CS6354: Memory models

To read more...

This day's papers:

Adve and Gharachorloo, "Shared Memory Consistency Models: A Tutorial"
Boehm and Adve, "Foundations of the C++ Concurrency Memory Model", section 1 only

Supplementary readings:

Hennessy and Patterson, section 5.6
Sorin, Hill, and Wood. *A Primer on Memory Consistency and Coherence.*
Boehm, "Threads Cannot Be Implemented as a Library."

double-checked locking

```
class Foo { // BROKEN code
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
  }
  int value;
  // ...
}
```


double-checked locking

```
class Foo { // BROKEN code
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
  }
  int value;
  // ...
}
```

`helper.value` write visible after `helper` write?

compare-and-swap

```
compare–and–swap(address, old, new) {
    with ownership of *address in cache:
        if (*address == old) {
            *address = new;
            return TRUE;
        } else {
            return FALSE;
        }
}
```


CAS lock

Alleged lock with compare-and-swap:

```
class Lock {
    int lockValue = 0;
    void lock() {
        while (!compare–and–swap(&lockValue,
                                 0, 1)) {
            // retry
        }
    }

    void unlock() {
        lockValue = 0;
    }
};
```

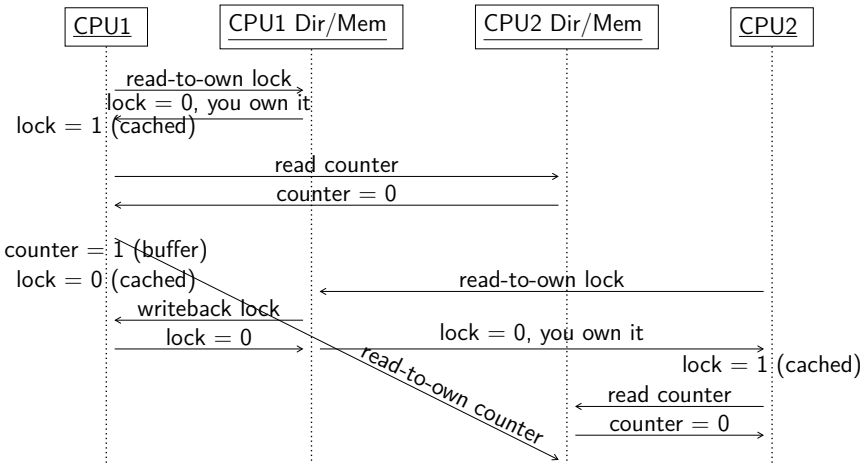

CAS lock: usage

```
Lock counterLock;
int counter = 0;
```

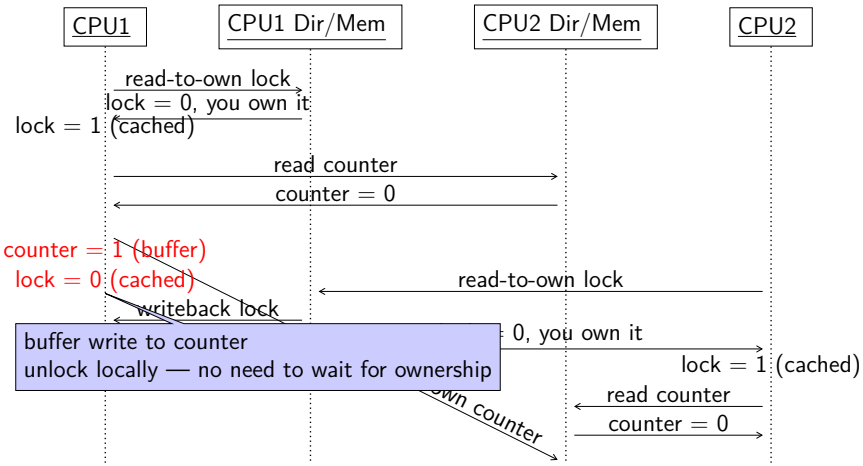
Thread 1	Thread 2
`counterLock.lock();`	`counterLock.lock();`
`counter += 1;`	`counter += 1;`
`counterLock.unlock();`	`counterLock.unlock();`

possible result: `counter == 2`

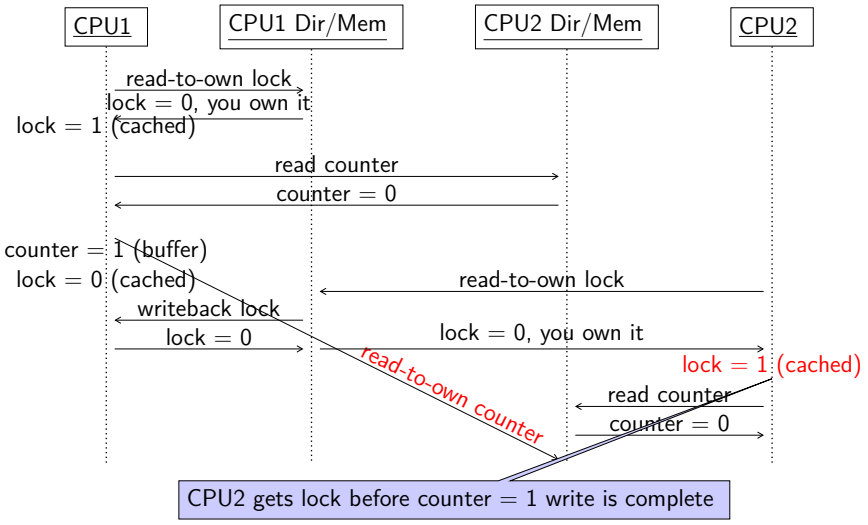
CAS lock: broken timeline



CAS lock: broken timeline



CAS lock: broken timeline



Writing lock before counter?

write buffering — hides their latency

lock release is `lockValue = 0` — nothing special

local write could happen faster than remote

CAS lock: fixed

```
class Lock {
    int lockValue = 0;
    void lock() {
        while (!compare-and-swap(&lockValue,
                                 0, 1)) {
            // retry
        }
        MEMORY_FENCE();
    }

    void unlock() {
        MEMORY_FENCE();
        *lockValue = 0;
    }
};
```

fences

completely complete operations before fence

includes waiting for invalidations

... but doesn't change order of other threads

the acquire/release model

acquire — one-way fence:

operations after acquire aren't done earlier

release — one-way fence:

operations before release aren't done later

memory inconsistency on x86

x = y = 0

thread 1	thread 2
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

possible orders

Thread 1	**Thread 2**
`x = 1`	
	`y = 1`
`r1 = y`	
	`r2 = x`
`r1 == 1`	`r2 == 1`

Thread 1	**Thread 2**
`x = 1`	
`r1 = y`	
	`y = 1`
	`r2 = x`
`r1 == 0`	`r2 == 1`

Thread 1	**Thread 2**
	`y = 1`
	`r2 = x`
`x = 1`	
`r1 = y`	
`r1 == 1`	`r2 == 0`

memory inconsistency on x86

x = y = 0

thread 1	thread 2
`x = 1;`	`y = 1;`
`r1 = y;`	`r2 = x;`

outcomes on my desktop (100M trials)

r1=0	r2=0	3914 (00.003%)
r1=0	r2=1	50196062 (50.196%)
r1=1	r2=0	49798135 (49.798%)
r1=1	r2=1	1889 (00.001%)

X86's omission

stores can be reordered after loads to different addresses

...but thread always sees its own writes immediately

inconsistency causes

in the interprocessor network (not possible with bus)

in the processor

- out-of-order execution of reads and/or writes
- write buffering (don't wait for invalidates)

out-of-order read/write

track dependencies between loads and stores

don't move loads across stores to same address

don't move stores across stores to same address

with one CPU — provides sequential consistency

load bypassing

pending load

0x5678

pending stores

stores before load

address	value
0x1234	not computed
0x2345	0xFFFED
0x4567	not computed
0x9543	0x4123
not computed	not computed

load bypassing

pending load

0x5678

check for conflicts

pending stores

stores before load

address	value
0x1234	not computed
0x2345	0xFFFED
0x4567	not computed
0x9543	0x4123
not computed	not computed

if no conflicts,
run load immediately

load forwarding

pending load

0x5678

pending stores

stores before load

address	value
0x1234	not computed
0x5678	0xFFFED
0x4567	not computed
0x9543	0x4123
not computed	not computed

load forwarding

pending load

0x5678

check for conflicts

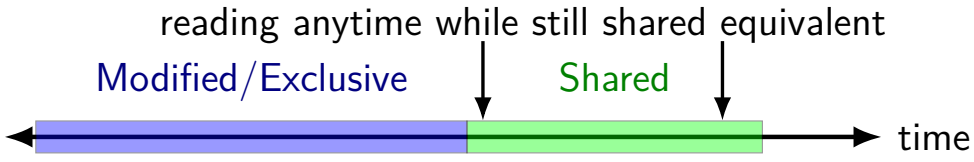
pending stores

stores before load

address	value
0x1234	not computed
0x5678	0xFFFED
0x4567	not computed
0x9543	0x4123
not computed	not computed

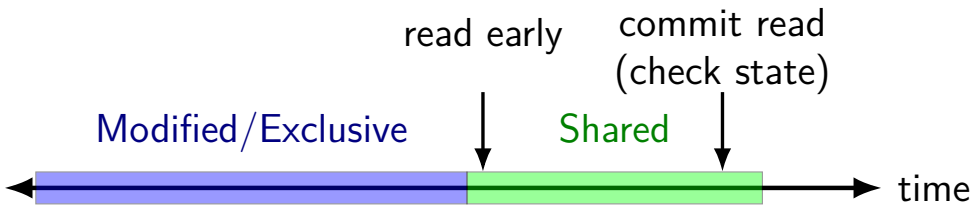
use value from store

sequentially consistent reordering

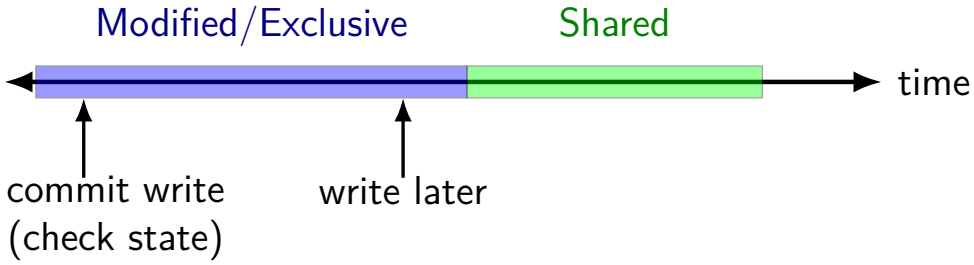


writing anytime while still exclusive equivalent

sequentially consistent reordering



sequentially consistent reordering



conflicts with optimizations

write buffers — need to reserve cache blocks early

load bypassing — needs to check cache state after stores happen

load forwarding — needs to check cache state (even though value from buffer)

interaction with compilers

compilers also reorder loads/stores

e.g. loop optimization for instruction scheduling

is this correct?

depends on memory model compiler presents to user

two definitions

starting point: sequential consistency

System-centric: what reorderings can I observe?

Programmer-centric: what do I do to get sequential consistency?

relaxations

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

read other's write early

```
                Initially X=Y=0
T1              T2              T3              T4
X=1             Y=1             r1=X            r3=Y
                                fence           fence
                                r2=Y            r4=X

r1=1, r2=0, r3=1, r4=0 violates write atomicity
```

T3 reads X, post-update, before T4 receives its update

delay reads until invalidations entirely finished

figures from Boehm, "Foundations of the C++ Concurrency Memory Model"

read other's write early

```
          Initially X=Y=0
T1          T2          T3
X=1       r1=X        r2=Y
          fence       fence
          Y=1         r3=X

r1=1, r2=1, r3=0 violates write atomicity
```

delay reads until invalidations entirely finished

figures from Boehm, "Foundations of the C++ Concurrency Memory Model"

data-race-free

race

- two operations, at least one write
- not separated by synchronization operation

sequentially consistent only if no races

solution to races: add synchronization operation

example: C++ memory model

almost data-race-free

explicit synchronization operations
- library functions

compiler can do aggressive optimization in between

user's perspective: anything can happen if you don't synchronize

prohibited optimization (1)

x = y = 0

thread 1

```
if (x == 1) ++y;
```

optimized to:

```
++y;
if (x != 1) --y;
```

thread 2

```
if (y == 1) ++x;
```

optimized to:

```
++x;
if (y != 1) --x;
```

Example from: Boehm, "Threads Cannot be Implemented as a Library", 2004.

prohibited optimization (2)

```
struct { char a; char b; char c; char d; } x;
...
x.b = 1; x.c = 2; x.d = 3;
```

optimized to:

```
struct { char a; char b; char c; char d; } x;
...
// pseudo-C code:
value = x.a | 0x01020300;
x = value;
```

Example from: Boehm, "Threads Cannot be Implemented as a Library", 2004.

lock-free stack (1)

```
class StackNode { StackNode *next; int value; };
StackNode *head;

void Push(int newValue)  {
    StackNode* newItem = new QueueNode;
    newItem->value = newValue;
    do {
        newItem->next = head;
        MEMORY_FENCE(); // ???
    } while (!compare-and-swap(&head, newItem->next, newItem));
}
```


lock-free stack (2)

```
class StackNode { StackNode *next; int value; };
StackNode *head;

int Pop() {
    StackNode* removed;
    do {
        removed = head;
        MEMORY_FENCE(); // ???
    } while (!compare-and-swap(&head, removed, removed->next));
    /* missing: deallocating removed safely */
    return removed->value;
}
```


wait-freedom

if you stop all other threads, one thread can always make progress

not true with locks — no progress if thread holding lock is stopped

good for latency?

next time: synchronization performance

this lock has a performance problem if contended

cache block changes ownership lots of times

```
class Lock {
    int lockValue = 0;
    void lock() {
        while (!compare-and-swap(&lockValue, 0, 1)) {
            // retry
        }
    }

    void unlock() {
        MEMORY_FENCE();
        *lockValue = 0;
    }
};
```


next time — two papers

Anderson, 1990: how to do better than spinlocks

Guiroux et al, 2016: benchmarks 27 different locks on 35 applications

aside: futex

Linux kernel mechanism to deschedule thread

avoids race condition where lock value changes after unscheduling

explicit call to reschedule thread

Homework 2 notes