Analysis for Safe Concurrency

Optional supplementary reading: *Assuring and Evolving Concurrent Programs: Annotations and Policy*

15-214: Principles of Software System Construction

Jonathan Aldrich    Charlie Garrod
public class Logger {
    private Filter filter;

    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
Example: java.util.logging.Logger

public class Logger {
    ... 
    private Filter filter;

    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess(); 
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}

Consider log() in isolation

21 November 2013
Analysis of Software Artifacts: Concurrency
Example: `java.util.logging.Logger`  

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger {
    private Filter filter;

    /** ...
     * @param newFilter a filter object (may be null)
     */
    public void setFilter(Filter newFilter) {
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
```

Consider class `Logger` in it's entirety!
Example: java.util.logging.Logger

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger {
    private Filter filter;

    /** ... */
    * @param newFilter a filter object (may be null)
    */
    public void setFilter(Filter newFilter){
        if (!anonymous) manager.checkAccess();
        filter = newFilter;
    }

    public void log(LogRecord record) {
        synchronized (this) {
            if (filter != null && !filter.isLoggable(record)) return;
        }
    }
}
```

Class Logger has a **race condition**.
Example: java.util.logging.Logger

```java
/** ... All methods on Logger are multi-thread safe. */
public class Logger {
  private Filter filter;

  /** ... */
  * @param newFilter a filter object (may be null)
  */
  public synchronized void setFilter(Filter newFilter) {
    if (!anonymous) manager.checkAccess();
    filter = newFilter;
  }

  public void log(LogRecord record) {
    synchronized (this) {
      if (filter != null && !filter.isLoggable(record)) return;
    }
  }
}
```

**Correction:** synchronize `setFilter()`
Problem: Race condition in class Logger

- **Race condition:**
  - A situation in which the result of computation is dependent on the sequence or timing of program events

- **Data race:** a common source of race conditions
  (From Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*)
  - Two threads access the same variable
  - At least one access is a write
  - No explicit mechanism prevents the accesses from being simultaneous
Race Condition Challenges

**Problem:** Race condition in class **Logger**

- Non-local error
  - Had to inspect whole class
    - Bad code invalidates good code
  - Could have to inspect all clients of class

- Hard to test
  - Problem occurs non-deterministically
    - Depends on how threads interleave
Races and Invariants

**Problem:** Race condition in class **Logger**

- Not all race conditions result in errors
- Error results when invariant is violated
  - **Logger invariant**
    - filter is not null at call following null test
  - **Race-related error**
    - race between write and dereference of filter
    - if the write wins the race, filter is null at the call
Problem: Race condition in class **Logger**

- Need to know *design intent*
  - *Should instances be used across threads?*
  - *If so, how should access be coordinated?*
    - Assumed **log** was correct: **synchronize** on **this**
    - Could be caller’s responsibility to acquire lock
      - **log** is incorrect
      - Need to check call sites of **log** and **setFilter**
Review: Avoiding Races

How would you make sure your code avoids race conditions?

• Keep some data local to a single thread
  – Inaccessible to other threads
  – e.g. local variables, Java AWT & Swing, thread state
• Protect shared data with locks
  – Acquire lock before accessing data, release afterwards
  – e.g. Java synchronized, OS kernel locks
• Forbid context switches/interrupts in critical sections of code
  – Ensures atomic update to shared state
  – e.g. many embedded systems, simple single processor OSs
• Analyze all possible thread interleavings
  – Ensure invariants cannot be violated in any execution
  – Does not scale beyond smallest examples
• Future: transactional memory
Lock-based Concurrency

- Associate a lock with each shared variable
  - Acquire the lock before all accesses
  - Group all updates necessary to maintain data invariant
  - Hold all locks until update is complete

- Granularity
  - Fine-grained locks allow more concurrency
    - Can be tricky if different parts of a data structure are protected by different—perhaps dynamically created—locks
  - Coarse-grained locks have lower overhead
JSure: Tool Support for Safe Concurrency
Races and Design Intent

**Problem:** Race condition in class **Logger**

- Need to know *design intent*
  - *Should instances be used across threads?*
  - *If so, how should access be coordinated?*
    - Assumed **log** was correct: *synchronize* on **this**
    - Could be caller’s responsibility to acquire lock
      - ⇒ **log** is incorrect
      - ⇒ Need to check call sites of **log** and **setFilter**
Models are Missing

• **Programmer design intent is missing**
  – Not explicit in Java, C, C++, etc
    • *What lock protects this object?*
      – “This lock protects that state”
    • *What is the actual extent of shared state of this object?*
      – “This object is ‘part of’ that object”

• **Adoptability**
  – Programmers: “Too difficult to express this stuff.”
  – Annotations in tools like Fluid: *Minimal effort* — concise expression
    • Capture what programmers are *already thinking about*
    • No full specification

• **Incrementality**
  – Programmers: “I’m too busy; maybe after the deadline.”
  – Tool design (e.g. Fluid): Payoffs early and often
    • Direct programmer utility — *negative marginal cost*
    • Increments of payoff for increments of effort
Capturing Design Intent

- What data is shared by multiple threads?
- What locks are used to protect it?
  - Annotate class: `@RegionLock("FL is this protects filter")`
Reporting Code–Model Consistency

- Tool analyzes consistency
  - No annotations $\Rightarrow$ no assurance
  - Identify likely model sites

- Three classes of results
  - Code–model consistency
  - Code–model inconsistency
  - Informative — Request for annotation
Fluid Demonstration: Locks
Incremental Assurance

Payoffs early and often to reward use

• **Reassure after every save**
  – Maintain model–code consistency
  – Find errors as soon as they are introduced

• **Focus on interesting code**
  – Heavily annotate critical code
  – Revisit other code when it becomes critical

• **Doesn’t require full annotation to be useful**
Fluid Demonstration: Aliasing, Inheritance, and Constructors
Analysis Issues: Aliasing

• Other pointers can invalidate reasoning
  – @Unique – can other threads access through an alias?

• Similar issues in other analyses, e.g. Typestate

```java
FileInputStream a = ...  
FileInputStream b = ...  
a.close()                // what if a and b alias?  
b.read(...)             // may read a closed file
```

• Solution from Fugue (Microsoft Research)
  – @NotAliased annotation indicates that b has no aliases
  – Therefore closing a does not affect b
  – Requires alias analysis to verify
  – Can sometimes be inferred by analysis
    • e.g. see Fink et al., ISSTA ’06
Capturing Design Intent

- *What data is shared by multiple threads?*
- *What locks are used to protect it?*
  - Annotate class: @RegionLock("FL is this protects filter")
- *Is this delegate object owned by its referring object?*
  - Annotate field: @Unique
- *Can an object escape to the heap during construction?*
  - Annotate constructor: @Unique("return")
Constructors
- Often special cases for assurance
- Fluid: can’t protect with “this” lock
  - But OK since usually not multithreaded yet
- Others
  - Invariants may not hold until end of constructor

Subtyping
- Subclass must inherit specification of superclass
- Example: `@Unique("return")` for Formatter
- Sometimes subclass extends specification
  - e.g. to be multi-threaded safe
  - requires care in inheriting or overriding superclass methods

Inheritance
- Representation of superclass may have different invariants than subclass
- super calls must obey superclass specs
  - e.g. call to Formatter constructor
Fluid Demonstration: Cutpoints, Aliasing
How Incrementality Works 1

- How can one provide incremental benefit with mutual dependencies?
How Incrementality Works 2

• How can one provide incremental benefit with mutual dependencies?
  • Cut points
    – Method annotations partition call graph
    – Can assure property of a subgraph
    – Assurance is contingent on accuracy of trusted cut point method annotations
Cutpoint Example: \texttt{@requiresLock}

- Analysis normally assumes a method acquires and releases all the locks it needs.
  - Prevents caller’s correctness from depending on internals of called method.

- Method can require the caller to already hold a certain lock: \texttt{@RequiresLock(BufLock)}
  - Analysis of method gets to assume the lock is held.
    - Doesn’t need to know about caller(s).
  - Analysis of caller checks for lock acquisition.
    - Still ignores internals of called method.
Capturing Design Intent

• What data is shared by multiple threads?

• What locks are used to protect it?
  – Annotate class: @RegionLock("FL is this protects filter")

• Is this delegate object owned by its referring object?
  – Annotate field: @Unique

• Whose responsibility is it to acquire the lock?
  – Annotate method: @RequiresLock(BufLock)
Concurrency: Summary

• Many ways to make concurrency safe
  – Single-threaded data
  – Locks
  – Disabled interrupts
  – Analysis of interleavings (simple settings)
  – Transactions (future)

• Design intent useful
  – Document assumptions for team
  – Aids in manual analysis
  – Enables automated analysis