Analysis for Safe Concurrency

Reading: *Assuring and Evolving Concurrent Programs: Annotations and Policy*

17-654/17-754: Analysis of Software Artifacts

Jonathan Aldrich

---

Example: java.util.logging.Logger

```java
public class Logger { ...
    private Filter filter;

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

Consider `setFilter()` in isolation
```

[Source: Aaron Greenhouse]
Example: java.util.logging.Logger

```java
public class Logger { 
    private Filter filter;

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

Consider log() in isolation

Concurrence

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!

Concurrence
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.**

Correction: synchronize `setFilter()`

Concurrent
Example: Summary 1

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

- **Race condition** defined:
  (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

Example: Summary 2

**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
Example: Summary 3

**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

Example: Summary 4

**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`
Software Disasters: Therac-25

- Delivered radiation treatment
- 2 modes
  - Electron: low power electrons
  - X-Ray: high power electrons converted to x-rays with shield
- Race condition
  - Operator specifies x-ray, then quickly corrects to electron mode
  - Dosage process doesn’t see the update, delivers x-ray dose
  - Mode process sees update, removes shield
- Consequences
  - 3 deaths, 3 serious injuries from radiation overdose


Thought Experiment

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
Thread Locality in the Java AWT

- Event thread
  - Started by the AWT library
  - Invokes user callbacks
    - e.g. to draw a window
- Rules
  - Can create a component from any thread
  - Once component is initialized, can only access from Event thread
  - To access from another thread, register a callback function to be invoked in the Event thread
- Many other GUI libraries have similar rules
  - Microsoft Windows Presentation Foundation: one thread per window
- Why (e.g. vs. locks)?
  - Simple: no need to track relationship between lock and state
  - Predictable: less concurrency in GUI
  - Efficient: acquiring locks is expensive
- Why not?
  - Less concurrency available

Thread Locality: Variations

- Read-only data structures
  - May be freely shared between threads
  - No changes to data allowed
- Ownership transfer
  - Initialize a data structure in thread 1
  - Transfer ownership of data to thread 2
    - Now thread 2 may access the data, but thread 1 may not
    - Transfer may be repeated
    - Note that transfer usually requires synchronization on some other variable
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

Deadlock

- Bank transfer
  - Debit one account and credit another
  - (broken) protocol: lock debit account, then credit account

- Deadlock scenario
  - Thread 1 acquires lock A
  - Thread 2 acquires lock B
  - Thread 2 attempts to acquire lock A and waits
  - Thread 1 attempts to acquire lock B and waits
  - Neither thread 1 nor thread 2 may proceed

- Deadlock definition
  - A set of threads that forms a cycle, such that each thread is waiting to acquire a lock held by the next thread

thread1() {
  lock(A); // protects X
  lock(B); // protects Y
  debit(X);
  credit(Y);
  unlock(B);
  unlock(A);
}
	hread2() {
  lock(B);
  lock(A);
  debit(Y);
  credit(X);
  unlock(A);
  unlock(B);
}
**Dealing with Deadlock**

- **Lock ordering**
  - Always acquire locks in a fixed order
    - Cycles impossible—both thread 1 and thread 2 will attempt to acquire A before B
    - Release locks in the opposite order

- **Detect cycles as they form**
  - Runtime system checks for cycles when waiting to acquire
    - Expensive in practice, but simplifies development
    - Force one thread in cycle to give up its lock
      - Typically the last thread, or the lowest priority

**Disabling interrupts/context switches**

- **Disable interrupts for critical sections of code**
  - Should be short, so that interrupts aren’t delayed too long
  - Must be long enough to update shared data consistently
  - Common in single-processor embedded systems

- **Why?**
  - Cheap, simple, predictable

- **Why not?**
  - Does not support true multiprocessor concurrency
  - Suspending interrupts can mean missing real time I/O deadlines
  - Like having a global lock: forbids concurrent access even to different data structures
Analyzing All Possible Interleavings

- **Race condition** defined:
  
  (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

```c
thread2() {
    unlock();
    access x;
}
thread1() {
    lock();
    write x;
    unlock();
}
```

Interleaving 1: **OK**
Analyzing All Possible Interleavings

thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
    read x
}

Interleaving 1: OK
Interleaving 2: OK

Analyzing All Possible Interleavings

thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
    read x
}

Interleaving 3: Race
Analyzing All Possible Interleavings

thread1() {
    read x;
}
thread2() {
    lock();
    write x;
    unlock();
}

Interleaving 1: OK
Interleaving 2: OK
Interleaving 3: Race
Interleaving 4: Race

• What
  • No race conditions
  • More important: data invariants always hold at appropriate program points

• Why?
  • You are implementing a new synchronization primitive
  • Building on top of other synchronization mechanisms is too expensive

• Why not?
  • Does not scale to large bodies of code
  • Complex and error prone
  • May not be portable, depending on memory model
  • No guarantee the result will be faster!
Transaction Memory

• Group update operations into a transaction
  • Goal: invariant holds after operations are complete
• Run-time system ensures update is atomic
  • i.e. updates are consistent with running complete transactions in a linear order
• Implementation
  • Track reads and writes to memory
  • At end, ensure no other process has overwritten cells that were read or written
  • Commit writes if no interference
  • Abort writes (with no effect) if interference observed

21 April 2008 Analysis of Software Artifacts: Concurrency 25

Transaction Memory

• Why?
  • Simpler model than others, therefore much easier to get right
  • No problem with deadlock
  • Allows more concurrency
  • Supports reuse of concurrent code

• Why not?
  • Overhead may be high
  • Still experimental

• My view: inevitable as concurrency becomes more common

21 April 2008 Analysis of Software Artifacts: Concurrency 26
Fluid: Tool Support for Safe Concurrency

Example: Summary 4

**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: @lock FL is this protects filter
Reporting Code–Model Consistency

• Tool analyzes consistency
  • No annotations ⇒ 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
  - @singleThreaded – can other threads access through an alias?
  - @aggregate ... into Instance – can the field be accessed through an alias that is not protected by the lock?
- 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: @lock FL is this protects filter
- **Is this delegate object owned by its referring object?**
  - Annotate field: @aggregate ... into Instance
- **Can this object be accessed by multiple threads?**
  - Annotate method: @singleThreaded
- **Can this argument escape to the heap?**
  - Annotate method: @borrowed this
Analysis Issues: Constructors, Inheritance

- **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: @singlethreaded 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: @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: @requiresLock FilterLock
  - 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: @lock FL is this protects filter

- Is this delegate object owned by its referring object?
  - Annotate field: @aggregate ... into Instance

- Whose responsibility is it to acquire the lock?
  - Annotate method: @requiresLock FL
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 (eventual) automated analysis

Questions?