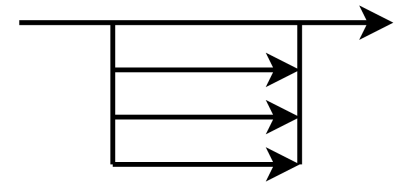
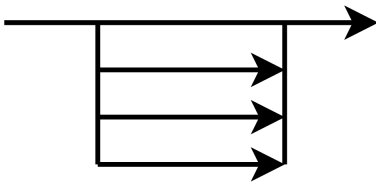


# Debugging Multithreaded Programs

**Debugging**  
**Armin Biere**  
**WS 2012**



# Deadlocks

- 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 massif *load*, insert *jitter*
  - » wait for random time between locks/unlocks
  - » add artificial work

# 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`
- trade-off between *printf style* debugging and symbolic debugging
- use external tools that monitor locking order
  - » for instance *helgrind* (which uses sand boxing)
- programming discipline
  - » add wrappers around locking instead of directly calling `pthread_...`
  - » add checker code that looks incompatible invalid locking order

# Proper Lock Protection

THREAD1

```
lock (mu);
```

```
v = v + 1;
```

```
unlock (mu);
```

THREAD2

```
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 ordered by happens before relation
  - » otherwise data races
  - » non deterministic behavior
  - » usually also incorrect

# Improper Lock Protection 1

THREAD1	$m1 \neq m2$	THREAD2
<code>lock (m1);</code>		
<code>v = v + 1;</code>		
<code>unlock (m1);</code>		
		<code>lock (m2);</code>
		<code>v = v + 1;</code>
		<code>unlock (m2);</code>

# Improper Lock Protection 2

THREAD1

```
y = y + 1;  
lock (mu);  
v = v + 1;  
unlock (mu);
```

THREAD2

```
lock (mu);  
v = v + 1;  
unlock (mu);  
y = y + 1;
```

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

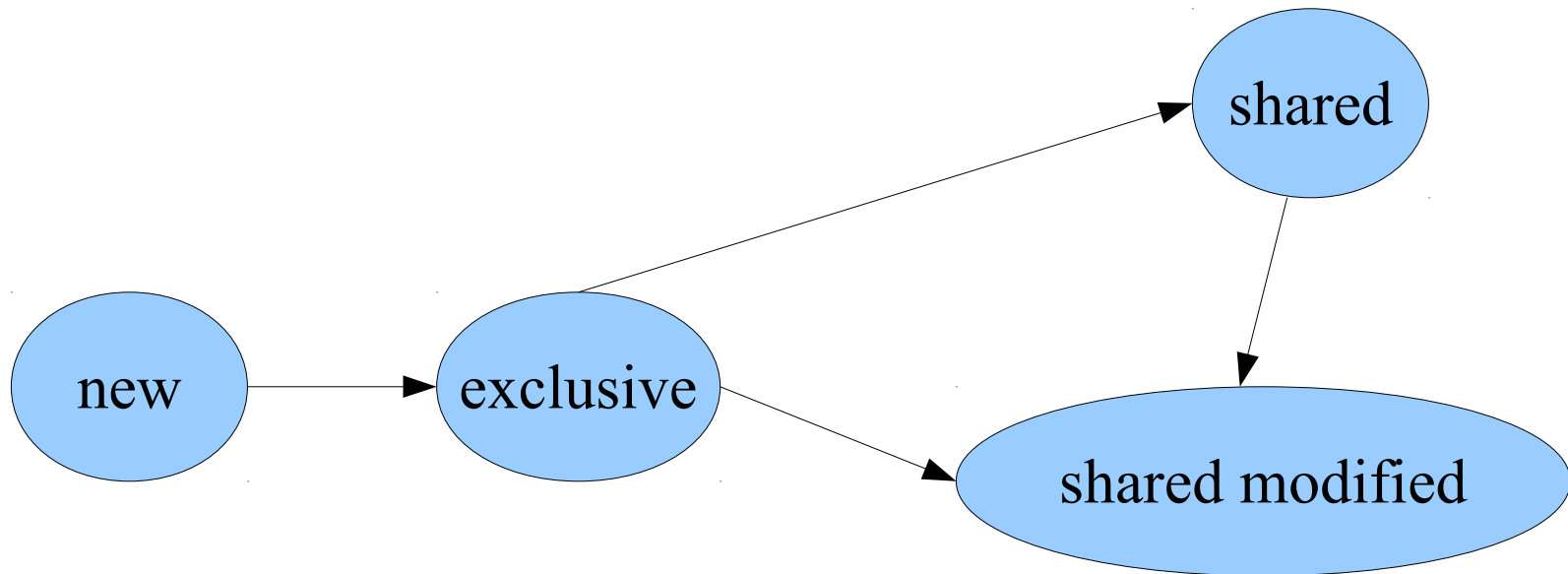


# Eraser/Lock Set Algorithm

- 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*
  - » for instance if threads work in phases
  - » phases are scheduled properly
  - » objects are exclusively own by a certain thread such a phase

# 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
  - » same problems with false positives
  - » needs more programming discipline

# Debugging Data-Races

- tools that implement Eraser algorithm
  - » example again is *helgrind*
  - » usually need sand boxing and thus
  - » much slower than actual code
  - » danger of *Heisenbugs*
- alternatively: programming discipline
  - » wrap access to shared data
  - » add checker for locking discipline
  - » still potential for Heisenbugs
- much more difficult than debugging deadlocks
  - » need to check *all accesses* to data
  - » compared to just checking lock/unlock of a mutex in debugging deadlocks
  - » even worse than debugging pointer related bugs
- schedule steering: massif load and/or random jitter