Announcements

- Requirements due Thur, Mar 10th, 5pm

Outline

- What are data races?
- Preventing data races
- Detecting data races

Data Races. Example

- A banking example
  - Account modeled as a global variable "balance"
  - ATM withdrawal modeled as threads running
    \[ \text{balance} = \text{balance} - 1 \]
  - In reality the machine code is more like
    \[ \text{temp} = \text{balance} - 1 \]
    \[ \text{balance} = \text{temp} \]
    where \( \text{temp} \) is some thread-local variable

Data Races

- Data races are a multithreading bug
  - At least two threads access a shared variable
  - At least one of the threads writes the variable
  - The accesses are (potentially) simultaneous

- Races are usually undesirable
  - Source of nondeterminism
  - Program state depends on timing
  - Very hard to reproduce bugs

Data Races. Example

Consider the following interleaving:

\[
\begin{align*}
\text{ATM1} & : \\
\text{temp}_1 &= \text{balance} - 1 \\
\text{balance} &= \text{temp}_1 \\
\text{ATM2} & : \\
\text{temp}_2 &= \text{balance} - 1 \\
\text{balance} &= \text{temp}_2
\end{align*}
\]

- This is a data race on global "balance"
- It only happens on rare interleavings
How to Prevent Races

- Locks
  - Special objects with two states: locked, unlocked
  - A locked lock is owned by the thread that locked it
  - At most one thread can own the lock at one time
  - The other threads are blocked and waiting

- Example:
  ```
  lock(balanceLock)
  balance = balance - 1
  unlock(balanceLock)
  ```

Data Races (Cont.)

- Note: Not all races are bad
  - Just the vast majority are bad

- Example
  - Threads execute
    ```
    if (predicate) x = 1
    ```
  - Threads where test passes race to set x
    - But x will be 1 if any thread’s test is true
    - And it is Ok to set x to 1 several times!

Intentional Data Races

- Consider the (single-threaded) code
  ```
  if(f == null) { f = fopen("foo"); } 
  ```
  - What goes wrong in multithreaded code?

- Fix with locks:
  ```
  lock(L); if(f == null) { f = fopen("foo"); }; unlock(L)
  ```

- Optimization (with intentional race):
  ```
  if(f == null) {
    lock(L); if(f == null) { f = fopen("foo"); }; unlock(L)
  }
  ```

Bottom Line on Preventing Data Races

- Java includes language-level locks, but …

- Consider the code (from java.lang.StringBuffer)
  ```
  lock(buff); len = buff.length(); unlock(buff);
  ```

- Optimization (with intentional race):
  ```
  lock(buff); c = buff.getChar(len - 1); unlock(buff);
  ```

- Do you see the problem?
  ```
  It is still very hard to program correctly with locks
  ```

Finding Data Races

- Fact: programs with data races will still be written, at least for a while

- Can we write tools that detect data races?

Happens Before

- Event A happens before event B if
  - B follows A in a single thread of control
  - A in thread a, B in thread b, event C such that
    - A happens in a
    - C is a synch event after A in thread a, and before B in b
    - B happens in b

- This is the guaranteed ordering of events
  - Events in separate threads without synchronization do not have a guaranteed order
Happens Before. Example

Thread 1
- lock(balanceLock)
- balance = balance - 1
- unlock(balanceLock)

Thread 2
- lock(balanceLock)
- balance = balance - 1
- unlock(balanceLock)

• Accesses to balance are ordered by happens-before!

Old Checking Algorithms

• First race detection tool based on happens before

• Sketch
  - Monitor all data references, synch operations
  - Watch for
    - Access of v in thread a
    - Access of v in thread b
    - With no intervening synch between a and b

Problem 1

• This is expensive

• At each access to shared data
  - Find the last access to data from another thread
  - See if there is intervening synchronization

• Requires per thread
  - List of accesses to shared data
  - List of synchronization operations

Problem 2

Thread 1
- balance = balance - 1
- lock(balanceLock)
- unlock(balanceLock)

Thread 2
- lock(balanceLock)
- unlock(balanceLock)
- balance = balance - 1

• Can miss races: must try many schedules!

A Different Approach

• Happens-before tools look for actual races
  - Moments in time when multiple threads access a shared variable without protection

• But actual races are hard to reproduce

• A different approach is to check invariants
  - Look for examples that violate invariants that might lead to races

A Race-Avoidance Discipline

• Shared variables are protected by locks

• Discipline:
  - Every access to a shared variable is protected by at least one lock
  - Any access to a shared variable unprotected by a lock is an error
Which Lock?

• How do we know which lock protects a variable?
  - The program may hold many unrelated locks
  - Linkage between locks and shared variables undeclared

• Issue
  - Like any instrumentation approach, we don’t have the resources to do intensive analysis during execution

Problem 1: Uninitialized Data

• Data often initialized by one owner
  - No need to lock at this time

• How do we know when initialization is done?
  - Answer: We don’t
  - But, we can tell when the value is accessed by a second thread

Problem 2: Read Shared

• Once created, some data is only read
  - No need to lock read-only data

  • Idea: Don’t update locksets until at least
    - More than one thread has the value, and
    - At least one is writing to the value

Locksets

• Idea 1: Infer the locks
  • Observation: It must be one of the locks held at the time of access

  Initialize $C(v)$ to the set of all locks (for each $v$)
  On access to $v$ by thread $t$
  $C(v) \leftarrow C(v) \cap \text{locks\_held}(t)$;
  if $C(v) = \emptyset$ then print warning;

State Transitions

• Each shared value (memory location) is in one of four states:

New
Write Thread $t_1$
Exclusive $t_1$
Read/Write Thread $t_2$
Shared
Write any thread
Exclusive
Read Thread $t_3$
Read any thread
Shared/Modified

New Algorithm

• The algorithm is as before

  • But only locations in the Shared/Modified and Shared states have locksets inferred

  • Errors are reported only in Shared/Modified state
    - Allows initialization and read-shared behavior
Read-Write Locks

- Single writer, multiple reader locks
- Locks can be held either in write mode or in read mode
- Discipline: Some lock (a particular one) must be held
  - in write mode for all writes to a shared location
  - in either write mode or read mode for all accesses

Solution

- Refine computation of locksets to express single write exclusivity
  - For each read of a location, compute \[ C(v) \cap \text{locks\_held}(t); \]
  - For each write of a location, compute \[ C(v) \cap \text{write\_locks\_held}(t); \]

Implementation

- Done at the binary level
  - Could have been a source code tool
- Every memory word has a shadow word
  - 30 bits designated for the lockset key
  - Sets of locks represented by small integers in a hashtable
  - Depends on having not very many distinct sets of locks
  - 2 bits for state in the DFA

Results

- This works
  - Checking the discipline finds errors with few runs
  - Many imitators
- Eraser will miss many concurrency errors
  - See earlier example with java.lang.Stringbuffer
- Eraser is slow
  - 10-30X slowdown
  - Could be made faster with static analysis