Threaded Programming

- SMT, Multi-Core Processors
  - are commodity now
  - key to *faster* processors and applications
- complex programming
  - new kind of bugs: data races, dead locks
  - more complex than pointers (as in C)
  - non deterministic (schedule)
- but also allow simpler designs
  - pipelining, boss/workers, producer/consumer

Threads

- parallel/distributed programming
  - threads, processes, clusters
  - requires synchronisation
- single address space
  - same global data and heap data
  - seperate stacks, program counter, registers
  - data synchronization: *shared variables*
  - control synchronization: *mutex, conditions*

Deadlock

- threads T1, T2, synchronization m1, m2
  - T1 waits to synchronize with T2 on m1
  - T2 waits to synchronize with T1 on m2
  - m1 can only be established by T2 after m2
  - m2 can only be established by T1 after m1
- a deadlock *freezes* a system
- may only occur in rare corner cases
  - hard to find and debug

Finding Deadlocks

- models
  - either build or extract abstract model
  - model checking or unit testing
  - goal is exhaustive simulation of all schedules
- search for cyclic dependencies
  - priority inversion (static lock/mutex order)
  - cycles in lock dependency graph
- generate masif *load,* insert *jitter*

Debugging Deadlocks

- access to program state of all threads
  - either through debugging/logging thread
  - or with symbolic debuggers
- *attaching* symbolic debuggers
  - after program seemed to be *frozen*
  - `gdb program.exe` *`pid`*
  - `threads`, `thread 2`, `bt`
- again trade-off between *printf style* debugging and symbolic debugging

Data Races

- unprotected access to shared data
  - protection: *locks/mutex/semaphore/monitor*
  - read/write access by multiple threads
  - value of shared data depends on schedule
- hard to find without *sandboxing*
  - access is just a pointer dereference
- in contrast to cyclic lock dependencies
  - locking can be wrapped in checking code

Proper Lock Protection

THREAD1	THREAD2
lock (mu);	
v = v + 1;	
unlock (mu);	
	lock (mu);
	v = v + 1;
	unlock (mu);

Happens-Before Relation

- dependency between events
  - events in the same thread/process ordered by execution order
  - synchronization among threads/processes
    - sending/receiving message
    - locking/unlocking (of one particular lock)
    - waiting for a condition/enabling a condition
- shared access events should be orderered by happens before relation

Improper Lock Protection 1

m1 != m2

THREAD1	THREAD2
lock (m1);	
v = v + 1;	
unlock (m1);	
	lock (m2);
	v = v + 1;
	unlock (m2);

Improper Lock Protection 2

THREAD1	THREAD2
y = y + 1;	
lock (mu);	
v = v + 1;	
unlock (mu);	
	lock (mu);
	v = v + 1;
	unlock (mu);
	y = y + 1;

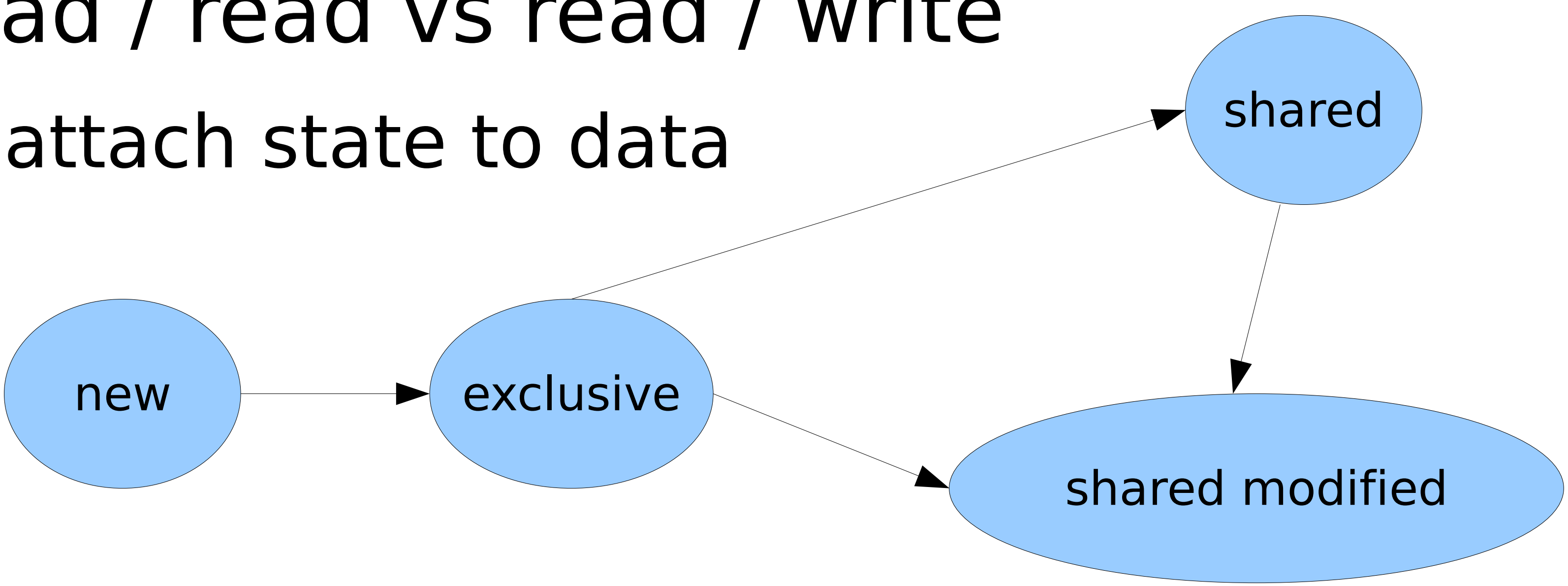
But access events to y still in *happens-before* relation!

Eraser/Lock Set Algorithm

- check for *locking discipline*
  - shared access protected by at least one lock
  - collect lock sets at access events
  - check intersection of lock sets non empty
- if a lock set becomes empty
  - either improper locking
  - even though no problem in *this* run
  - some cases of *false positives / warnings*

Eraser False Warnings

- initialization / collection example
  - data is initialized by boss thread
  - work is spawned off to worker threads
  - results are collected and displayed by boss
- read / read vs read / write
  - attach state to data



High-Level Data Races

- *view* on protected data consistent
  - data X and Y accessed *together* in thread 1
  - access to X alone in thread 2 is fine
  - but it is not *view consistent* to access Y in thread 3 alone
- similar refinements as with Eraser