Lecture Topics

- detecting errors in synchronization
- Eraser + implementation

Administrivia

- …

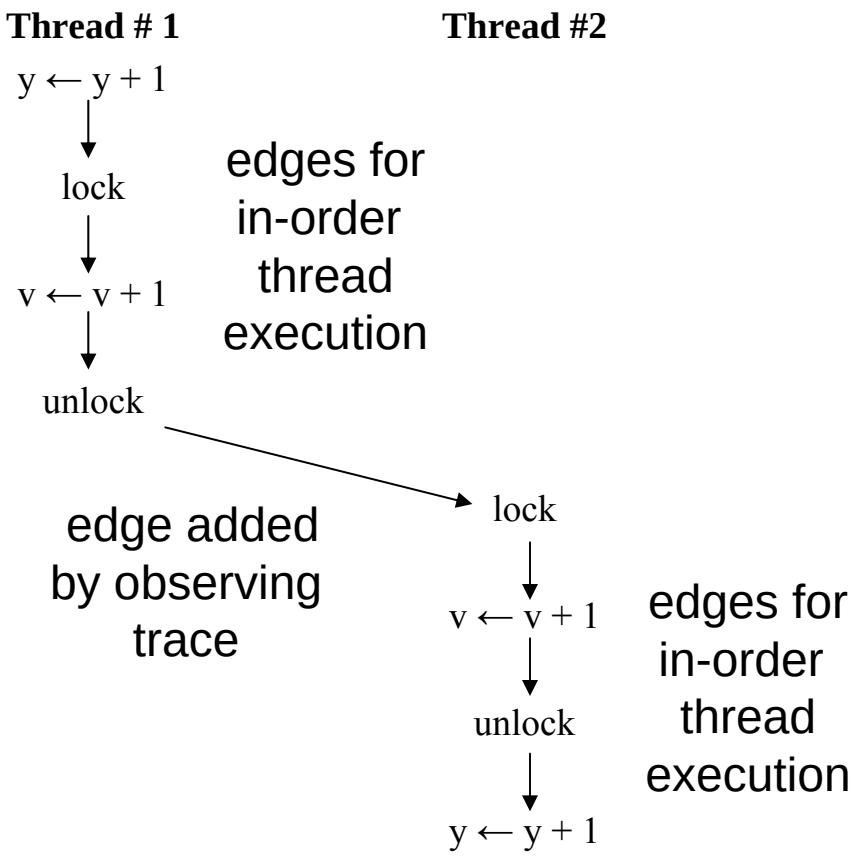
[review]

- the basic problem
  - once variables may be visible to other code/unsynchronized
  - need to worry about the following scenario
    - two blocks of code, B and C
    - each block reads some variables: R_B and R_C are sets
    - each block writess some variables: W_B and W_C are also sets
  - conflict/data race may happen
    - if one block writes a variable read/written by the other
    - that is
      - $W_B \cap (R_C \cup W_C) \neq \varnothing$ OR
      - $W_C \cap (R_B \cup W_B) \neq \varnothing$

- Lamport
  - defined “happens before” relation
  - to mean something that could be causal in an execution
- generalize to “must happen before”: B→C means that
  - some synchronization enforces
  - completion of B before start of C
- if B→C or B→C , no data race is possible
- if neither is true
  - B and C may execute concurrently
  - data race is possible!

- for now
  - we restrict our attention to lock-based critical sections
  - we will return to the more general problem later

- Dinning & Schonberg, 1991
  - introduced the notion of lock covers
  - track which locks held during shared variable accesses
  - not so useful for the general problem of analyzing traces/on-the-fly
  - [example on next page]



- the edge from unlock to lock
  - is added based on observed trace behavior
  - squashes real conflict between updates to y
  - by introducing false ordering

- the core problem: atomicity ≠ order/dependence/concurrent correctness

- analyzing all orderings is not typically feasible
  - N critical sections that could overlap
  - means N! orderings to explore

Eraser

- N.B. became a product of DEC

- starting point: lock covers

- definitions
  - $L_t \equiv$ set of locks held by thread t
  - $C_v \equiv$ candidate locks protecting variable v

- basic algorithm
  - initialization:

    for all v, $C_v \leftarrow$ set of all locks
  - on each access to variable v by thread t

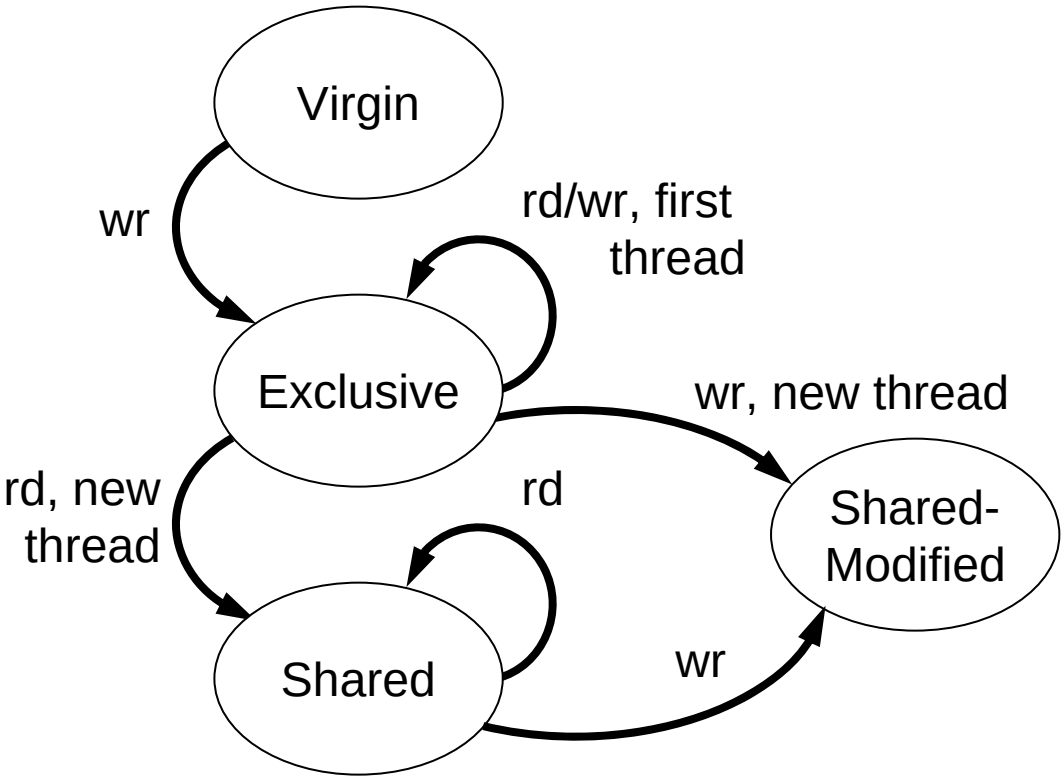
    $C_v \leftarrow C_v \cap L_t$

    if $C_v = \varnothing$, issue warning

- common patterns that violate the basic algorithm
  - initialization
    - create variable, initialize it, then expose it to other threads
    - initialization
      - requires no lock
      - since only one thread can access the new variable
    - after C_v becomes $\varnothing$, lose ability to find errors

  - read sharing
    - write during initialization
    - read-only afterward
    - no need for locks

- addressing initialization
  - when is initialization over?
  - no easy way to be exact
  - conservative: when a second thread accesses variable
- addressing read sharing (again conservatively)
  - assume read only
  - unless the variable is written twice after init is over
- use a simple state machine to track…
  - C_v updated only in Shared/Shared-Modified states
  - warnings issued only from Shared-Modified state



- >1 write by same thread
  - also not acceptable if other threads read without a lock
  - paper text has error (figure is ok, as above)

- another common pattern: reader/writer locks
  - one lock may allow writes sometimes but not at other times
  - need to treat read and write lock variants distinctly

- addressing reader/writer locks
  - change update routine in Shared-Modified state
  - add new variable
    - $W_t \equiv$ set of write locks held by thread t
  - as before
    - $L_t \equiv$ set of locks held by thread t
    - $C_v \equiv$ candidate locks protecting variable v
  - new update routine
    - on each read of variable v by thread t

      $C_v \leftarrow C_v \cap L_t$

      if $C_v = \varnothing$, issue warning
    - on each write of variable v by thread t

      $C_v \leftarrow C_v \cap W_t$

      if $C_v = \varnothing$, issue warning

Eraser Implementation

- based on ATOM binary modification tool (like Pin/MP1)
- start with an observation
  - set of all locks is one abstract set (one thing to represent)
  - even if new locks are created dynamically
- # of lock sets needed in practice is tiny
  - ~10k for commercial software packages
  - use small (30-bit) integer to store index into table of lock vectors

- lock vectors
  - sorted to speed comparison
  - linked from hash table for fast lookup
  - cache of intersections also maintained for speed

- each 8kB data page
  - matched with a shadow page
  - 4B word (unit of coherence) matches 4B shadow word
    - 2-bit state + 30-bit identifier
    - in Exclusive state, id is thread id
    - in other states, id is lock vector #
  - note
    - direct 8kB L1 cache on Alpha
    - requires bit flipping/other offset change to avoid L1 thrashing
    - [noted by Mike Burrows; made perf. usable as tool]

- information in a warning
  - PC, SP, memory address, type of access
  - source line (mapped back from PC)
  - stack backtrace

- performance slowdown: 10-30×
  - not for use in the field (production code)
  - but good for catching most races during debug

- some problems
  - memory reuse
    - many programs use app-specific alloc/reclaim/etc.
    - not understood by Eraser, thus get false positives
  - private lock interfaces
    - Eraser caught only pthreads API
    - Intel now provides SDK for interfacing
      - app-specific synchronization to
      - their race detection tools
  - benign races
    - deliberate and inconsequential
    - e.g., single-producer, single-consumer queue

Catching Errors in the Field

- Can we detect these errror in production code running in the field?

- to answer
  - return to idea of read and write sets for blocks of code
  - all of this work traces back to
    - database work in mid-1970s
      (Eswaran et al., 1974; Stearns et al., 1976)
    - H.T. Kung's optimistic concurrency, 1980

- Tom Knight in 1986 suggested
  - track read and write sets using caches