Principles of Concurrency and Parallelism

Lecture 11: Data Races

4/12/12

Data Race

- A data race occurs when two concurrently executing threads access a shared variable and when:
  - at least one of the accesses is a write
  - there is no explicit mechanism used to prevent the accesses from being simultaneous
- Meaning of programs with data races depends upon interleaving of thread executions.
  - Sometimes this is ok (when?)
  - Usually, it is not

Example

```
public class Example extends Thread {
  private static int cnt = 0;  // shared state
  public void run() {
    int y = cnt;
    cnt = y + 1;
  }
  public static void main(String args[]) {
    Thread t1 = new Example();
    Thread t2 = new Example();
    t1.start();
    t2.start();
  }
}
```

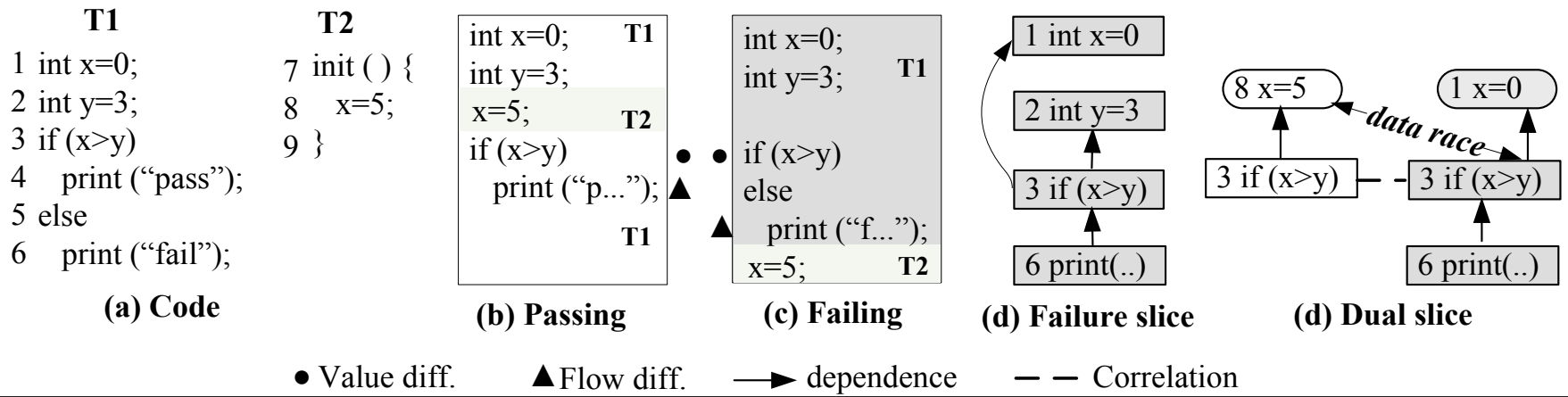
What can go wrong?

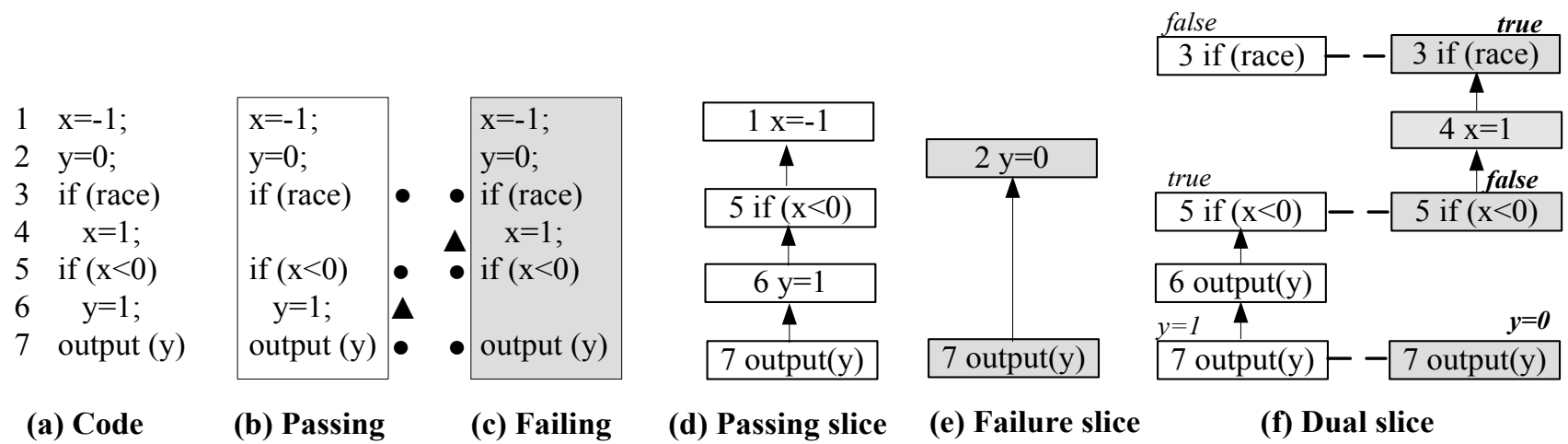
- A data race manifests as the result of an undesirable schedule or interleaving
- Key is to prevent such interleavings
  - Judicious use of locks or synchronization
  - Not always repeatable (Heisenbugs)
- How can we tell that a program does not have a data race?
  - Dynamic (monitor its execution)
  - Static (apply compile-time analysis)

Current State

- 2008 study (Lu et. al, ASPLOS'08)
- Examined 74 non-deadlock bugs in MySQL, Apache, Mozilla, OpenOffice
  - 1/3 of the bugs caused by violation of program order
  - 34% involved multiple variables
  - 92% can be triggered by enforcing certain schedules involving no more than 4 memory accesses
  - 73% could not be fixed by simply adding locks
- Concurrency bugs not easily repaired or detected

Explanation of Failures





How can data races be detected and prevented?

- Enforce the use of high-level language mechanisms
  - monitors, synchronized, etc.
  - Monitors: (Hoare 1974)
    - a group of shared variables along with procedures to access them.
    - all accesses protected by the same (anonymous) lock acquired and released upon entry/exit of the monitor
    - shared variable not visible outside monitor
    - lots of issues
      - dynamically allocated data, waiting, exceptions, nesting, ...

Dynamic Approaches

- Happens-before relation
  - partial order on events of all threads in a concurrent execution
  - Between threads, events are ordered according to the synchronization objects they access

Formally

- An interleaving is an execution in which
  - lock/unlock alternates correctly
  - each read sees the most recent write to the same location
    - *sequentially consistent* semantics
- Totally orders all actions
  - does not keep track of which actions take place in parallel

Formally

Definition [program order]: **program order**, $<_{po}$, is a total order over the actions of *the same thread in an interleaving.*

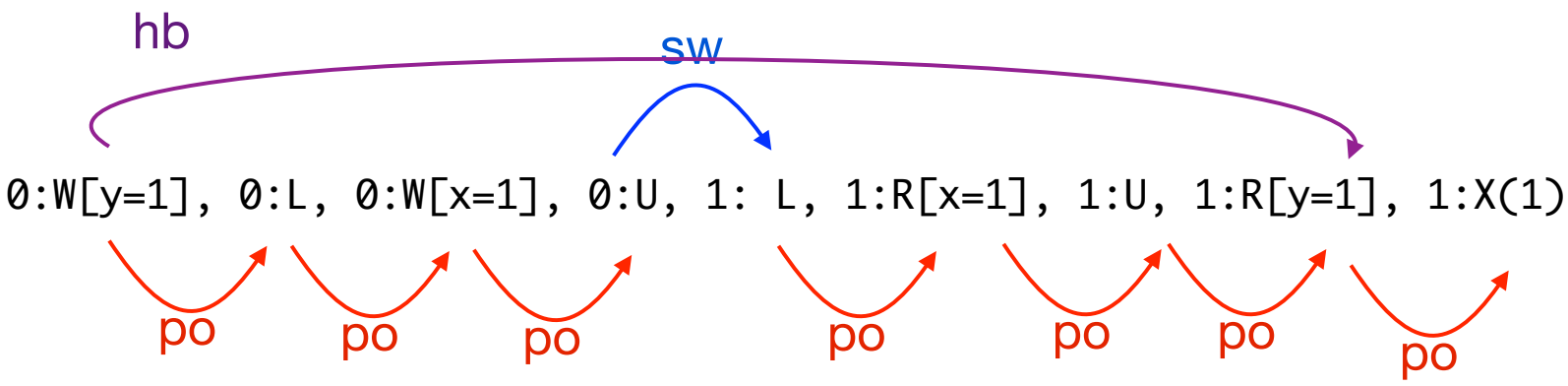
Definition [synchronises with]: in an interleaving *I*, index *i* **synchronises-with** index j, $i <_{sw} j$, if $i < j$ and $A(I_i) = U$ (unlock), $A(I_j) = L$ (lock).

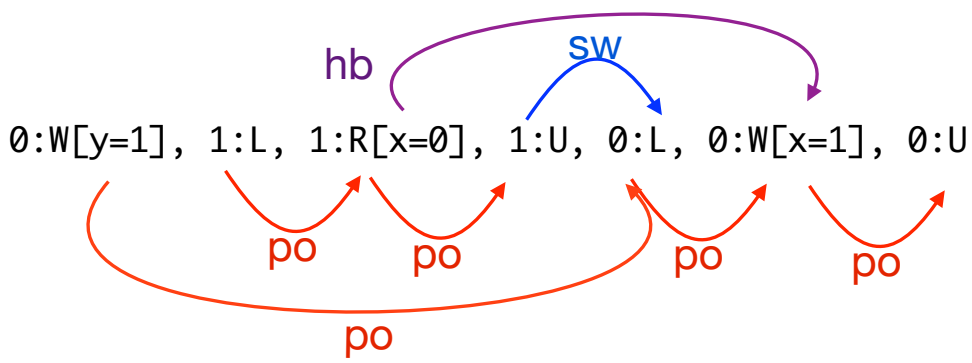
Definition [happens-before]: **Happens-before** is the transitive closure of program order and synchronises with.

Example

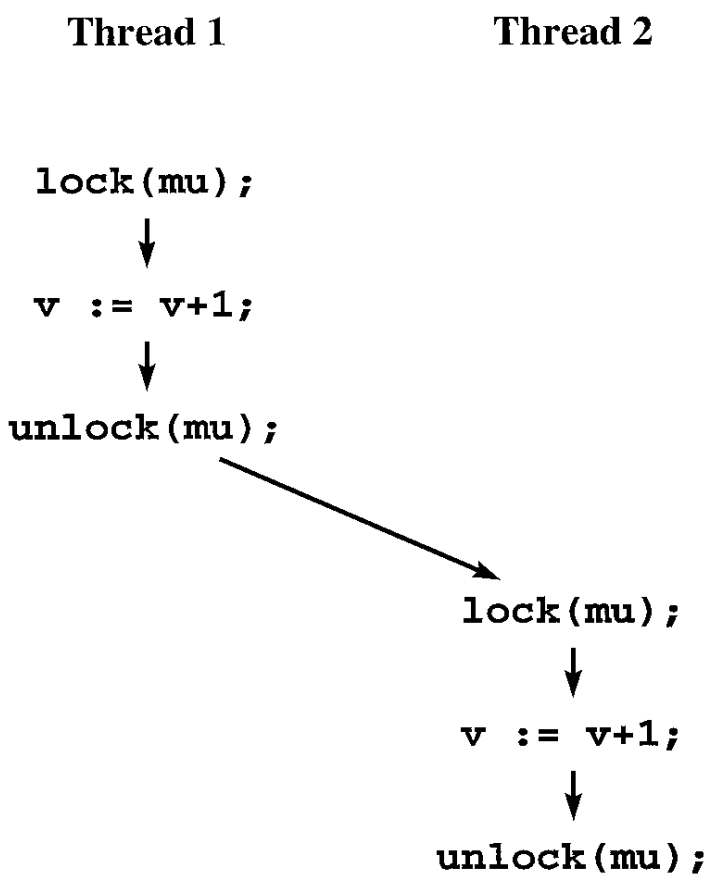
Thread 0	Thread 1

`*y = 1`
`lock();`
`*x = 1`
`unlock();`
	`lock();`
	`tmp = *x;`
	`unlock();`
	`if tmp = 1`
	`then print *y`





Happens-Before



Data Race

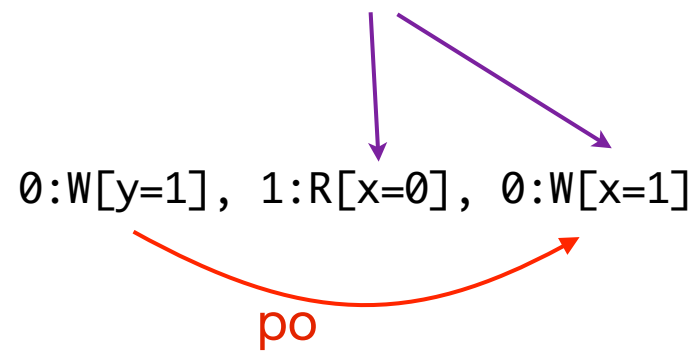
- If two threads access a shared variable, and the accesses are not ordered under a happens-before relation, then there is a potential data race.
- Dynamic detection of happens-before violations is difficult:
  - Require per-thread information about concurrent accesses to shared memory
  - But, there are techniques that can be used to alleviate this issue
    - vector clocks

Example

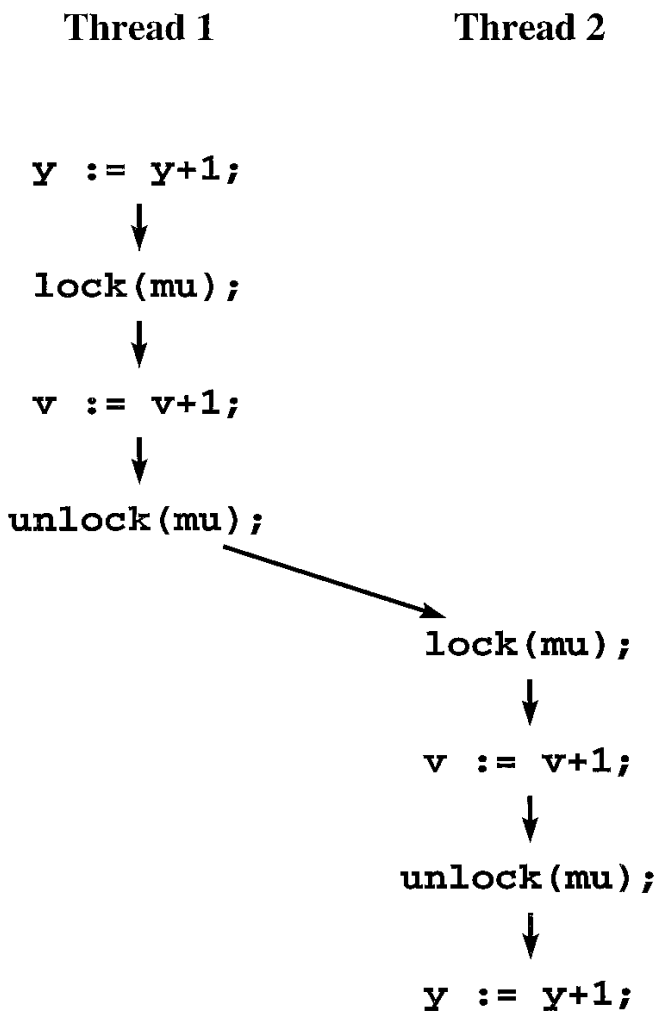
A racy program

Thread 0	Thread 1
`*y = 1` `*x = 1`	`if *x == 1` `then print *y`

Two conflicting accesses
not related by happens before.



Happens-Before



Using just happens-before, need a large number of test cases to catch the error.

Can we do better?

Eraser Lockset Algorithm

- To avoid data races, every shared variable must be protected by some lock.
- A dynamic tool must infer what these locks are
- For each shared variable v, maintain the set C(v) of candidate locks for v.
  - This set contains those locks that have protected v for the computation so far.
  - Initially, the set holds all possible locks.
  - When v is accessed, compute the interesection of C(v) with the current set of locks held by the thread
  - If the set is empty, there is no lock that consistently protects v

Example

Program	*locks_held*	*C(v)*
	{}	{mu1,mu2}
lock(mu1);		
	{mu1}	
v := v+1;		
		{mu1}
unlock(mu1);		
	{}	
lock(mu2);		
	{mu2}	
v := v+1;		
		{}
unlock(mu2);		
	{}	

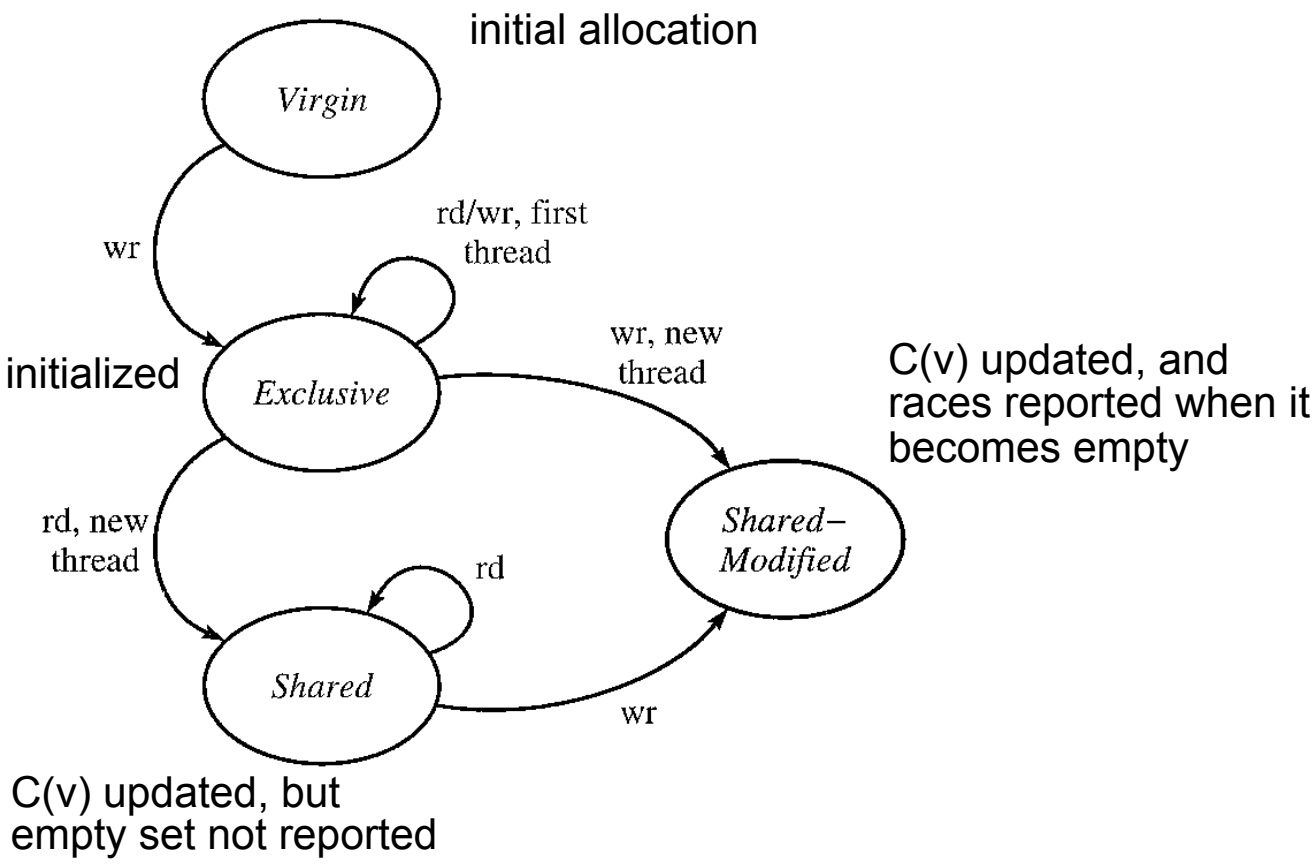
Improvements

- Common programming practices often violate locking discipline, but are still race free:
  - Initialization
  - Reading shared data
  - Read-write locks:
    - multiple readers, single (exclusive) writer

Initialization

- How can we tell when initialization is complete?
  - Assume initialization is complete if a variable is accessed by a second thread.
  - As long as a variable is only accessed by a single thread, reads and writes have no effect on the candidate lock set.
  - Similar conditions hold for read-only data

State Transition Diagram



Read-Write Locks

- For each variable v, some lock m protects v
  - m is held in write mode for every write of v
  - m is held in some mode (read or write) for every read of v

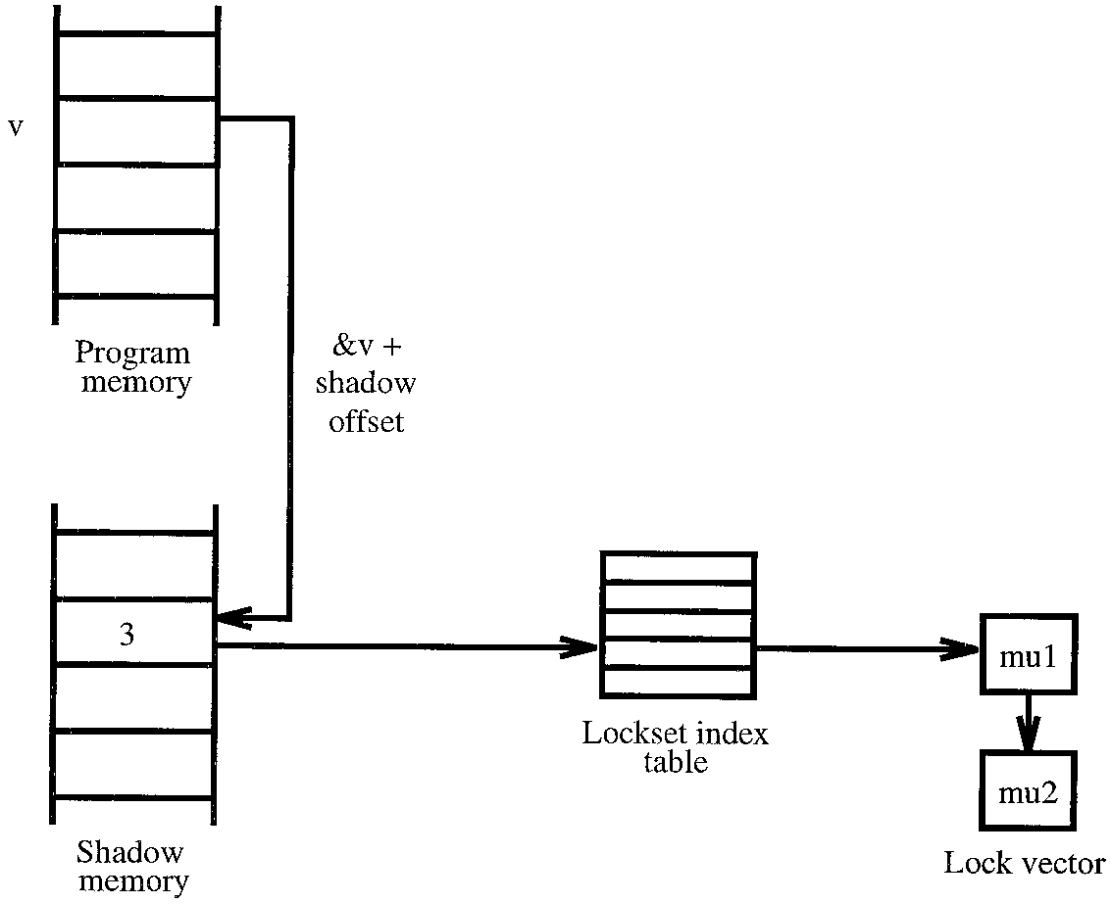
```
Let locks_held(t) be the set of locks held in any mode by thread t.
Let write_locks_held(t) be the set of locks held in write mode by thread t.
For each v, initialize C(v) to the set of all locks.
On each read of v by thread t,
  set C(v) := C(v) ∩ locks_held(t);
  if C(v) :  { }, then issue a warning.
On each write of v by thread t,
  set C(v) := C(v) ∩ write_locks_held(t);
  if C(v) = { }, then issue a warning.
```

  - Locks held in read mode are removed from the candidate set when a write occurs
    - such locks held by a writer do not protect against a data race between the writer and some other reader thread

Implementation

- Binary instrumentation
  - Each load and store
    - Except loads/stores indirect off the stack
  - Each lock/unlock
  - Storage allocator
- Lockset representation
  - Each set of locks represented as index into a table of lock addresses
- Use a shadow memory to hold lockset index

Implementation



Issues

- Lots of imprecision
  - Leads to false alarms for programs that use different kinds of synchronization idioms
    - fork/join
    - barriers
- How do we make race detectors more precise?
  - Hopefully, without losing efficiency

Vector Clocks

- A vector clock records a clock for each thread in the system
  - partially ordered point-wise
  - each thread has its own clock incremented at a lock release operation
  - each thread also keeps a vector clock that records the clock for the last operation of any other thread that happens-before the current operation of this thread
  - each thread also maintains a vector clock for each lock
  - clocks updated on synchronization operations

$$
\begin{array}{rcl}
V_1 \sqsubseteq V_2 & \text{iff} & \forall t.\ V_1(t) \leq V_2(t) \\
V_1 \sqcup V_2 & = & \lambda t.\ max(\,V_1(t),\, V_2(t)) \\
\bot_V & = & \lambda t.\ 0 \\
inc_t(\,V) & = & \lambda u.\ \texttt{if}\ u = t\ \texttt{then}\ V(u) + 1\ \texttt{else}\ V(u)
\end{array}
$$

Vector clocks

- To identify conflicting accesses, keep two vector clocks for Rx and Wx for each variable x.
  - Rx(t) and Wx(t) records the clock of the last read and write to x by thread t.
  - A read from x by thread u is race-free provided it happens-after the last write of each thread, $Wx \sqsubseteq Cu$.
  - A write to x by thread u is race-free provided the write happens after all previous accesses to that variable, $Wx \sqsubseteq Cu$ and $Rx \sqsubseteq Cu$.

Example

$\mathbb{C}_0$	$\mathbb{C}_1$	$\mathbb{L}_m$	$\mathbb{W}_x$
$\langle 4,0,\ldots\rangle$	$\langle 0,8,\ldots\rangle$	$\langle 0,0,\ldots\rangle$	$\langle 0,0,\ldots\rangle$
$\downarrow wr(0,x)$			
$\langle 4,0,\ldots\rangle$	$\langle 0,8,\ldots\rangle$	$\langle 0,0,\ldots\rangle$	$\langle 4,0,\ldots\rangle$
$\downarrow rel(0,m)$			
$\langle 5,0,\ldots\rangle$	$\langle 0,8,\ldots\rangle$	$\langle 4,0,\ldots\rangle$	$\langle 4,0,\ldots\rangle$
	$\downarrow acq(1,m)$		
$\langle 5,0,\ldots\rangle$	$\langle 4,8,\ldots\rangle$	$\langle 4,0,\ldots\rangle$	$\langle 4,0,\ldots\rangle$
	$\downarrow wr(1,x)$		
$\langle 5,0,\ldots\rangle$	$\langle 4,8,\ldots\rangle$	$\langle 4,0,\ldots\rangle$	$\langle 4,8,\ldots\rangle$

Issues

- A target program with n threads
  - requires O(n) storage
  - each vector clock operation (copying, comparing, joining, ...) requires O(n) time
- But, do we really need the full generality of vector clocks?
  - Avoid expensive operations in the general case

FastTrack

- Consider a write-write race:
  - Avoid recording the entire clock
  - In the previous example:
    - critical information is the clock (4) and identity (0) of the thread
  - Denote a pair of clock and thread as an *epoch*
    - Epochs require constant space
    - copying is a constant-time operation
    - comparing an epoch to a vector clock requires constant time

FastTrack

C_0	C_1	L_m	W_x
$\langle 4,0,...\rangle$	$\langle 0,8,...\rangle$	$\langle 0,0,...\rangle$	$\bot_e$
$\downarrow wr(0,x)$			
$\langle 4,0,...\rangle$	$\langle 0,8,...\rangle$	$\langle 0,0,...\rangle$	4@0
$\downarrow rel(0,m)$			
$\langle 5,0,...\rangle$	$\langle 0,8,...\rangle$	$\langle 4,0,...\rangle$	4@0
	$\downarrow acq(1,m)$		
$\langle 5,0,...\rangle$	$\langle 4,8,...\rangle$	$\langle 4,0,...\rangle$	4@0
	$\downarrow wr(1,x)$		
$\langle 5,0,...\rangle$	$\langle 4,8,...\rangle$	$\langle 4,0,...\rangle$	8@1

$$W_x = 4@0 \preceq \langle 4, 8, ...\rangle = C_1$$

FastTrack

- Read-write races
  - more difficult because reads are not totally ordered (even in race-free programs)
  - A write to a variable x can conflict with the last read of x performed by *any* thread, not just the last thread seen in the trace
  - But, in practice
    - reads are totally ordered
      - thread local data
      - lock-protected data
    - only record epochs in this case