Java Concurrency Gotchas

Alex Miller

Questions to answer

- What are common concurrency problems?
- Why are they problems?
- How do I detect these problems?
- How to I correct these problems?

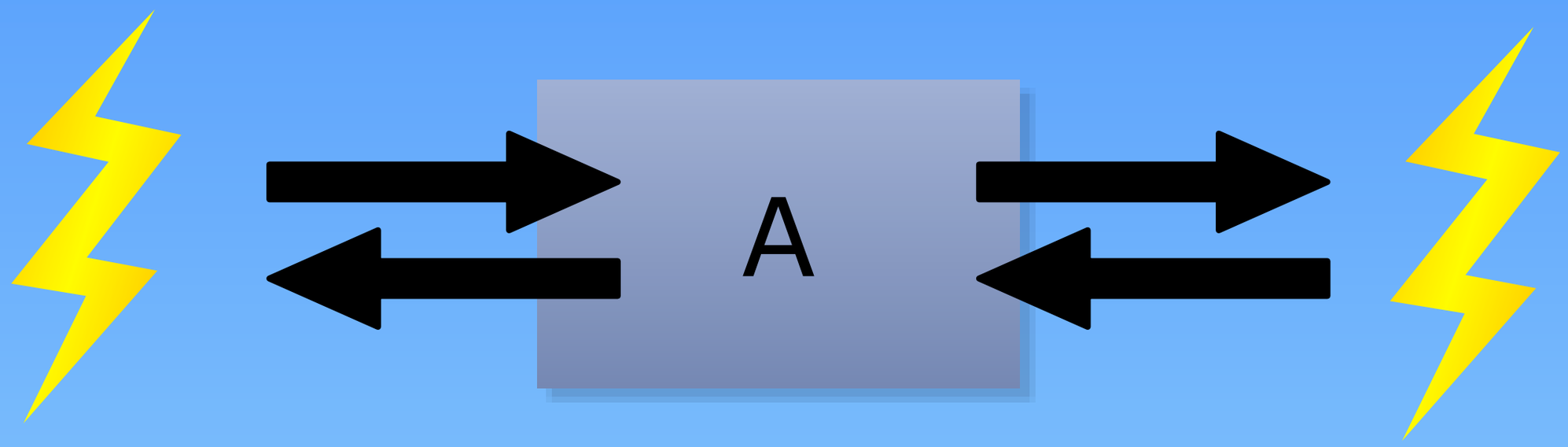
Areas of Focus

- Shared Data
- Coordination
- Performance

Areas of Focus

- **Shared Data**
  - Locking
  - Visibility
  - Atomicity
  - Safe Publication
- Coordination
- Performance

Unprotected Field Access

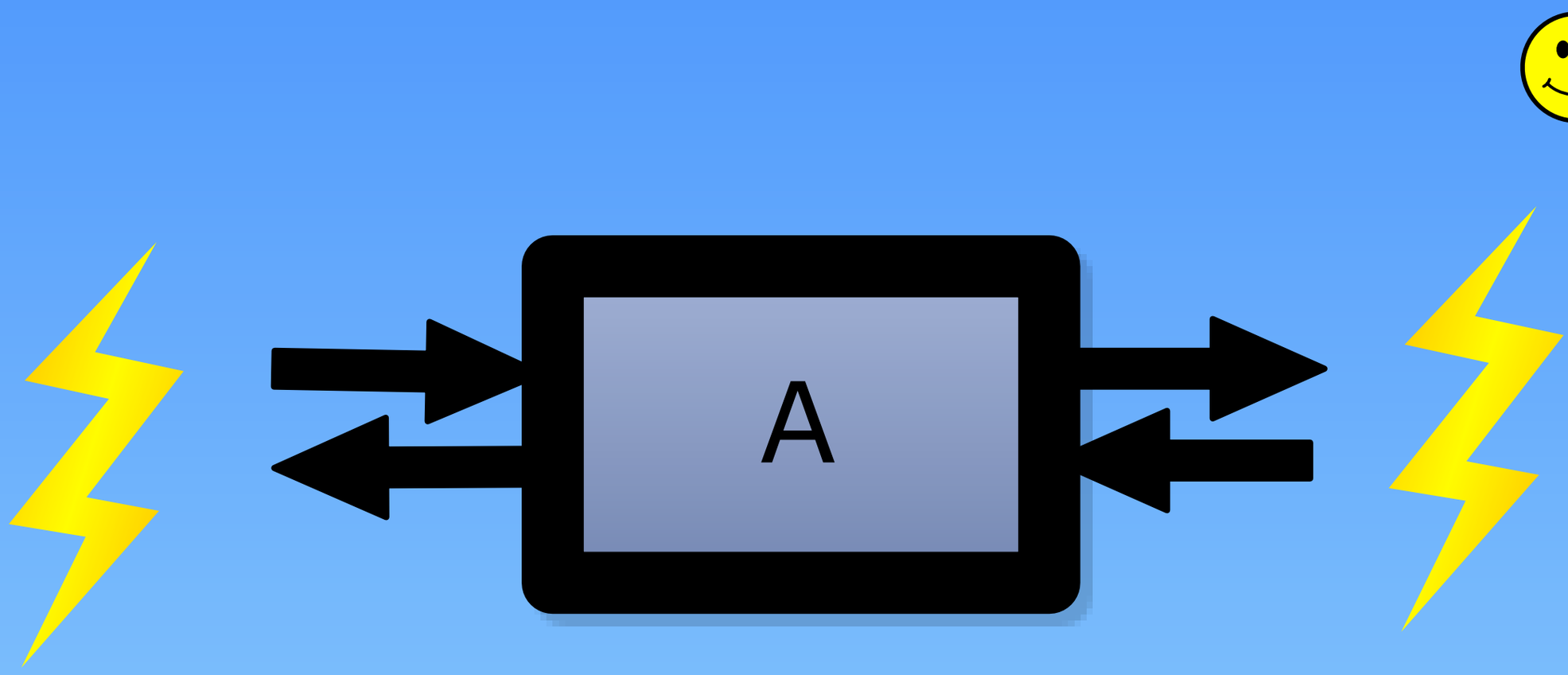


What happens if we modify data without locking?


inevitable FAIL
failblog.org


Writer
Readers
Shared state
inevitable FAIL
failblog.org

Locking



Shared Mutable Statics

```
public class MutableStatics {

  private static final DateFormat FORMAT =
   DateFormat.getDateInstance(DateFormat.MEDIUM);

  public static Date parse(String str)
  throws ParseException {
    return FORMAT.parse(str);
  }

  public static void main(String arg[])
  throws Exception {
    MutableStatics.parse("Jan 1, 2000");
  }
}
```

FORMAT is mutable

...and this mutates it outside synchronization

Shared mutable statics - instance per call

```
public class MutableStatics {

  public static Date parse(String str)
  throws ParseException {
    DateFormat format =
      DateFormat.getDateInstance(DateFormat.MEDIUM);
    return format.parse(str);
  }

  public static void main(String arg[])
  throws Exception {
    MutableStatics.parse("Jan 1, 2000");
  }
}
```

Shared mutable statics - ThreadLocal

```
public class MutableStatics {
  private static final ThreadLocal<DateFormat> FORMAT
    = new ThreadLocal<DateFormat>() {
      @Override protected DateFormat initialValue() {
        return DateFormat.getDateInstance(
                                    DateFormat.MEDIUM);
      }
  };

  public static Date parse(String str)
  throws ParseException {
    return FORMAT.get().parse(str);
  }
}
```

Common JDK Examples

Danger!

- DateFormat
- Calendar
- Matcher

Safe

- Random
- Pattern

Synchronization

```
private int myField;

synchronized( What goes here? ) {
  myField = 0;
}
```

DO NOT:
synchronize on null

```
MyObject obj = null;

synchronized( obj ) {
  // work
}
```

NullPointerException!

DO NOT:
change instance

```
MyObject obj = new MyObject();

synchronized( obj ) {
  obj = new MyObject();
}
```

no longer synchronizing
on same object!

DO NOT:
synch on string literals

```
private static final String LOCK = "LOCK";
synchronized( LOCK ) {
  // work
}
```

What is the scope of LOCK?

DO NOT:
synch on autoboxed vals

```
private static final Integer LOCK = 0;
synchronized( LOCK ) {
  // work
}
```

What is the scope of LOCK?

DO NOT:
synch on ReentrantLock

```
Lock lock = new ReentrantLock();
synchronized(lock) {
  // ...
}
```

Probably not what you meant here

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // ...
} finally {
  lock.unlock();
}
```

Probably more like this...

What should I lock on?

```
// The field you're protecting
private final Map map = ...
synchronized(map) {
  // ...access map
}
```

```
// Explicit lock object
private final Object lock = new Object();
synchronized(lock) {
  // ...modify state
}
```

Visibility

Visibility problems

int x = 5;

Thread 1:

```
if(x == 5) {
    x = 10;
}
```

Thread 2:

```
System.out.println(x);
```

Visibility problems

volatile int x = 5;

Thread 1:

```
if(x == 5) {
   x = 10;
}
```

Thread 2:

```
System.out.println(x);
```

Inconsistent Synchronization

```
public class SomeData {
  private final Map data = new HashMap();

  public void set(String key, String value) {
    synchronized(data) {
      data.put(key, value);
    }
  }

  public String get(String key) {
    return data.get(key);
  }
}
```

Protecting writes

...but not reads

Double-checked locking

```
public class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if(instance == null) {
      synchronized(Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Attempt to avoid synchronization

Double-checked locking

```
public class Singleton {
  private static Singleton instance;

  public static Singleton getInstance() {
    if(instance == null) {
      synchronized(Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

READ

READ

WRITE

Double-checked locking - volatile

```
public class Singleton {
  private static volatile Singleton instance;

  public static Singleton getInstance() {
    if(instance == null) {
      synchronized(Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Double-checked locking - initialize on demand

```
public class Singleton {
  private static class SingletonHolder {
    private static final Singleton instance
        = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.instance;
  }
}
```

volatile arrays

```
public final class VolatileArray {

  private volatile boolean[] vals;

  public void flip(int i) {
    vals[i] = true;
  }

  public boolean flipped(int i) {
    return vals[i];
  }
}
```

Is the value of vals[i] visible to other threads?

Atomicity

Volatile counter

```
public class Counter {

   private volatile int count;

   public int next() {
      return count++;
   }
}
```

Looks atomic to me!

AtomicInteger counter

```
public class Counter {

  private AtomicInteger count =
    new AtomicInteger();

  public int next() {
    return count.getAndIncrement();
  }
}
```

Really atomic via encapsulation over multiple actions

Composing atomic actions

```
public Object putIfAbsent(
  Hashtable table, Object key, Object value) {

  if(table.containsKey(key)) {
    // already present, return existing
    table.get(key);
    return null;

  } else {
    // doesn't exist, create new value
    return table.put(key, value);
  }
}
```

Hashtable is thread-safe

{ READ

{ READ

{ WRITE

Composing atomic actions

```
public Object putIfAbsent(
  Hashtable table, Object key, Object value) {

  if(table.containsKey(key)) {
    // already present, return existing
    table.get(key);
    return null;

  } else {
    // doesn't exist, create new value
    return table.put(key, value);
  }
}
```

Hashtable is thread-safe

READ

READ

WRITE

Participate in lock

```
public Object putIfAbsent(
   Hashtable table, Object key, Object value) {

   synchronized(table) {
     if(table.containsKey(key)) {
       table.get(key);
       return null;
     } else {
       return table.put(key, value);
     }
   }
}
```

Hashtable is thread-safe

Encapsulated compound actions

```
public Object putIfAbsent(
   ConcurrentHashMap table, Object key, Object value) {

   return table.putIfAbsent(key, value);
}
```

Encpasulation FTW!

Assignment of 64 bit values

```
public class LongAssignment {

  private long x;

  public void setLong(long val)  {
    x = val;
  }

}
```

Looks atomic to me, but is it?

Assignment of 64 bit values - volatile

```
public class LongAssignment {

  private volatile long x;

  public void setLong(long val) {
    x = val;
  }

}
```

Safe publication

Listener in constructor

```
public interface DispatchListener {
   void newFare(Customer customer);
}

public class Taxi
   implements DispatchListener {

   public Taxi(Dispatcher dispatcher) {
      dispatcher.registerListener(this);
      // other initialization
   }

   public void newFare(Customer customer) {
      // go to new customer's location
   }
}
```

We just published a reference to **this**...oops!

Starting thread in constructor

```
public class Cache {

  private final Thread cleanerThread;

  public Cache() {
    cleanerThread = new Thread(new Cleaner(this));
    cleanerThread.start();
  }

  // Clean will call back to this method
  public void cleanup() {
    // clean up Cache
  }
}
```

this escapes again!

Static factory method

```
public class Cache {
  // ...

  public static Cache newCache() {
    Cache cache = new Cache();
    cache.startCleanerThread();
    return cache;
  }
}
```

Areas of Focus

- Shared Data
- **Coordination**
  - Threads
  - Wait/notify
- Performance

Threads

- **THOU SHALT NOT:**
  - call Thread.stop()

    All monitors unlocked by ThreadDeath
  - call Thread.suspend() or Thread.resume()

    Can lead to deadlock
  - call Thread.destroy()

    Not implemented (or safe)
  - call Thread.run()

    Won’t start Thread! Still in caller Thread.
  - use ThreadGroups

    Use a ThreadPoolExecutor instead.

wait/notify

```
// Thread 1
synchronized(lock) {
  while(! someCondition()) {
    lock.wait();
  }
}


// Thread 2
synchronized(lock) {
  satisfyCondition();
  lock.notifyAll();
}
```

You **must** synchronize.

Always wait in a loop.

Synchronize here too!

Condition is similar

```
private final Lock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();

public void waitTillChange() {
   lock.lock();
   try {
     while(! someCondition())  condition.await();
   } finally {
     lock.unlock();
   }
}
public void change() {
   lock.lock();
   try {
     satisfyCondition();
     condition.signalAll();
   } finally { lock.unlock(); } }
```

Condition is more flexible than wait/notify.

Areas of Focus

- Shared Data
- Coordination
- **Performance**
  - Deadlock
  - Spin wait
  - Thread contention

Deadlock

```
// Thread 1
synchronized(lock1) {
  synchronized(lock2) {
    // stuff
  }
}

// Thread 2
synchronized(lock2) {
  synchronized(lock1) {
    // stuff
  }
}
```

Classic deadlock.

Deadlock avoidance

- Lock splitting
- Lock ordering
- Lock timeout
- tryLock

Spin wait

```
// Not efficient
private volatile boolean flag = false;

public void waitTillChange() {
   while(! flag) {
      Thread.sleep(100);
   }
}

public void change() {
   flag = true;
}
```

Spin on flag, waiting for a change.

Replace with Condition

```
private final Lock lock = new ReentrantLock();
private final Condition condition = lock.newCondition();
private boolean flag = false;

public void waitTillChange() {
   lock.lock();
   try {
     while(! flag)  condition.await();
   } finally {
     lock.unlock();
   }
}
public void change() {
   lock.lock();
   try {
     flag = true;
     condition.signalAll();
   } finally { lock.unlock(); } }
```

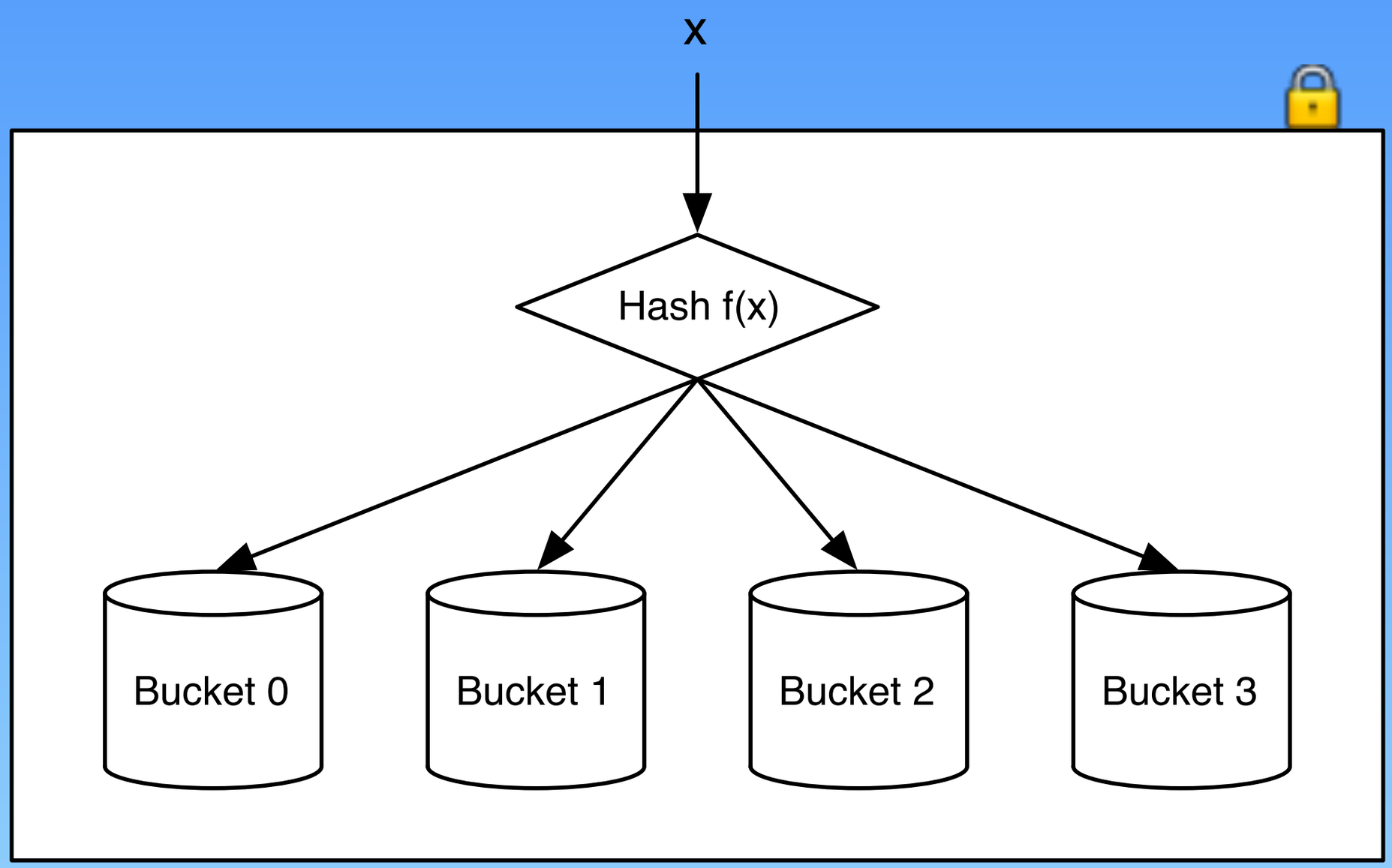
Better but longer.

CountDownLatch

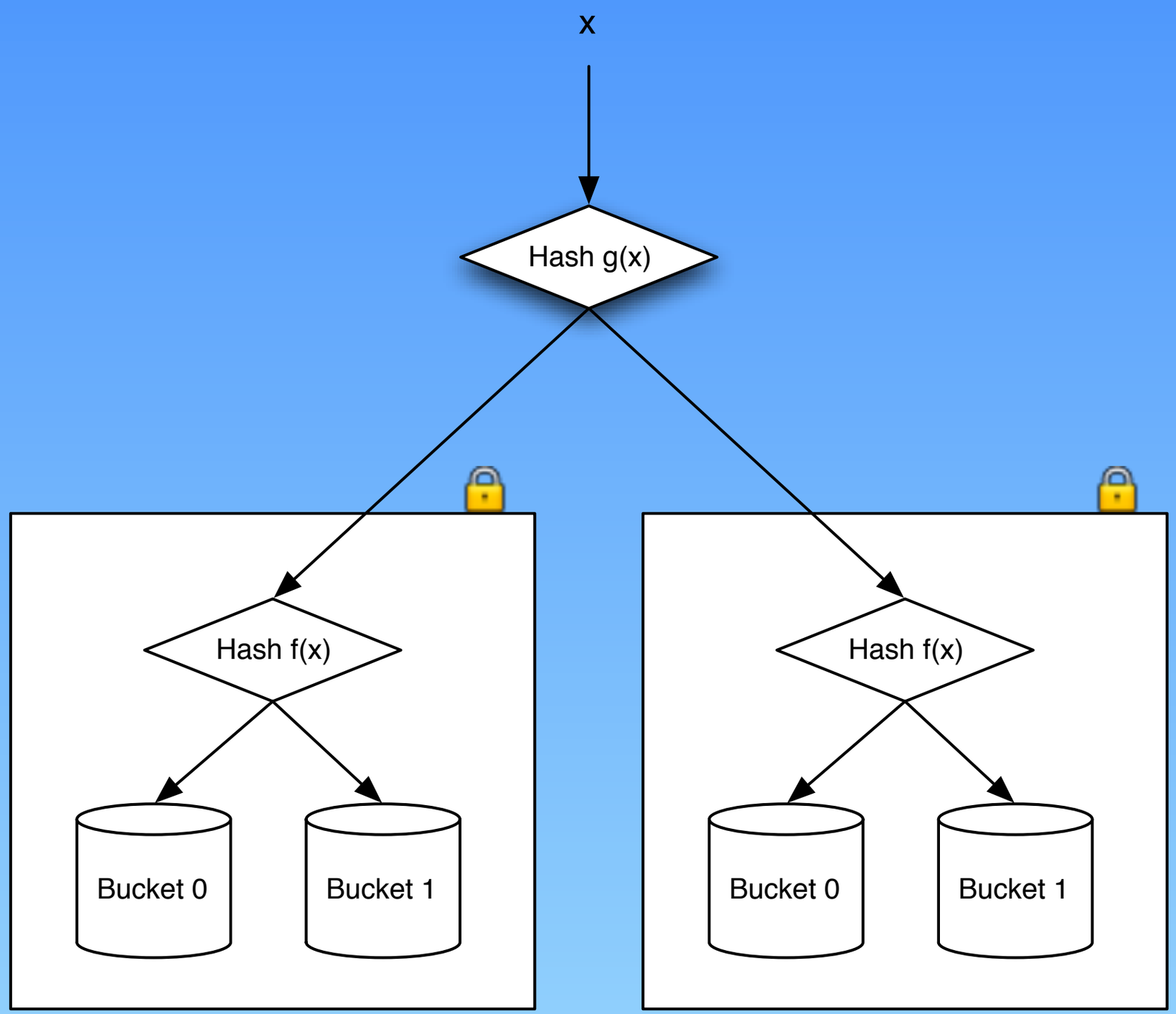
```
private final CountDownLatch latch =
   new CountDownLatch(1);

public void waitTillChange() {
   latch.await();
}

public void change() {
   latch.countDown();
}
```

Coordination classes like CountDownLatch and CyclicBarrier cover many common uses better than Condition.

Lock contention



Lock striping



Final Exam

```
public class StatisticsImpl implements Statistics,
StatisticsImplementor {
  private long queryExecutionCount;

  public synchronized void queryExecuted(
                      String hql, int rows, long time) {
    queryExecutionCount++;
    // ... other stat collection
  }

  public long getQueryExecutionCount() {
    return queryExecutionCount;
  }

  public synchronized void clear() {
    queryExecutionCount = 0;
    // ... clear all other stats
  }
}
```

Final Exam

```
public class StatisticsImpl implements Statistics,
StatisticsImplementor {
  private long queryExecutionCount;

  public synchronized void queryExecuted(
                        String hql, int rows, long time) {
    queryExecutionCount++;
    // ... other stat collection
  }

  public long getQueryExecutionCount() {
    return queryExecutionCount;
  }

  public synchronized void clear() {
    queryExecutionCount = 0;
    // ... clear all other stats
  }
}
```

Single shared lock for ALL stat values

Read shared value w/o synchronization

Non-atomic read of long value

Race condition if reading stat and clearing

Thanks...

Twitter	http://twitter.com/puredanger
Blog	http://tech.puredanger.com
Concurrency links	http://concurrency.tumblr.com
Refcard	http://refcardz.dzone.com/refcardz/core-java-concurrency
Slides	http://slideshare.net/alexmiller