

# Runtime Monitoring

## Lecture 4 CS295

### Outline

---

- Runtime monitoring of code
- Three examples:
  - Purify
  - Detecting data races
  - Detecting memory leaks

# Purify

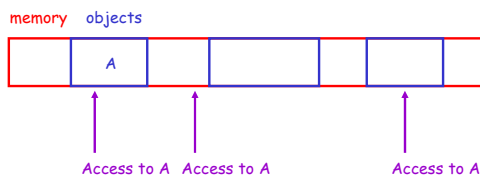
### The Problem

---

- C/C++ are not type safe
  - Neither the compiler nor the runtime system enforces type abstractions
- Possible to read or write outside of your intended data structure
  - Among other bad behaviors

### Picture

---



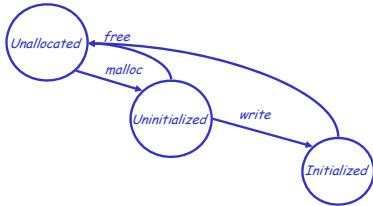
### The Idea

---

- Each byte of memory is in one of three states:
- Unallocated
  - Cannot be read or written
- Allocated but uninitialized
  - Cannot be read
- Allocated and initialized
  - Anything goes

## State Machine

Associate an automaton with each byte



Prof. Aiken CS 295 Lecture 4

7

## Instrumentation

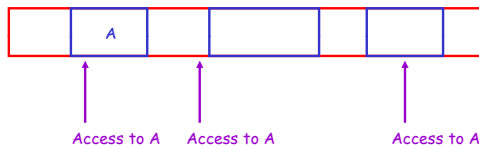
- Check the state of each byte on each access
- Binary instrumentation
  - Add code before each load and store
  - Represent states as giant array
    - 2 bits per byte of memory
- 25% memory overhead
  - Catches byte-level errors
  - Won't catch bit-level errors

Prof. Aiken CS 295 Lecture 4

8

## Picture

memory objects



Note: We can detect invalid accesses to red areas, but not to blue areas.

Prof. Aiken CS 295 Lecture 4

9

## Improvements

- We can only detect bad accesses if they are to unallocated or uninitialized memory
- So try to make all bad accesses be of those two forms

Prof. Aiken CS 295 Lecture 4

10

## Red Zones

- Leave buffer space between allocated objects that is never allocated
  - The "red zone"
- Guarantees that walking off the end of an array accesses unallocated memory

Prof. Aiken CS 295 Lecture 4

11

## Aging Freed Memory

- When memory is freed, do not reallocate immediately
  - Wait until the memory has "aged"
- Helps catch dangling pointer errors
- Red zones and aging are easily implemented in the malloc library

Prof. Aiken CS 295 Lecture 4

12

## Summary

---

- These ideas work well in practice
- Widely used
  - E.g., Purify
  - Other systems use similar approaches
- Illustrates ideas of dynamic analysis
  - Runtime instrumentation of program
  - Willing to pay large overhead for systematic checking

## Data Races

## 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 (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

## 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 *A* and before *b* in *B*
    - *B* happens in *b*
- This is the natural partial order on events

## Pre-Eraser

---

- First race detection tools based on *happens before*
- Sketch
  - Monitor all data references, synch operations
  - Watch for
    - Access of *v* in thread 1
    - Access of *v* in thread 2
    - With no intervening synch between 1 and 2

## Problem 1

---

- This is expensive
- Requires per thread
  - List of accesses to shared data
  - List of synchronization operations

## Problem 2

---

- False negatives
  - Can miss races
  - Needs to be tested with many schedules
- Thread 1
  - $y = y + 1$
  - lock(m)
  - unlock(m)
- Thread 2
  - lock(m)
  - unlock(m)
  - $y = y + 1$

## A Different Approach

---

- Happens-before tools look for actual races
  - Moments in time when multiple threads access a shared variable without protection
- A different approach is to check invariants
  - Look for examples that violate invariants that might lead to races

## The 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

## 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  $t$

$C(v) \sim C(v) \_ \text{locks\_held}(t);$

if  $C(v) = \{ \}$  then print warning;

## Problems

---

- This doesn't quite work
- We need to deal with
  - Uninitialized data
  - Read-Shared Data
  - Read-Write Locks

## 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

## Read Shared

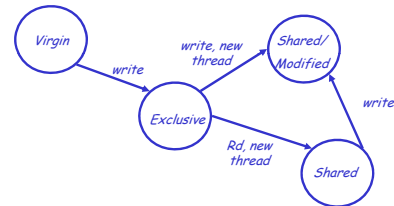
---

- 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
  - At least one is writing to the value

## State Transitions

---

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



## New Algorithm

---

- The algorithm is as before
- But only locations in the shared-modified state have locksets inferred
- None of the other cases requires checking

## Read-Write Locks

---

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

## Solution

---

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

## 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 is slow
  - 10-30X slowdown
  - Could be made faster with static analysis
- Many Eraser-like tools available now

## Memory Leaks

## Another Class of Errors: Memory Leaks

---

- Memory leaks are at least as serious as memory corruption errors
- Also very difficult to find
  - Manifest only over hours, days, weeks
  - Often persist in production code

## The Basic Idea

---

- We can find many memory leaks using techniques borrowed from garbage collection
- Any memory with no pointers to it is leaked
  - There is no way to free this memory
- Run a garbage collector
  - But don't free any garbage
  - Just detect the garbage
  - Any inaccessible memory is leaked memory

## Issues with C/C++

- It is sometimes hard to tell what is inaccessible in a C/C++ program
- Cases
  - No pointers to a malloc'd block
    - Definitely garbage
  - No pointers to the head of a malloc'd block
    - Maybe garbage
  - Pointers to the head of a malloc'd block
    - Not garbage by usual definition

Prof. Aiken CS 295 Lecture 4

37

## Summary of Basic Leak Detection

- From time to time, run a garbage collector
  - Use mark and sweep
- Report areas of memory that are definitely or probably garbage
  - Need to report who malloc'd the blocks originally
  - Store this information in the red zone between objects
- Used in Purify

Prof. Aiken CS 295 Lecture 4

38

## A Limitation

- Only finds leaks to unreachable objects
- Doesn't cover leaks in languages with GC
  - Retaining data structures longer than needed
  - In practice, also a serious source of leaks
  - Especially in Java . . .
- Idea
  - look for objects not accessed for a "long time"

Prof. Aiken CS 295 Lecture 4

39

## Outline

- For each object
  - Track it from the moment it is allocated
  - Record the time of the last access
    - Read or write
  - Discard information when object is deallocated
- Periodically
  - Scan all objects
  - Warn about objects unused for a "long time"

Prof. Aiken CS 295 Lecture 4

40

## Problem 1: Tracking All Objects

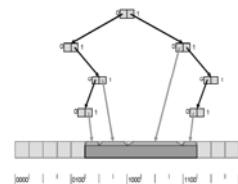
- How do we
  - Maintain information about each object
  - Without interfering with the running program
  - Standard problem in dynamic analysis
- Solution 1
  - Add extra fields to objects
  - Requires compiler support
- Solution 2
  - Keep information "on the side"
  - Use pointer to object as the key

Prof. Aiken CS 295 Lecture 4

41

## Solution

- Search tree
  - Paths are addresses
  - Each edge = 1 bit
  - Leaves are objects
- Advantage
  - No space wasted on unused addresses
  - Supports interior pointers
- Disadvantage
  - Lookup is slow
- Compare with Eraser



Prof. Aiken CS 295 Lecture 4

42

## Problem 2: Tracking Uses

- On every read/write to the heap
  - Update last access time
  - Time = count of all memory accesses
    - Not wall-clock time
  - This metric works better with interactive programs
    - No time passes when program is blocked on the user
- Again, this is very slow

## Solution

- Don't update access time on every read/write
  - Active data structures are accessed regularly
  - Checking 1/1000 accesses should do



- Note leaked structures are still never accessed after some point
- See paper for details on sampling

## Problem 3: What is a "Long Time"?

- Three options discussed in paper
- Never accessed
- Idle for time  $N$
- Idle for  $((\text{time of last access}) - (\text{time allocated})) * N$

## Putting It All Together

- Want to report most important leaks
  - Also the most likely leaks
- Cost of leaked byte  $B = B$ 's idle time
- Cost of an allocation site  $A =$   
Sum of the cost of all leaked bytes allocated at  $A$
- This is the *drag*
  - Rank allocation sites by drag

## Results

- Found a number of leaks in real applications
- Could run applications for a long time
  - Time overhead of instrumentation only 5-10%
  - Space overhead 10-80%

## Summary: Issues in Dynamic Analysis

- Inferring high-level properties from observations of concrete events
  - Which lock guards a location
  - When an object has been leaked
  - Note: Important to think about what property to infer!
- Keeping state per object
  - Typically, keep it on the side
- Performance
  - Slow by default
  - Sampling one generic way to improve performance