Eraser

```
What happens if you run this function in two threads concurrently?
        void f () { ++x; }
  Get a data race--C, C++, many languages define behavior as undefined
    Could increment x by 1 or 2, depending on interleaving
    Could also kill program or worse and still be C11-compliant
  Is it safe if you know compiler will generate single instruction (incl)?
    Not on a multiprocessor
    (x86 allows atomic accesses, but requires expensive lock prefix)

What is a *data race*?
  1. Two evaluations access same non-atomic variable,
  2. At least one is a write, and
  3. Neither evaluation happens before the other

When does one evaluation A *happen before* another B (in C or C++)? (fig. 1)
  1. A & B in same thread, and A sequenced before B, or
  2. A synchronizes with B, E.g.:
     - A releases a lock, B acquires same lock
     - A and B access the same atomic variable normally
     - A is a release store, B is an acquire load on same location
     - A & B relaxed order atomics but surrounded by appropriate fences
  3. There's an X such that A happens before X and X happens before B
  4. Super gnarly stuff involving memory_order_consume (maybe next week)
  5. volatile std::sig_atomic_t accessed in signal handler in same thread

What primitives do we use to avoid data races in concurrent code?
* Mutex (lock):  bracket access to shared variable w. lock (m) & unlock (m)
    System ensures only one lock (m) returns before next unlock (m)
     memory ops after lock (m) appear to happen after ones before lock (m)
     memory ops before unlock (m) happen before ones after it
* Language-level atomics (_Atomic int, std::atomic<int>, ...)
* Semaphore: Two functions P(S) [a.k.a wait] and V(S) [a.k.a. signal]
    Only N more waits will return than signals (for initialization parm N)
    If N == 1, will act like a mutes
* Monitor: exclusive code and associated data
    Shared variables in monitor can only be accessed by its methods
    System ensures only one method of a given monitor executes at a time
    Accesses to variables ordered across method invocations
* Interrupt masking:  bracket code w. x = spl{high,net,...}()  and  splx(x)
    Hierarchical -- e.g., splhigh implies splnet
    For 1-CPU Unix kernel, where threads non-preemptive, but interrupts not
    Or for multiprocessor kernel when interrupt handler might acquire spinlock

Why do we need a special tool to find data races?
  Highly timing dependent
  Often has to do with interaction of two different modules
  Manifestation of bug may be far away from or long after buggy code
    E.g., Linked list corrupted, discover next time you traverse

What is a happens before race detector?
  Check that no data races occurred according to the definition
  What's a brute-force way to check happens before ordering dynamically?
    Instrument code to keep a *vector timestamp* V_t for each thread t
    Increment V_t[t] every time t synchronizes with some other thread t'
    Set V_t[x] := max(V_t[x], V_{t'}[x]) whenever t' synchronizes with t
    V1 happens before V2 when forall x V1[x] <= V2[x]
  What are deficiencies of happens before approach?
    Hard to implement efficiently
    False negatives depending on interleaving of events--See Figure 2
      Want to find all bugs, not just ones you hit in a test run
```

```
What is the key simplifying assumption in this paper?
  Assumes programmers follow a consistent *locking discipline*
    Every shared variable must be protected by some lock
  How can we detect this?  Use ATOM to implement lockset algorithm

What is ATOM?  Allows you to re-write programs (inspired intel Pin)
  Create two .c files:
    eraser.inst.c - allows you to navigate structure of program including
                      adding calls to your own functions
    eraser.anal.c - contains functions you can call when modified program run

What is lockset algorithm?
  For each variable, keep track of set of candidate locks
  If set becomes empty, no lock protecting data, so flag error
  Example:  Figure 3
  Why is this better than happens before?
    Because Eraser detects violations of locking discipline, not races,
    it can detect possible races even if they never occur during testing!

What does eraser instrument with ATOM?
  Every load & store (except stack-relative)
  Every mutex operation (so you know what locks a thread has)
  Thread initialization and finalization code
  Malloc & free operations (so you know when memory reused)

But locking discipline only necessary if >1 thread accesses data
  What memory locations should not be subject to lockset algorithm?
    Stack - heavily used, generally not shared
      Approximate by not instrumenting loads/stores off stack pointer
    Initialization - data only accessed by one thread during initialization
      Example? Initialize thread control block, but on run queue
    Read-shared data - no one writes, so no lock
      Examples? Version number of program, global configuration parameter,
        Constant string entered into hash table, ...
    Read-write locks
  How to avoid generating false positives for these?
    Keep track of state of each variable (Figure 4, p. 398)
    Modify lockset algorithm slightly for read-write locks
      On read of v by t, C(v) gets intersected with locks_held(t)
      On write of v by t, C(v) gets intersected with write_locks_held(t)

p. 398 bottom:  "Our support for initialization makes Eraser's checking
more dependent on the scheduler than we would like."  Why?
  If variable made available before initialization complete, might not detect

What does program report?  How to find the bugs?  (Sec 3, p. 399)
  Reports line of code (backtrace, regs) where lockset becomes emptyset
  Is that enough?
    Say in 100 places var accessed w. correct lock, in 1 it's not
    Likely lockset becomes empty on offending unless it was first access
  But what if unlucky?
    Ask eraser to log all accesses that change a variable's lockset
    One of them will have to be incorrect line of code

Implementation details:
  How much memory?  Slightly more than doubles heap
    For each 32-bit word, keep extra 32-bits of state:
      2 bits for state - Initialization, Exclusive, Shared, Shared-Modified
      30 bits for thread ID (Initialization) or lockset index number
    Also maps for lockset index values, precomputed intersections
    Small expansion of text segment for instrumentation code
  What is lockset index number and why?
    Number of locksets much smaller than maximum possible
    Only keep one, sorted copy of lockset in a table
    Sorting helps for comparison/intersection
```

    Hash list of locks to look up possible index number in hash table
    Cache intersection of different locksets
  Why does this work? Greatest number of locksets seen 10,000
    Could have been exponential in number of locks--why not?
    Lock usage very stylized--same patterns, sets of locks, etc.
      E.g., each instance of object foo might have an internal lock
            but foo objects don't call each other's methods
            so will only lock one foo at a time (# lock sets = # locks)

What other possible causes of false positive are there?
  Memory reuse - private allocators
    Example: Vesta locks head of log, not each element
      Flush routine effectively makes all entries private & empties list
  Private locks - Why would you do this?
    Pthreads does not include shared/exclusive locks
    How to do reader/writer locks?
      Intersect with with only writer_locks_held on writes
  Benign races - Examples?
    Set kill_queries = 1, then when threads notice they exit
    Not really benign since 2011, but same issues arise with atomics
  Multiple locks - any required for read, all required for write
    Design pattern for callbacks, helps avoid deadlocks
    How to fix Eraser?
      Only reduce lockset on writes, just check lockset on reads
      But dangerous because might produce false negatives
  Producer/consumer (post/wait) synchronization - Example? Semaphores
    Pass buffer to disk driver, interrupt handler notifies you when read done
    Might use P/V-style semaphore mechanism to accomplish this
    Why is this hard? Don't know what semaphores are owned by current thread

Partial solution? Allow annotations
  - EraserIgnoreOn (), EraserIgnoreOff ()
  - EraserReuse (address, size)
  - EraserReadLock (lock), EraserReadUnlock (lock)
    EraserWriteLock (lock) EraserWriteUnlock (lock)

How does Eraser work on kernel with splhigh () ... splx (), etc?
  Pretend that each spl level has a lock
  At particular spl or in interrupt, hold locks on current and all lower levels

p. 405 (Sec 4.1): "... it might seem safe to access the p->ip_fp field
in the rest of the procedure... But in fact this would be a mistake."
  Why? p->ip_fp might point to unitialized data
    Alpha architecture does not offer sequential consistency
    Can even reorder dependent instructions!
      Say you access p->ip_fp->some_field where "..." is on p. 404
      Your CPU might load p->ip_fp->some_field before loading p->ip_fp
        Even with a write barrier in NI2_LOCKS_UNLOCK...
          no guarantee on order in which pi->ip_fp vs. some_field pushed out
p. 406 (Sec 4.2): Why is Combine::XorFPTag::FPVal incorrect?
  Again, no sequential consistency; need mb before setting this->validFP = true

What to do about deadlock?
  Add partial order checking to enforced locking discipline
  Seemed to find one deadlock easily this way

How is evaluation?
  Performance? Factor of 10-30 slowdown (Sec 3.2)
  Utility?
    No serious bugs in AltaVista
      but might have found bugs that earlier took several months to fix
    One bug in Vesta
    No serious bugs in Petal
  How about claim that less sensitive to schedule than happens before?
    Found same bugs in AltaVista & Vesta using 2 or 10 threads--good sign

Undergraduate programs? 10% of seemingly working programs had bugs

What should you do about race detection today?
  Compile with -fsanitize=thread
  Hybrid of lockset/happens before algorithms
  Note: Atomics make happens-before tracking much cheaper
    Don't have to instrument all memory accesses, just those on atomics