Parallel Programming Practice

Java Concurrency: Thread Safety

Susanne Cech Previtali
Thomas Gross

Last update: 2009-10-19, 09:27

Practical view on the memory model

Multiple threads share the same mutable shared variable without appropriate synchronization

- Program is broken
- Incorrectly synchronized program

How to fix it

- Don't share the variable
- Make the variable immutable (and initialize properly)
- Use synchronization whenever accessing the variable

Categorization of variables

	Local ⇒ stack	Shared ⇒ heap
Immutable	Constant values	final fields, Strings
Mutable	local variables, arguments ⇒ stack	

Today

Thread safety

- Atomicity
- Locking

Sharing objects

Thread safety

About state, but applied to code

Thread safe classes

- Class encapsulate its state

Thread safe programs

- May include not thread-safe classes

Definition

conforms to its specification

A class is *thread-safe* if

- it behaves correctly when accessed from multiple threads
- regardless of the interleaving of the execution of those threads
- with no additional synchronization on the part of the calling code

Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own

Goetz et al.: Java Concurrency in Practice, Chapter 2, p. 18.

Stateless Classes

Stateless classes are always thread-safe

- No fields
- References no fields from other classes
- Only *transient* state in local variables

```
@ThreadSafe                                                        ✓
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

Consider state addition

```
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;
    public long getCount() { return count; }
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count++;
        encodeIntoResponse(resp, factors);
    }
}
```

No happens-before ordering

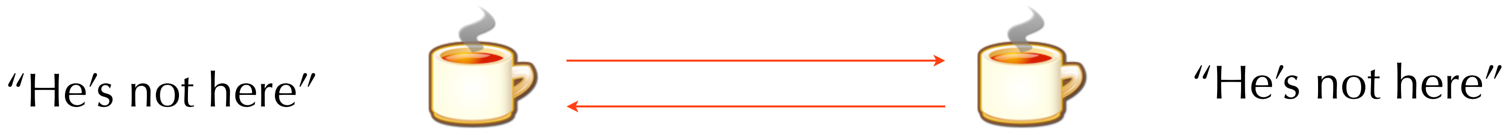
Atomicity

Race conditions

When correctness depends on the relative timing or interleaving of threads

- Right answer relies on lucky timing *(no happens-before ordering)*

Starbucks example



Check-then-act

- Stale (“old”) observation is used to decide what to do next
- State change in between

Kinds of race conditions

Read-modify-write operation

- Increment operation

Check-then-act operations

- Lazy initialization

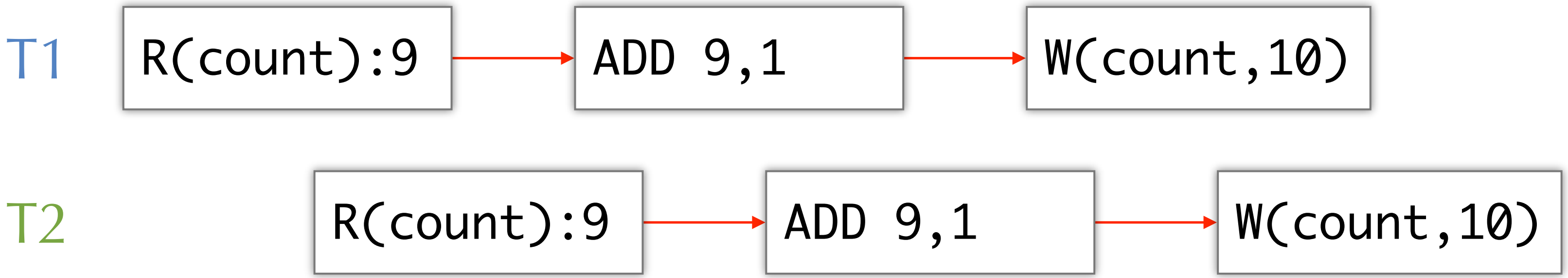
Read-modify-write operations

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;
    public long getCount() { return count; }
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count++; // read-modify-write operation
        encodeIntoResponse(resp, factors);
    }
}
```

T1 and T2 may write the same value

Problem: Lost updates

Increment operation not atomic



Read-modify-write operations

- Define a a transformation of an object's state in terms of its previous state
- `counter++;`
  - Know its previous value *and* make sure no one else changes/uses the value while you are updating

Check-then-act operations

Lazy initialization

- To defer initialization until the object is needed
- To ensure that it is initialized only once

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;
    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

T1 and T2 may receive two different objects

Atomic operations

Operations A and B are atomic with respect to each other if

- from the perspective of T_A when T_B executes B
- either all of B has executed or none of it has

An atomic operation is one that

- Is atomic with respect to all operations, including itself, that operate on the same state

Compound actions

Compound actions

- Sequences of operations that must be executed atomically to remain thread-safe

Examples

- Read-modify-write operations
- Check-then-act operations

Atomicity for compound actions

Mechanisms

- Atomic variable classes ($\geq$ Java 1.5)
- Locking
- Synchronized

Example fixed

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private AtomicLong count = new AtomicLong(0);
    public long getCount() { return count.get(); }
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet(); // incr. and return current value
        encodeIntoResponse(resp, factors);
    }
}
```

✓

Atomic variable classes

Atomic variable classes

Package `java.util.concurrent.atomic`

- Lock-free and thread-safe
- Extension of `volatile` values, fields, and array elements
- Conditional update operation

```
boolean compareAndSet(expectedValue, updatedValue) {
    if (this.value == expectedValue) {
        this.value = updatedValue;
        return true;
    }
    return false;
}
```

Pseudo code!

atomic operation

Categorization of classes

Single value classes

- `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, `AtomicReference`

Field updater classes

- `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater`, `AtomicReferenceFieldUpdater`

Array classes

- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray`

Markable classes

- `AtomicMarkableReference`, `AtomicStampedReference`

1 Single value classes

Reads and writes to a single variable

```
X get()
set(newValue)
compareAndSet(expect, update)
weakCompareAndSet(expect, update)
```

- Similar to `compareAndSet()`
- More efficient in the normal case
- May fail for no apparent reason
- Repeated invocation will eventually succeed

Utility methods

- For `AtomicLong` and `AtomicInteger`

Memory effects of single value classes

Method	Has memory effect of
`get()`	volatile read
`set()`	volatile write
`weakCompareAndSet()`	ordered with other ops on variable, non-volatile access
read-and-update operations	volatile read and volatile write

All single value classes

```
compareAndSet()
```

AtomicLong, AtomicInteger

```
addAndGet(), getAndAdd()
decrementAndGet(), getAndDecrement()
incrementAndGet(), getAndIncrement()
```

2 Field updater classes

“Wrappers” around volatile field

- Reflection-based
- Compare-and-set operations for specific class-field pair
- Several fields of the same node are independently subject of atomic updates
- Used inside Java library

Usage

- Occasionally need atomic get/set operations

Java library example

`java.io.BufferedInputStream`

type holding the updatable field

field type

```
protected volatile byte[] buf;
static AtomicReferenceFieldUpdater<BufferedInputStream, byte[]>
    bufUpdater = AtomicReferenceFieldUpdater.newUpdater
    (BufferedInputStream.class, byte[].class, "buf");
```

class holding the field | *class of the field* | *field name*

```
if (bufUpdater.compareAndSet(this, buffer, null)) { ... }
```

object | *expect* | *update*

3 Atomic array classes

Array elements can be updated atomically

- `AtomicIntegerArray`
- `AtomicLongArray`
- `AtomicReferenceArray<E>`

`E`.. *base class of elements*
`int` *for AtomicIntegerArray*
`long` *for AtomicLongArray*

Some methods

`E get(int i)`
`boolean set(int i, E newVal)`
`E getAndSet(int i, E newVal)`
`boolean compareAndSet(int i, E expected, E update)`
`boolean weakCompareAndSet(int i, E expected, E update)`

4 Markable classes

`AtomicMarkableReference<V>`

- Objects internally "boxed" `[reference, boolean]` pairs
- Pairs can be updated atomically

`AtomicStampedReference<V>`

- Objects internally "boxed" `[reference, integer]` pairs
- Pairs can be updated atomically

When atomic classes are not enough

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
    private AtomicReference<BigInteger> lastNumber = new ...
    private AtomicReference<BigInteger[]> lastFactors = new ...
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get())
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);          // must be updated
            lastFactors.set(factors);   // atomically
            encodeIntoResponse(resp, factors);
        }
    }}
```

Locking

Guarding state with locks

Make compound action atomic by

- Holding a lock for the *entire* duration of the compound action
- *All* accesses of the variable with the *same* lock
  - reads and writes

A variable guarded by a lock

Intrinsic locks

Only one thread at a time can execute a block of code guarded by a given lock

- Synchronized blocks execute atomically with respect to one another
- No thread executing a synchronized block can observe another thread to be in the middle of a synchronized block guarded by the same lock

reference to an object that serves as lock

```
synchronized (m) {
    ...
}
```

block of code guarded by lock

Synchronized: poor performance

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    public void synchronized service
             (ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get())
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }}
```

Locks and super-calls

```
public class Widget {
    public synchronized doSmth() {
        ....
    }
}
```

What would happen if Java had not taken care about it?

⇒ *deadlock*

```
public class LoggingWidget extends Widget {
    public synchronized doSmth() {
        System.out.println("Logging: " + toString());
        super.doSmth();
    }
}
```

Java solution: Reentrant locks

A thread that tries to acquire a lock that it already holds *succeeds*

Intrinsic locks are reentrant

- Locks are acquired on a per-thread-basis
- (rather than on a per-invocation-basis)

Acquisition count for each lock

```
owner: null
count: 0
```

lock not owned

```
owner: A
count: 2
```

lock owned by A, acquired twice

Remarks

Acquiring a lock associated with an object

- Does *not* prevent other threads from accessing that object
- Prevents other threads from acquiring that same lock

It is up to you to create synchronization policies

Conventions: Synchronize everything

Synchronize any code path with object's intrinsic lock

- Encapsulate mutable state within an object

Example

- `java.util.Vector`

Discussion

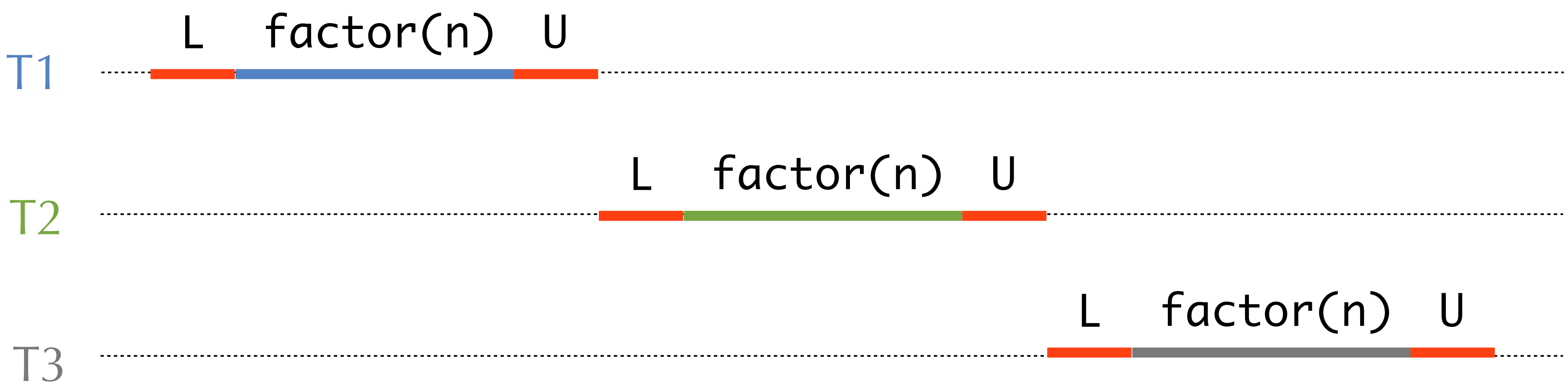
- Add a new method and forget to synchronize it
- Too little synchronization

```
if (!vector.contains(element))
    vector.add(element);
```

- Too much synchronization ⇒ poor concurrency

Poor concurrency

Solution `SynchronizedFactorizer` *(see Slide 21)*



Conventions: Specific locks

Guard variables individually with specific locks

Class invariants that involve more than one variables

- All such variables must be guarded by the *same* lock
- Example
  - `SynchronizedFactorizer` *(see Slide 32)*

Visibility!

```java
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            if (i.equals(lastNumber)) factors = lastFactors.clone();
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

✓

Today

Thread safety

- Atomicity
- Locking

Sharing objects

It is all about visibility

Volatile variables

Locking

Publication: objects are made visible

- Thread confinement --- do not publish
- Immutability --- do not synchronize
- Safe publication

Publication vs Escape

An object is *published* when

- it has been made available outside of its current scope
- How?
  - Store a reference where other code can access it
  - Return a reference from a non-private method
  - Pass a reference to a method in another class
- May break encapsulation

An object is *escaped* when

- It is published and should not have been published
- May break thread safety

Escaped objects

Problems with escaped objects

Consequences

- Any caller can modify object

Properties

- Publishing one object also publishes all its *reachable* objects
  - Follow chain of references
  - "Alien" method calls of a class C with object as argument
    - Methods in other classes
    - Overridable methods of C

How to escape

Store a reference in a public static field

Return a reference from a non-private method

Publish an inner class instance ⇒ publish `this`

Example escaped objects

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>;
}
```

✗

```
class UnsafeStates {
    private String[] states = new String[] { “A”, “B”, ... };

    public String getStates() { return states; }
}
```

✗

Proper construction

Object is *not* properly constructed if `this` escapes during construction

- Consistent state only after constructor returns

Do not

- Start a thread in the constructor
- Call a overridable method in the constructor

Escaped This reference to Inner classes

```java
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }

    void doSomething(Event e) {
    }
}
```

Implicitly publishes ThisEscape instance

- Generated inner classes contains a reference to the outer class

Fixed example using factory method

```
public class SafeListener {
    private final EventListener listener;
    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }
    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
    void doSomething(Event e) { }
}
```

✓