CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS [THREAD SAFETY]

Shrideep Pallickara
Computer Science
Colorado State University

February 14, 2017

Topics covered in this lecture
- Sharing & Composing Objects
- Making a class thread-safe
- Multivariable invariants and thread-safety
- Adding functionality to a thread-safe class

Pitfalls of over synchronization
- Number of simultaneous invocations?
  - Not limited by processor resources, but is limited by the application structure
  - Poor concurrency

Antidote for poor concurrency
- Control the scope of the lock
  - Too large: Invocations become sequential
  - Don’t make it too small either
    - Operations that are atomic should not be in a synchronized block

Sharing Objects

February 14, 2017

CS455: Introduction to Distributed Systems [Spring 2017]
Dept. Of Computer Science, Colorado State University
What we will be looking at

- Techniques for sharing and publishing objects
- Safe access from multiple threads
- Together with synchronization, sharing objects lays foundation for thread-safe classes

Synchronization

- What we have seen so far:
  - Atomicity and demarcating critical sections
  - But it is also about memory visibility
    - We prevent one thread from modifying object state while another is using it
    - When state of an object is modified, other thread can see the changes that were made

Publication and Escape

- Publishing an object
  - Makes it available outside current scope
    - Storing a reference to it, returning from a non-private method, passing it as argument to another method
- Escape
  - An object that is published when it should not have been

Pitfalls in publication

- Publishing internal state variables
  - Makes it difficult to preserve invariants
- Publishing objects before they are constructed
  - Compromises thread-safety

Most blatant form of publication

- Storing a reference in a public static field
  - public static Set<Secret> knownSecrets;
  - public void initialize() { knownSecrets = new HashSet<Secret>(); };
  - If you add a Secret to knownSecrets
  - You also end up publishing that Secret

Allowing internal mutable state to escape

public class PublishingState {
  private String[] states = new String[] { "AK", "AL", ... };
  public String[] getStates() {return states;};

  \* states has escaped its intended scope
  \* What should have been private is now public
  \* Any caller can modify its contents
Another way to publish internal state

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

- When EventListener is published, it publishes the enclosing ThisEscape instance
- Inner class instances contain hidden reference to enclosing instance

Safe construction practices

- An object is in a predictable, consistent state only after its constructor returns
- Publishing an object within its constructor?
  - You are publishing an incompletely constructed object
  - Even if you are doing so in the last line of the constructor
- **RULE:** Don't allow this to escape during construction

A common mistake is to start a thread from a constructor

- When an object creates a thread in its constructor
  - Almost always shares its this reference with the new thread
    - Explicitly: Passing it to the constructor
    - Implicitly: The Thread or Runnable is an inner class of the owning object
- Nothing wrong with creating a thread in a constructor
  - Just don't start the Thread
  - Expose an initialize() method

Thread confinement

- Language has no means of confining an object to a thread
- Thread confinement is an element of program's design
  - Enforced by implementation
- Language and core libraries provide mechanisms to help with this
  - Local variables and the ThreadLocal class
Stack confinement

- Object can only be reached through local variables
- Local variables are **intrinsically confined** to the executing thread
  - Exist on executing thread’s stack
  - Not accessible to other threads

Thread confinement of reference variables

```java
public int loadTheArk() {
    SortedSet<Animal> animals;
    // animals confined to method don't let
    // them escape
    return numPairs;
}
```

If you were to publish a reference to `animals`, stack confinement would be violated.

ThreadLocal

- Allows you to associate a per-thread value with a value-holding object
- Provides set and get accessor methods
  - Maintains a separate copy of value for each thread that uses it
  - get returns the most recent value passed to set
    - From the currently executing thread

```java
private static ThreadLocal<Connection> connectionHolder = new ThreadLocal<Connection>() {
    public Connection initialValue() {
        return DriverManager.getConnection(DB_URL);
    }
};
```

Public static Connection getConnection() {
    return connectionHolder.get();
}

Using ThreadLocal for thread confinement

Each thread will have its own connection

When thread calls ThreadLocal.get for the first time?
initialValue() provides the initial value

Common use of ThreadLocal

- Used when a frequently used operation requires a temporary object
- Wish to avoid reallocating temporary object on each invocation
- `Integer.toString()`
  - Before 5.0 used ThreadLocal to store a 12-byte buffer for formatting result

Immutable Objects
Immutable objects

- State cannot be modified after construction
- All its fields are final
- Properly constructed
  - The this reference does not escape during construction

Design makes it impossible to modify after construction
The stooges reference is final
All object state reached through a final field

```
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();
    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }
    public boolean isStooge() {return stooges.contains(name);}
}
```

Safe publication of objects

- Storing reference to an object into a public field is not enough to publish that object safely

```
public Holder holder;
public void initialize() {
    holder = new Holder(42);
}
```

Holder could appear to be in an inconsistent state
Even though invariants may have been established by constructor

Thread may see a stale value first time it reads the field and an up-to-date value the next time

```
public class Holder {
    private int n;
    public Holder(int n) {this.n = n}
    public void assertSanity() {
        if (n != n) {
            throw new AssertionError("Statement is false");
        }
    }
}
```

Class at risk of failure if not published properly

```
public class Holder {
    private int n;
    public Holder(int n) {this.n = n}
    public void assertSanity() {
        if (n != n) {
            throw new AssertionError("Statement is false");
        }
    }
}
```

Composing Objects

- We don't want to have to analyze each memory access to ensure program is thread-safe
- We wish to take thread-safe components and compose them into larger components or programs
Basic elements of designing a thread-safe class

- Identify **variables** that form the object's state
- Identify **invariants** that constrain the state variables
- Establish a **policy** for managing concurrent access to the object's state

Synchronization policy

- Defines how object coordinates access to its state
- Without violating its invariants or post-conditions
- Specifies a combination of:
  - Immutability
  - Thread confinement
  - Locking

Looking at a counter

```java
public final class Counter {
    private long value=0;
    public synchronized long getValue() {
        return value;
    }
    public synchronized long increment() {
        if (value == Long.MAX_VALUE) {
            throw new IllegalStateException("Counter Overflow");
        }
        value++;
        return value;
    }
}
```

Making a class thread-safe

- Ensure that invariants hold under concurrent access
- We need to reason about state
- Object and variables have **state space**
  - Range of possible states
  - Keep small so that it is easier to reason about

Classes have invariants that tag certain states as valid or invalid

- Looking back at our **Counter** example
- The value field is a long
- The state space ranges from Long.MIN_VALUE to Long.MAX_VALUE
- The class places constraints on value
  - Negative values are not allowed

Operations may have post conditions that tag state transitions as invalid

- Looking back at our **Counter** example
- If the current state of Counter is 17
  - The **only** valid next state is 18
  - When the next state is derived from the current state?
    - Compound action
- Not all operations impose state transition constraints
  - For e.g. if a variable tracks current temperature? Previous state doesn’t impact current state
Constraints and synchronization requirements

- If certain states are invalid?
  - Underlying state variables should be **encapsulated**
    - If not, client code can put it in an inconsistent state
- If an operation has invalid state transitions?
  - It must be made **atomic**

Looking at a case where invariants constrain multiple state variables

```java
public class NumberRange {
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        if (i > upper.get())
            throw IllegalArgumentException("lower > upper!");
        lower.set(i);
    }

    public void setUpper(int i) {
        if (i < lower.get())
            throw IllegalArgumentException("upper < lower!");
        upper.set(i);
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

Problems with NumberRange

- Does not preserve invariant that constrains lower and upper
- The methods `setLower` and `setUpper` attempt this preservation
  - But they do so poorly!
  - They are check-then-act sequences that use insufficient locking that precludes atomicity

Multivariable invariants

- Related variables must be fetched or updated in an atomic operation
- Don’t:
  - Update one
  - Release and reacquire lock, and ...
  - Then update others
- The lock that guards the variables
  - Must be held for the duration of any operation that accesses them

State-dependent operations

- Objects may have state-based pre-conditions
  - E.g., cannot remove item from an empty queue
- In a single-threaded program
  - Operations simply fail
- In a concurrent program
  - Precondition may be true later because of the actions of another thread
State dependent operations:

- **Mechanisms**
  - `wait() / notify()`
  - Supported by the JVM and closely tied with intrinsic locking
  - Other possibilities
  - Use classes such as blocking queues or semaphores

State ownership:

- **Defining which variables form an object’s state**
  - We wish to consider only that which the object owns
- **Ownership**
  - Not explicitly specified in the language
  - **Element of program design**

State ownership: Encapsulation and ownership go together

- Object encapsulates the state it owns
  - Owns the state it encapsulates
- Owner gets to decide on the **locking protocol**
- If you publish a reference to a mutable object?
  - You no longer have exclusive control

Instance confinement

- Object may not be thread-safe
  - But we could still use it in a thread-safe fashion
- Ensure that:
  - It is accessed by only one thread
  - All accesses guarded by a lock

Confinement and locking working together

```java
public class PersonSet {
    private final Set<Person> mySet = new HashSet<Person>();
    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }
    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```
Looking at our previous example

- State of PersonSet managed by HashSet, which is not thread-safe
- But mySet is
  - Private
  - Not allowed to escape
  - Confined to PersonSet

But we have made no assumptions about Person

- If it is mutable, additional synchronization is needed
  - When accessing Person from PersonSet
- Reliable way to achieve this?
  - Make Person thread-safe
- Less-reliable way?
  - Guard Person objects with a lock
  - Ensure that clients follow protocol of acquiring appropriate lock, before accessing Person

Instance confinement is the easiest way to build thread-safe classes

- Class that confines it state can be analyzed for thread-safety
  - Without having to examine the whole program

Guarding state with a private lock

```java
public class PrivateLock {
    private final Object myLock = new Object();
    Widget widget; //guarded by myLock

    public void someMethod() { //Access and modify the state of the widget
        synchronized(myLock) {
        }
    }
}
```

Guarding state with a private lock?

- Doing so encapsulates the lock
  - Client code cannot acquire it!
- Publicly accessible lock allows client code to participate in its synchronization policy
  - Correctly or incorrectly
- Clients that improperly acquire an object’s lock cause liveliness issues
- Verifying correctness with public locks requires examining the entire program not just a class
The contents of this slide-set are based on the following references