Eraser

- Problem: race conditions exist in many (industrial-strength) programs
  - Important: programs can crash, end up in inconsistent states
- Goal: execute program and find race conditions after the fact
  - i.e., a debugging tool
- Solution:
  - Dynamically determine race condition using a lock model
  - Report problem locations in code

Overview

- Supports lock-based multithreaded programs
  - Only consider Acquire and Release operations
    - Lock is always either free or busy, and if busy there is exactly one “owner” thread
    - Eraser does not allow general semaphores
- Race condition definition:
  - Multiple threads access a shared variable, outside of synchronization, and at least one thread writes
    - Same definition from class

Overview, cont.

- What is the universe of possible ways to find data races?
  - Monitors (doesn’t find, but potentially eliminates all possibility of races)
    - Why not?
  - Static analysis
    - What’s the problem here? The benefit?
  - Dynamic analysis
    - Eraser
    - Happens-before

Monitors

- Good: statically eliminates races
- Bad: dynamic data structures
  - Can argue about this one

Happens-Before

- Definition of happens-before relation
  - Partial ordering of executions of different threads, subject to these rules:
    - Within a thread, all statements are ordered by happens before by their sequential ordering.
    - Between threads, if thread A accesses a synchronization object (in this paper, a lock), and then thread B does, A's access happens before B's access
    - Happens-before is transitive
    - Any non-ordered events are called *concurrent*

- Happens-before finds races that could occur

Happens-Before Example

Thread 0 code	Thread 1 code
x = x + 1	Lock(L)
Lock(L)	y = y + 1
y = y + 1	Unlock(L)
Unlock(L)	x = x + 1

Happens-Before Example---Race Detected

Thread 0 code	Thread 1 code
	Lock(L)
	y = y + 1
	Unlock(L)
	x = x + 1
x = x + 1	
Lock(L)	
y = y + 1	
Unlock(L)	

Happens-Before Example--- No Race Detected

Thread 0 code

x = x + 1

Lock(L)

y = y + 1

Unlock(L)

Thread 1 code

Lock(L)

y = y + 1

Unlock(L)

x = x + 1

Static Analysis

- Analyze code, looking for race conditions
  - Good: may find race conditions that might not manifest in a particular program execution
    - Also, may be able to find (narrow down) potential race regions, then use dynamic analysis
  - Bad: Very hard or very conservative. Does not generally work well with pointers.

Static Analysis: Race Detected

Thread 0 code	Thread 1 code
x = x + 1	
Lock(L)	
y = y + 1	
Unlock(L)	
	Lock(L)
	y = y + 1
	Unlock(L)
	x = x + 1

Static Analysis: Race Almost Surely not Detected

Thread 0 code	Thread 1 code
***p = *p + 1**	
Lock(L)	
y = y + 1	
Unlock(L)	
	Lock(L)
	y = y + 1
	Unlock(L)
	***q = *q + 1**

What if p and q both point to x?

Sample code for Eraser

Thread 0 code

Lock(mu1)

v = v + 1

Unlock(mu1)

Thread 1 code

Lock(mu2)

v = v + 1

Unlock(mu2)

Eraser: Basic Algorithm
Each shared variable has a candidate lockset

Program	Locks Held	C(v)
(init)	nothing	mu1, mu2
T1:Lock(mu1)	mu1	mu1, mu2
T1:v = v + 1	mu1	mu1
T1:Unlock(mu1)	nothing	mu1
T2:Lock(mu2)	mu2	mu1
T2:v = v + 1	mu2	empty (!!)

Problems with simple algorithm

- Initialization (single thread)

```
main( ) {
    x = 4; x = x+1;   // Simple alg. flags an error
    thread_create( );
}
```

- Read sharing
  - Two or more threads accessing a variable, all reading
    - Eraser basic algorithm is on *access* (read or write)
    - Without changing this, read sharing would be disallowed

Initialization and Read Sharing

- Always start in init state (on first access)
- Proceed to exclusive state on a write
- From exclusive:
  - Same thread accesses: stay in exclusive
  - New thread reads: go to shared
  - New thread writes: go to shared-modified
- From shared:
  - New thread writes: go to shared-modified

Exclusive: don’t run Eraser alg

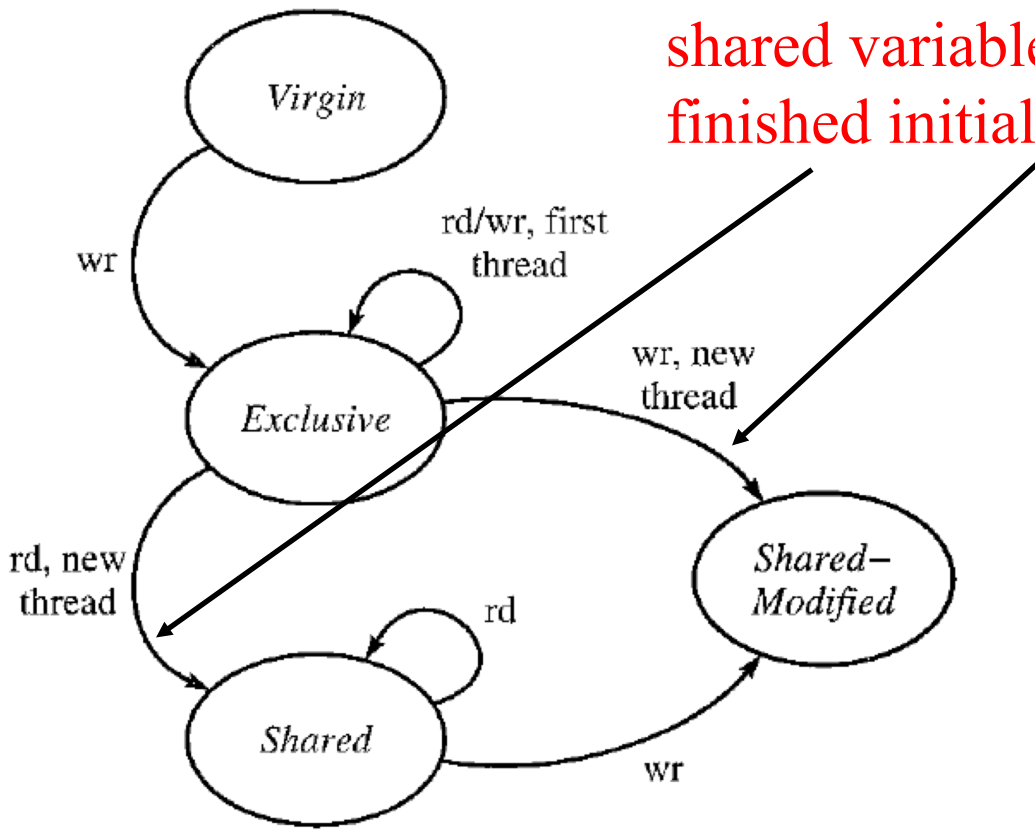
-- possible problem here---why?

Shared: run algorithm but do not flag errors

Shared-Modified: run algorithm and flag errors

Eraser State Machine courtesy of paper by Savage et al.

Problem here---can expose shared variable before finished initializing



Implementation

- Use a binary translation system
  - Instrument every load and store of a non-stack variable
  - Not clear how they determine current thread
    - Could be done by a call to the thread's getMyId
  - Each memory location is associated with an index into a hash table of different candidate lock sets
    - Shadow word per memory word (2x overhead)

Performance

- It's terrible.

Annotations

- When the Eraser algorithm fails, allow user to annotate
  - Memory re-use
    - User manages own memory; Eraser doesn't know about malloc/free
  - Private locks
    - User rolls their own locks; Eraser has no idea
  - Benign races
    - Race conditions that are "ok".

More on Benign Races

```
Acquire(L)
if (localmax > globalmax) {
  globalmax = localmax
}
Release(L)
```

Relatively common code pattern

```
if (localmax > globalmax) {
  Acquire(L)
  if (localmax > globalmax)
    globalmax = localmax
  Release(L)
}
```

Legal rewrite, but technically a race condition

Case Studies

- Found bugs in production software
- Found benign races also