplementedException();
    }
    public boolean drop(Student student) {
        throw new NotYetImplementedException();
    }
}
```

Atomicity Exercise

https://classes.engineering.wustl.edu/cse231/core/index.php/Atomicity

fix the atomicity race and build a class with tell-don't-ask encapsulation

yuck: CheckThenActCourse

addIfSpace returns boolean

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

note: tests have flexibility for both compute and merge fans

Atomicity Races

Frère Jacques, Frère Jacques,
Dormez-vous? Dormez-vous?
Sonnez les matines! Sonnez les matines!
Ding, dang, dong. Ding, dang, dong.

Get then put is not atomic
Call compute. Call compute.
Use ConcurrentHashMap! Use ConcurrentHashMap!
Or say shoot. Or say shoot.

Eminem and CCC (Cosgrove Creates Confusion)

Atomic Stack Exercise

AtomicityRaceJustWaitingToHappenPoorlyDesignedStack

```
public interface AtomicityRaceJustWaitingToHappenPoorlyDesignedStack<E> {
    boolean isEmpty();
    E peek();
    E pop();
    void push(E value);
}
```

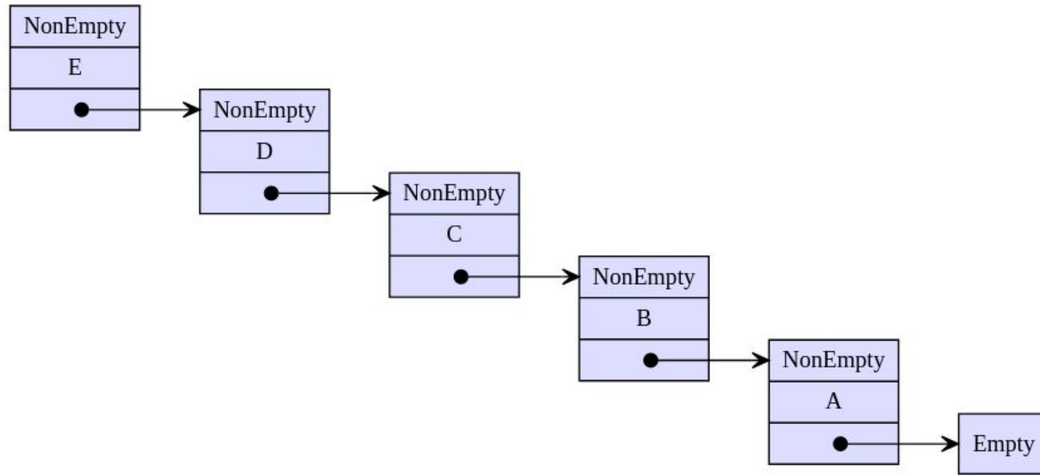
AtomicityRaceClient

```
public class AtomicityRaceClient {
   public static void consume(AtomicityRaceJustWaitingToHappenPoorlyDesignedStack<String> stack, int n) {
       for (int i = 0; i < n; ++i) {
           if (stack.isEmpty()) {
               // pass
           } else {
               String text = stack.pop();
               text.length();
           }
       }
   }

   public static void produce(AtomicityRaceJustWaitingToHappenPoorlyDesignedStack<String> stack, int n) {
       for (int i = 0; i < n; ++i) {
           stack.push("spam");
       }
   }
}
```

S&Q: Can you declare data structures as atomic, or is it just for more primitive types?

primitives (but you can sometimes build a data structure with atomic primitives)



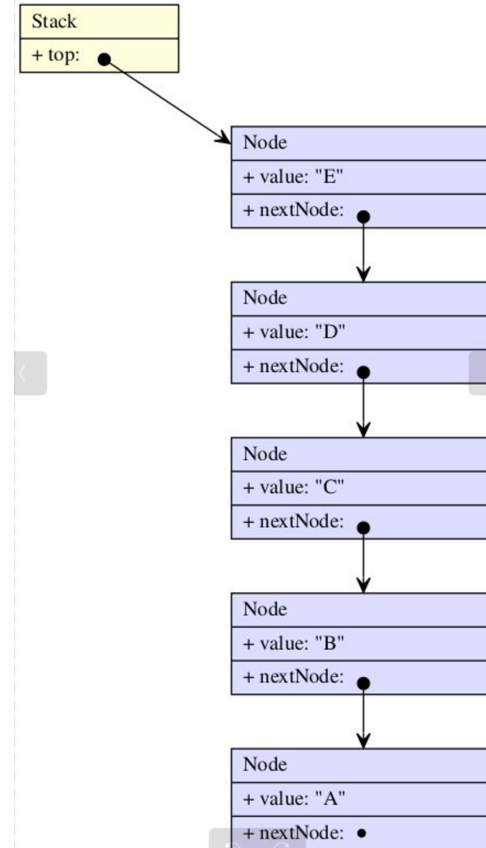
Stack<E>

```java
public interface Stack<E> {
    void push(E value);
    E peek();
    E pop();
}
```

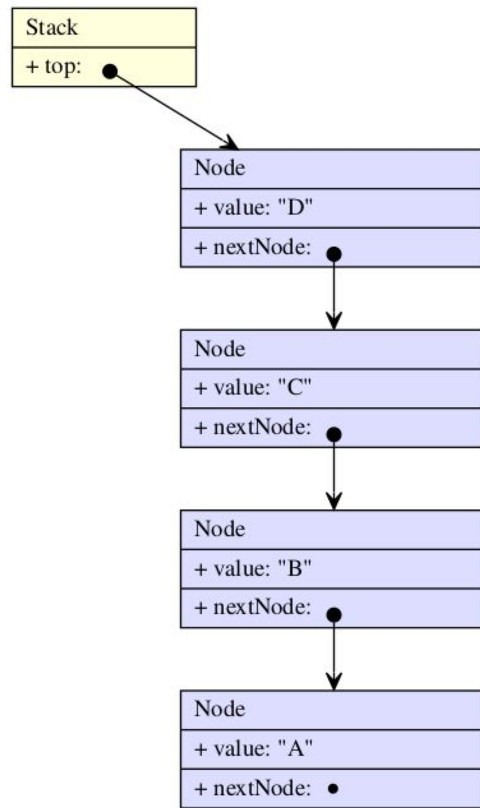
NotThreadSafeStack<E>

```
@NotThreadSafe
public class NotThreadSafeStack<E> implements Stack<E> {
   @Override
   public void push(E value) {
       throw new NotYetImplementedException();
   }
   @Override
   public E peek() {
       throw new NotYetImplementedException();
   }
   @Override
   public E pop() {
       throw new NotYetImplementedException();
   }
}
```

```java
Stack<String> stack = new NotThreadSafeStack<>();
stack.push("A");
stack.push("B");
stack.push("C");
stack.push("D");
stack.push("E");
String e = stack.pop();
```



```java
Stack<String> stack = new NotThreadSafeStack<>();
stack.push("A");
stack.push("B");
stack.push("C");
stack.push("D");
stack.push("E");
String e = stack.pop();
```



*** Easiest Way To Make Thread Safe?

```java
@ThreadSafe
public class ConcurrentStack<E> implements Stack<E> {
   public ConcurrentStack() {
       throw new NotYetImplementedException();
   }
   @Override
   public void push(E value) {
       throw new NotYetImplementedException();
   }
   @Override
   public E peek() {
       throw new NotYetImplementedException();
   }
   @Override
   public E pop() {
       throw new NotYetImplementedException();
   }
}
```

AtomicStack<E>

```
@ThreadSafe
public final class AtomicStack<E> implements Stack<E> {
   public AtomicStack(BiFunction<E, Optional<Node<E>>, Node<E>> nodeCreator) {
       throw new NotYetImplementedException();
   }
   public BiFunction<E, Optional<Node<E>>, Node<E>> nodeCreator() {
       throw new NotYetImplementedException();
   }
   @Override
   public void push(E value) {
       throw new NotYetImplementedException();
   }
   @Override
   public E peek() {
       throw new NotYetImplementedException();
   }
   @Override
   public E pop() {
       throw new NotYetImplementedException();
   }
}
```

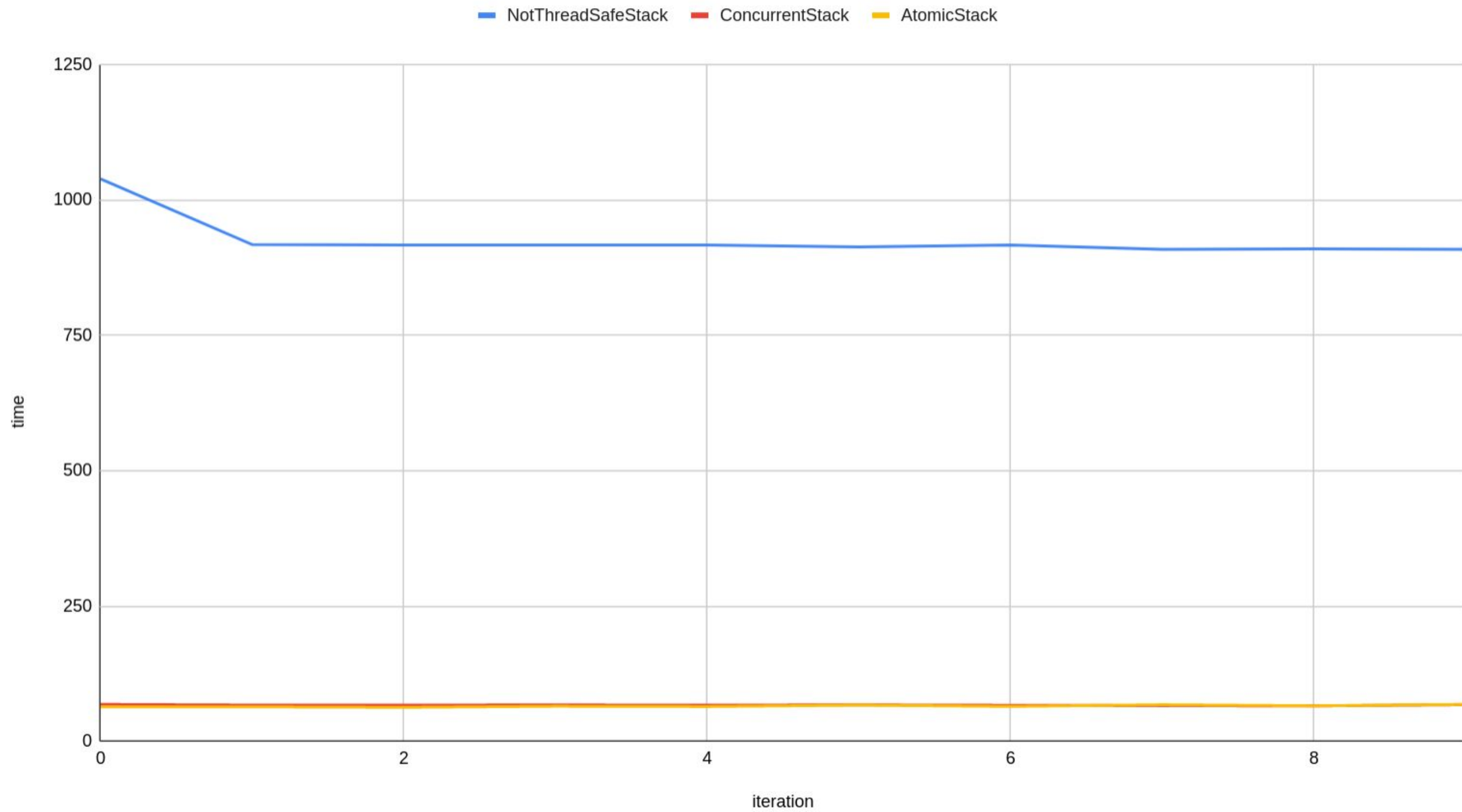
How would `incrementAndGet` Perform

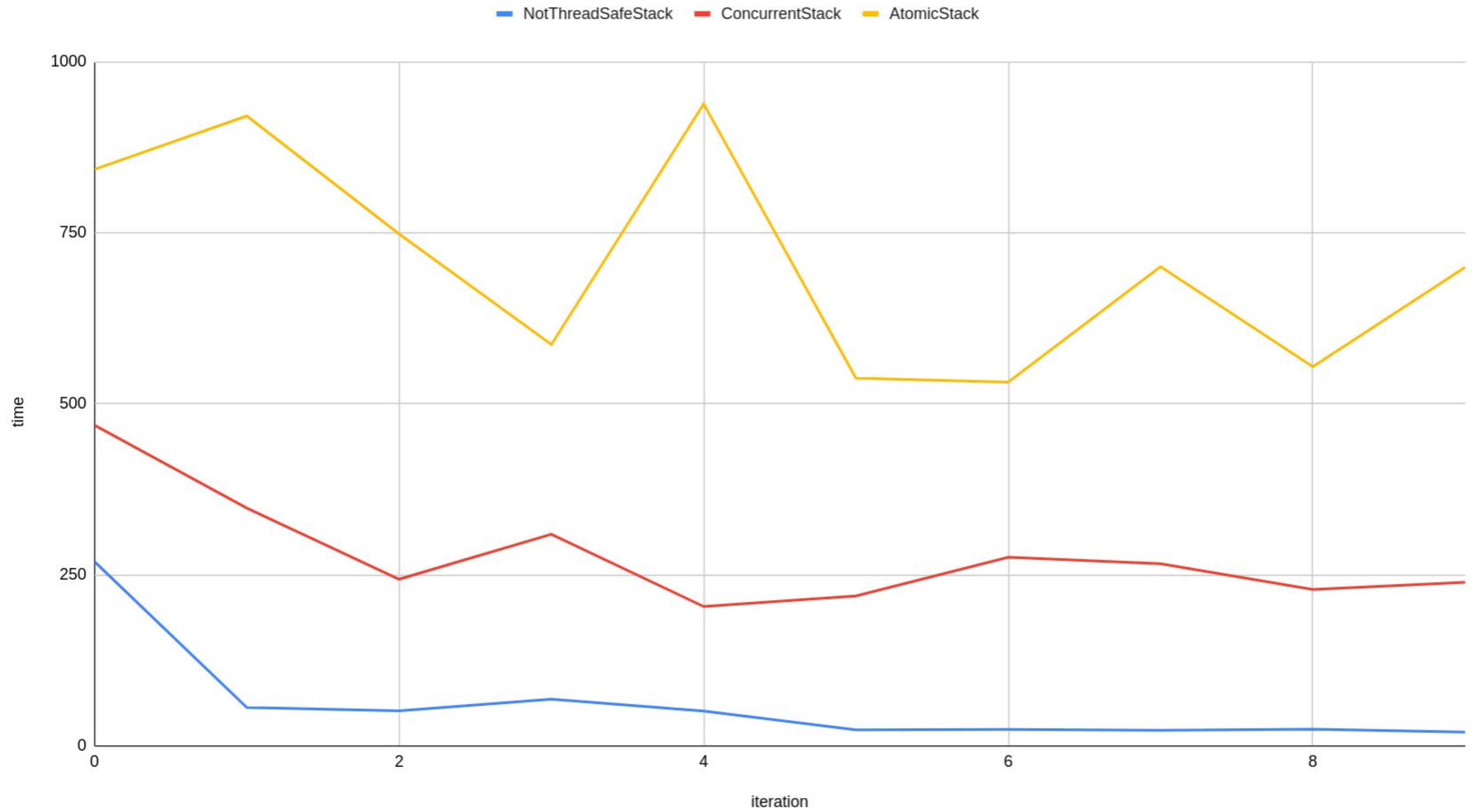
High contention?

Medium contention?

Low contention?

```
while(true)
    expected = atomic.get()
    desired = expected + 1
    if(atomic.compareAndSet(expected, desired))
        return desired
```


Not As Terrible Contention
NotThreadSafeStack
ConcurrentStack
AtomicStack
1250
1000
750
500
250
0
time
0
2
4
6
8
iteration



Extremely High Contention
NotThreadSafeStack
ConcurrentStack
AtomicStack
time
iteration
1000
750
500
250
0
0
2
4
6
8

NotAsTerribleContentionStackPerformanceTiming

```
public class NotAsTerribleContentionStackPerformanceTiming <E> {
  public static void main(String[] args) throws InterruptedException , ExecutionException {
      final int NUM_PROCESSORS = Runtime.getRuntime().availableProcessors ();
      final int NUM_VALUES_PER_PROCESSOR = 100;

      Function<Integer, Double> pushFunction = index -> {
          double result = index;
          for (int i = 0; i < 100_000; ++i) {
              result = Math.sqrt(result);
          }
          return result;
      };
      Function<Optional<Double>, Double> popFunction = valueOpt -> {
          double result = valueOpt.isPresent() ? valueOpt.get() : Math.PI;
          for (int i = 0; i < 100_000; ++i) {
              result = Math.sqrt(result);
          }
          return result;
      };

      PerformanceUtils.recordPerformances (Arrays.asList(
              new NotThreadSafeStackPerformanceRunnable <>(pushFunction, popFunction,
                      NUM_PROCESSORS * NUM_VALUES_PER_PROCESSOR),
              new ConcurrentStackPerformanceRunnable <>(pushFunction, popFunction, NUM_PROCESSORS,
                      NUM_VALUES_PER_PROCESSOR),
              new AtomicStackPerformanceRunnable <>(pushFunction, popFunction, NUM_PROCESSORS,
                      NUM_VALUES_PER_PROCESSOR)));
  }
}
```

ExtremelyHighContentionStackPerformanceTiming

```
public class ExtremelyHighContentionStackPerformanceTiming<E> {
   public static void main(String[] args) throws InterruptedException, ExecutionException {
       final int NUM_PROCESSORS = Runtime.getRuntime().availableProcessors();
       final int NUM_VALUES_PER_PROCESSOR = 100_000;

       Function<Integer, Integer> pushFunction = index -> index;
       Function<Optional<Integer>, Integer> popFunction = value -> 0;

       PerformanceUtils.recordPerformances(Arrays.asList(
               new NotThreadSafeStackPerformanceRunnable<>(pushFunction, popFunction,
                       NUM_PROCESSORS * NUM_VALUES_PER_PROCESSOR),
               new ConcurrentStackPerformanceRunnable<>(pushFunction, popFunction, NUM_PROCESSORS,
                       NUM_VALUES_PER_PROCESSOR),
               new AtomicStackPerformanceRunnable<>(pushFunction, popFunction, NUM_PROCESSORS,
                       NUM_VALUES_PER_PROCESSOR)));
   }
}
```

Atomics are Optimistic Concurrency

Optimistic Storytime

Distill Down To One Thing

atomics can help if you can distill down to one uncuttable operation

Atomic Stack Exercise