Java is a trademark of Sun Microsystems, Inc.

JavaOne℠

Java™ Platform Concurrency Gotchas

Alex Miller
Terracotta

Questions to answer

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

Taxonomy of Concurrency Gotchas

- Shared Data
- Coordination
- Performance

Shared Data

- Locking
- Visibility
- Atomicity
- Safe Publication

What happens if we modify data without locking?

What happens if we modify data without locking?

Hint: it’s not good.


inevitable FAIL
failblog.org

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[]) {
    MutableStatics.parse("Jan 1, 2000");
  }
}
```

FORMAT is mutable

...and this mutates it outside synchronization

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[]) {
    MutableStatics.parse(“Jan 1, 2000”);
  }
}
```

Synchronization

```
private int myField;

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

DO NOT synchronize on null

```
MyObject obj = null;

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

NullPointerException!

DO NOT change instance

```
MyObject obj = new MyObject();

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

No longer synchronizing on the same object!


FAIL
failblog.org

DO NOT synchronize on string literals

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

What is the scope of LOCK?

DO NOT synchronize on autoboxed instances

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

What is the scope of LOCK?

DO NOT synchronize on ReentrantLock

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

Probably not what you meant here

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

Probably should be this instead

What should I lock on?

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

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

Visibility

Inconsistent Synchronization

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

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

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

Modified under synchronization

Read without synchronization

Double-checked locking

```
public final 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 final 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;
  }
}
```

Racy single-check

```
public final class String {
  private int hash;                // default to 0
  private final char[] value;      // immutable

  public int hashCode() {
    int h = hash;
    if(h == 0) {
      // ... compute value for h from data
      hash = h;
    }
    return h;
  }
}
```

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 final AtomicInteger count = new AtomicInteger();
   public int next() {
      return count.getAndIncrement();
   }
}
```

Really atomic by encapsulating multiple actions

Composing atomic actions

```
public Object putIfAbsent(
    Hashtable table, Object key, Object value) {
        Hashtable is thread-safe
    if(table.containsKey(key)) {
      // already present, return existing value
      return table.get(key);
    } else {
      // doesn't exist, create and return new value
      table.put(key, value);
      return value;
    }
}
```

{ READ

{ READ

{ WRITE

Composing atomic actions

```
public Object putIfAbsent(
    Hashtable table, Object key, Object value) {
        Hashtable is thread-safe
    if(table.containsKey(key)) {
      // already present, return existing value
      return table.get(key);
    } else {
      // doesn't exist, create and return new value
      table.put(key, value);
      return value;
    }
}
```

READ

READ

WRITE

Participate in lock

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

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

Protect with synchronization

Encapsulated compound actions

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

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

Intentionally left blank.

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();
  }

  // Cleaner calls 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;
  }
}
```

Coordination

- Threads
- wait/notify

Threads

- DO NOT:
  - Call Thread.stop()
  - Call Thread.suspend() or Thread.resume()
  - Call Thread.destroy()
  - Call Thread.run()
  - Use ThreadGroups

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.

Update condition!

Performance

- Deadlock
- Spin wait
- Lock 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;
}
```

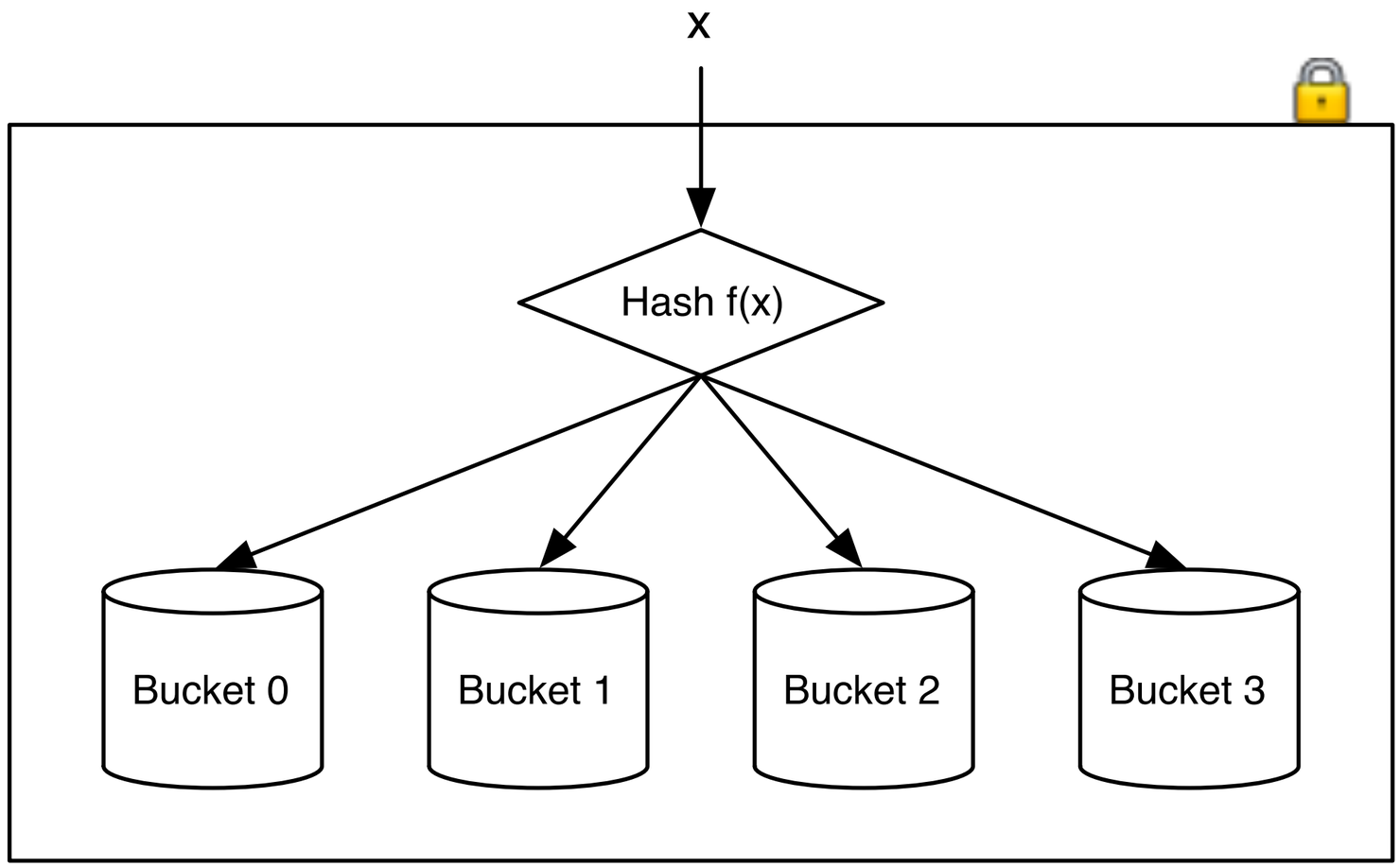
Spin on flag, waiting for change

Replace with wait/notify

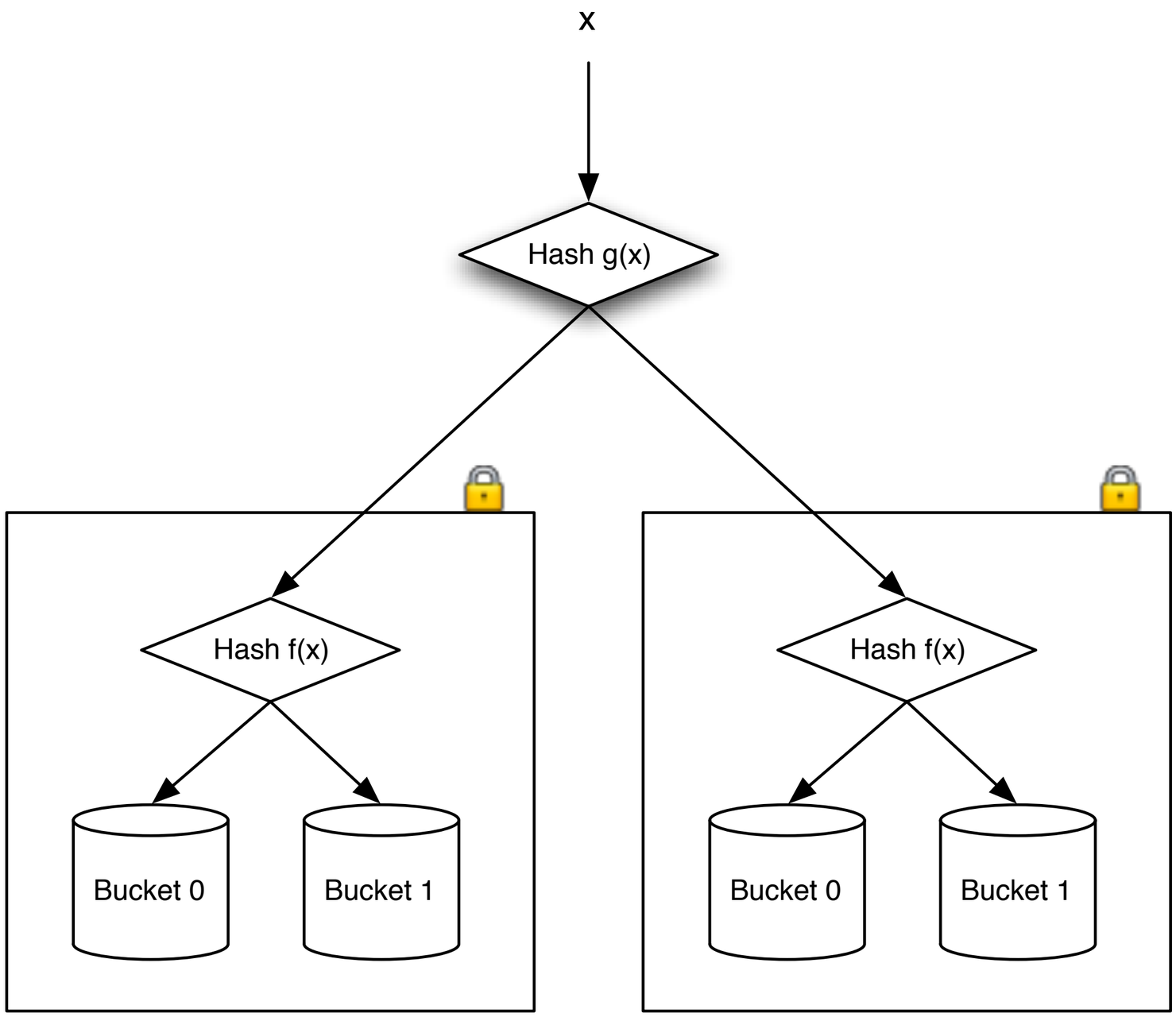
```
private final Object lock = new Object();
private boolean flag = false;

public void waitTillChange() {
   synchronized(lock) {
      while(! flag)
         lock.wait();
   }
}

public void change() {
   synchronized(lock) {
      flag = true;
      lock.notifyAll();
   }
}
```

Wait/notify is far more efficient than spin wait.

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
  }
}
```

Read of shared value without synchronization

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
  }
}
```

Read of shared value without synchronization

Non-atomic read of long value

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
  }
}
```

Read of shared value without synchronization

Non-atomic read of long value

Race condition if reading stat and clearing - could be compound action

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 of shared value without synchronization

Non-atomic read of long value

Race condition if reading stat and clearing - could be compound action

JavaOne℠

Thank You

Alex Miller
Terracotta

Blog: http://tech.puredanger.com
Twitter: @puredanger