



# thinking about race conditions (1)

what are the possible values of  $x$ ?

(initially  $x = y = 0$ )

**Thread A**   **Thread B**

---

$x \leftarrow 1$

$y \leftarrow 2$

## thinking about race conditions (2)

what are some possible values of  $x$ ?

(initially  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

---

$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

## thinking about race conditions (2)

what are some possible values of  $x$ ?

(initially  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

---

$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

## thinking about race conditions (3)

what are the possible values of  $x$ ?

(initially  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

---

$x \leftarrow 1$
------------------

$x \leftarrow 2$
------------------

## thinking about race conditions (2)

what are some possible values of  $x$ ?

(initially  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

---

$x \leftarrow y + 1$
----------------------

$y \leftarrow 2$
------------------

$y \leftarrow y \times 2$
---------------------------

# atomic operation

*atomic operation* = operation that runs to completion or not at all

we will use these to let threads work together

most machines: loading/storing (aligned) words is atomic

so can't get 3 from  $x \leftarrow 1$  and  $x \leftarrow 2$  running in parallel

aligned  $\approx$  address of word is multiple of word size (typically done by compilers)

but some instructions are not atomic; examples:

x86: integer add constant to memory location

many CPUs: loading/storing values that cross cache blocks

e.g. if cache blocks 0x40 bytes, load/store 4 byte from addr. 0x3E is not atomic

# lost adds (program)

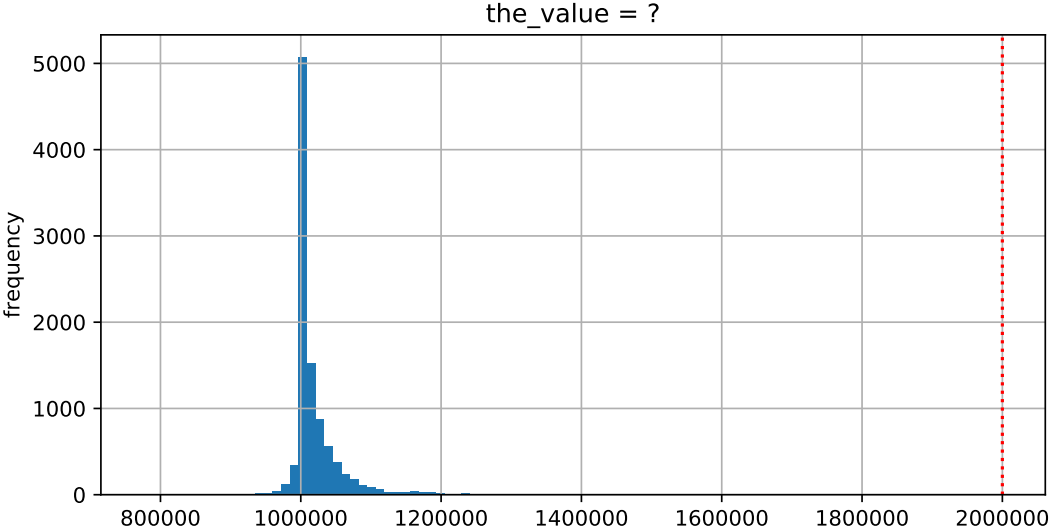
```
.global update_loop
update_loop:
    addl $1, the_value // the_value (global variable) += 1
    dec %rdi           // argument 1 -= 1
    jg update_loop    // if argument 1 >= 0 repeat
    ret
```

---

```
int the_value;
extern void *update_loop(void *);
int main(void) {
    the_value = 0;
    pthread_t A, B;
    pthread_create(&A, NULL, update_loop, (void*) 1000000);
    pthread_create(&B, NULL, update_loop, (void*) 1000000);
    pthread_join(A, NULL);
    pthread_join(B, NULL);
    // expected result: 1000000 + 1000000 = 2000000
    printf("the_value = %d\n", the_value);
}
```



# lost adds (results)



## but how?

probably not possible on single core

exceptions can't occur in the middle of add instruction

...but 'add to memory' implemented with multiple steps

still needs to load, add, store internally

can be interleaved with what other cores do

## but how?

probably not possible on single core

exceptions can't occur in the middle of add instruction

...but 'add to memory' implemented with multiple steps

still needs to load, add, store internally

can be interleaved with what other cores do

(and actually it's more complicated than that — we'll talk later)

# so, what is actually atomic

for now we'll assume: load/stores of 'words'  
(64-bit machine = 64-bits words)

in general: **processor designer will tell you**

their job to design caches, etc. to work as documented

## too much milk

roommates Alice and Bob want to keep fridge stocked with milk:

time	Alice	Bob
3:00	look in fridge. no milk	
3:05	leave for store	
3:10	arrive at store	look in fridge. no milk
3:15	buy milk	leave for store
3:20	return home, put milk in fridge	arrive at store
3:25		buy milk
3:30		return home, put milk in fridge

how can Alice and Bob coordinate better?

# too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

place before buying

remove after buying

don't try buying if there's a note

≈ setting/checking a variable (e.g. “note = 1”)

with atomic load/store of variable

```
if (no milk) {  
  if (no note) {  
    leave note;  
    buy milk;  
    remove note;  
  }  
}
```

# too much milk “solution” 1 (timeline)

**Alice**

```
if (no milk) {  
  if (no note) {  
  
    leave note;  
    buy milk;  
    remove note;  
  }  
}
```

**Bob**

```
if (no milk) {  
  if (no note) {  
  
    leave note;  
    buy milk;  
    remove note;  
  }  
}
```

## too much milk “solution” 2 (algorithm)

intuition: leave note when buying or checking if need to buy

```
leave note;
if (no milk) {
    if (no note) {
        buy milk;
    }
}
remove note;
```



## too much milk: “solution” 2 (timeline)

### Alice

```
leave note;  
if (no milk) {  
    if (no note) {  
        buy milk;  
    }  
}  
remove note;
```

## too much milk: “solution” 2 (timeline)

### Alice

```
leave note;  
if (no milk) {  
    if (no note) { ← but there's always a note  
        buy milk;  
    }  
}  
remove note;
```

## too much milk: “solution” 2 (timeline)

**Alice**

```
leave note;
```

```
if (no milk) {
```

```
  if (no note) {
```

```
    buy milk;
```

```
  }
```

```
}
```

```
remove note;
```

← but there's **always a note**

...will never buy milk (twice or once)

## “solution” 3: algorithm

intuition: label notes so Alice knows which is hers (and vice-versa)

computer equivalent: separate noteFromAlice and noteFromBob variables

### Alice

```
leave note from Alice;
if (no milk) {
    if (no note from Bob) {
        buy milk
    }
}
remove note from Alice;
```

### Bob

```
leave note from Bob;
if (no milk) {
    if (no note from Alice) {
        buy milk
    }
}
remove note from Bob;
```

## too much milk: “solution” 3 (timeline)

**Alice**

```
leave note from Alice
```

```
if (no milk) {
```

```
    if (no note from Bob) {
```

```
        buy milk
```

```
    }
```

```
}
```

```
remove note from Alice
```

**Bob**

```
leave note from Bob
```

```
if (no milk) {
```

```
    if (no note from Alice) {
```

```
        buy milk
```

```
    }
```

```
}
```

```
remove note from Bob
```

## too much milk: is it possible

is there a solutions with writing/reading notes?

≈ loading/storing from shared memory

yes, but it's not very elegant

## too much milk: solution 4 (algorithm)

### Alice

```
leave note from Alice
while (note from Bob) {
    do nothing
}
if (no milk) {
    buy milk
}
remove note from Alice
```

### Bob

```
leave note from Bob
if (no note from Alice) {
    if (no milk) {
        buy milk
    }
}
remove note from Bob
```

## too much milk: solution 4 (algorithm)

### Alice

```
leave note from Alice
```

```
while (note from Bob) {
```

```
    do nothing
```

```
}
```

```
if (no milk) {
```

```
    buy milk
```

```
}
```

```
remove note from Alice
```

### Bob

```
leave note from Bob
```

```
if (no note from Alice) {
```

```
    if (no milk) {
```

```
        buy milk
```

```
    }
```

```
}
```

```
remove note from Bob
```

exercise (hard): prove (in)correctness



## too much milk: solution 4 (algorithm)

### Alice

```
leave note from Alice
```

```
while (note from Bob) {  
    do nothing  
}
```

```
if (no milk) {  
    buy milk  
}
```

```
remove note from Alice
```

### Bob

```
leave note from Bob
```

```
if (no note from Alice) {  
    if (no milk) {  
        buy milk  
    }  
}
```

```
remove note from Bob
```

exercise (hard): prove (in)correctness

## too much milk: solution 4 (algorithm)

### Alice

```
leave note from Alice
```

```
while (note from Bob) {  
    do nothing  
}
```

```
if (no milk) {  
    buy milk  
}
```

```
remove note from Alice
```

### Bob

```
leave note from Bob
```

```
if (no note from Alice) {  
    if (no milk) {  
        buy milk  
    }  
}
```

```
remove note from Bob
```

exercise (hard): prove (in)correctness

exercise (hard): extend to three people

# Peterson's algorithm

general version of solution

see, e.g., Wikipedia

we'll use special hardware support instead

## some definitions

**mutual exclusion:** ensuring only one thread does a particular thing at a time

like checking for and, if needed, buying milk

## some definitions

**mutual exclusion:** ensuring only one thread does a particular thing at a time

like checking for and, if needed, buying milk

**critical section:** code that exactly one thread can execute at a time

result of critical section

## some definitions

**mutual exclusion:** ensuring only one thread does a particular thing at a time

like checking for and, if needed, buying milk

**critical section:** code that exactly one thread can execute at a time

result of critical section

**lock:** object only one thread can hold at a time

interface for creating critical sections

# the lock primitive

locks: an object with (at least) two operations:

*acquire* or *lock* — wait until lock is free, then “grab” it

*release* or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource

forget to acquire lock? weird things happen

```
Lock(MilkLock);  
if (no milk) {  
    buy milk  
}  
Unlock(MilkLock);
```

# pthread mutex

```
#include <pthread.h>

pthread_mutex_t MilkLock;
pthread_mutex_init(&MilkLock, NULL);
...
pthread_mutex_lock(&MilkLock);
if (no milk) {
    buy milk
}
pthread_mutex_unlock(&MilkLock);
```



## xv6 spinlocks

```
#include "spinlock.h"
...
struct spinlock MilkLock;
initlock(&MilkLock, "name for debugging");
...
acquire(&MilkLock);
if (no milk) {
    buy milk
}
release(&MilkLock);
```

## example 2: parallel processing

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU  
one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

## example 2: parallel processing

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU  
one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

## barriers API

`barrier.Initialize(NumberOfThreads)`

`barrier.Wait()` — return after all threads have waited

idea: multiple threads perform computations in parallel

threads wait for **all other threads** to call `Wait()`

# barrier: waiting for finish

```
barrier.Initialize(2);
```

Thread 0

```
partial_mins[0] =  
    /* min of first  
       50M elems */;
```

```
barrier.Wait();
```

```
total_min = min(  
    partial_mins[0],  
    partial_mins[1]  
);
```

Thread 1

```
partial_mins[1] =  
    /* min of last  
       50M elems */;  
barrier.Wait();
```

## barriers: reuse

barriers are reusable:

Thread 0

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```

Thread 1

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```

## barriers: reuse

barriers are reusable:

Thread 0

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```

Thread 1

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```

# barriers: reuse

barriers are reusable:

Thread 0

```
results[0][0] = getInitial(0);  
barrier.Wait();
```

```
results[1][0] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][0] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```

Thread 1

```
results[0][1] = getInitial(1);  
barrier.Wait();
```

```
results[1][1] =  
    computeFrom(  
        results[0][0],  
        results[0][1]  
    );  
barrier.Wait();
```

```
results[2][1] =  
    computeFrom(  
        results[1][0],  
        results[1][1]  
    );
```



# pthread barriers

```
pthread_barrier_t barrier;  
pthread_barrier_init(  
    &barrier,  
    NULL /* attributes */,  
    numberOfThreads  
);  
...  
...  
pthread_barrier_wait(&barrier);
```

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

dynamically allocated array  
reallocated on size changes

can access from multiple threads ...as long as not  
append/erase/etc.?

assuming it's implemented like we expect...

but can we really depend on that?

e.g. could shrink internal array after a while with no expansion save  
memory?

# C++ standard rules for containers

multiple threads can read anything at the same time

can only read element if no other thread is modifying it

can safely add/remove elements if no other threads are accessing container

(sometimes can safely add/remove in extra cases)

exception: vectors of bools — can't safely read and write at same time

might be implemented by putting multiple bools in one int

# implementing locks: single core

intuition: context switch only happens on interrupt  
timer expiration, I/O, etc. causes OS to run

solution: disable them  
reenable on unlock

# implementing locks: single core

intuition: context switch only happens on interrupt  
timer expiration, I/O, etc. causes OS to run

solution: disable them  
reenable on unlock

x86 instructions:

`cli` — disable interrupts

`sti` — enable interrupts

# naive interrupt enable/disable (1)

```
Lock() {  
    disable interrupts  
}
```

```
Unlock() {  
    enable interrupts  
}
```



# naive interrupt enable/disable (1)

```
Lock() {  
    disable interrupts  
}
```

```
Unlock() {  
    enable interrupts  
}
```

problem: user can **hang the system**:

```
Lock(some_lock);  
while (true) {}
```

# naive interrupt enable/disable (1)

```
Lock() {
    disable interrupts
}
Unlock() {
    enable interrupts
}
```

problem: user can **hang the system**:

```
Lock(some_lock);
while (true) {}
```

problem: can't do I/O within lock

```
Lock(some_lock);
read from disk
    /* waits forever for (disabled) interrupt
       from disk IO finishing */
```

## naive interrupt enable/disable (2)

```
Lock() {  
    disable interrupts  
}
```

```
Unlock() {  
    enable interrupts  
}
```

## naive interrupt enable/disable (2)

```
Lock() {  
    disable interrupts  
}
```

```
Unlock() {  
    enable interrupts  
}
```

## naive interrupt enable/disable (2)

```
Lock() {  
    disable interrupts  
}
```

```
Unlock() {  
    enable interrupts  
}
```

## naive interrupt enable/disable (2)

```
Lock() {                               Unlock() {
    disable interrupts                 enable interrupts
}
```

problem: nested locks

```
Lock(milk_lock);
if (no milk) {
    Lock(store_lock);
    buy milk
    Unlock(store_lock);
    /* interrupts enabled here?? */
}
Unlock(milk_lock);
```

# xv6 interrupt disabling (1)

```
...
acquire(struct spinlock *lk) {
    pushcli(); // disable interrupts to avoid deadlock
    ... /* this part basically just for multicore */
}
release(struct spinlock *lk)
{
    ... /* this part basically just for multicore */
    popcli();
}
```

## xv6 push/popcli

pushcli / popcli — need to be in pairs

pushcli — disable interrupts if not already

popcli — enable interrupts if corresponding pushcli disabled them  
don't enable them if they were already disabled



## a simple race

thread\_A:

```
movl $1, x    /* x ← 1 */  
movl y, %eax  /* return y */  
ret
```

thread\_B:

```
movl $1, y    /* y ← 1 */  
movl x, %eax  /* return x */  
ret
```

```
x = y = 0;  
pthread_create(&A, NULL, thread_A, NULL);  
pthread_create(&B, NULL, thread_B, NULL);  
pthread_join(A, &A_result); pthread_join(B, &B_result);  
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

## a simple race

thread\_A:

```
movl $1, x /* x ← 1 */  
movl y, %eax /* return y */  
ret
```

thread\_B:

```
movl $1, y /* y ← 1 */  
movl x, %eax /* return x */  
ret
```

```
x = y = 0;  
pthread_create(&A, NULL, thread_A, NULL);  
pthread_create(&B, NULL, thread_B, NULL);  
pthread_join(A, &A_result); pthread_join(B, &B_result);  
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

if loads/stores atomic, then possible results:

A:1 B:1 — both moves into x and y, then both moves into eax execute

A:0 B:1 — thread A executes before thread B

A:1 B:0 — thread B executes before thread A

# a simple race: results

thread\_A:

```
movl $1, x /* x ← 1 */  
movl y, %eax /* return y */  
ret
```

thread\_B:

```
movl $1, y /* y ← 1 */  
movl x, %eax /* return x */  
ret
```

```
x = y = 0;  
pthread_create(&A, NULL, thread_A, NULL);  
pthread_create(&B, NULL, thread_B, NULL);  
pthread_join(A, &A_result); pthread_join(B, &B_result);  
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	('A executes before B')
171 161	A:1 B:0	('B executes before A')
4 706	A:1 B:1	('execute moves into x+y first')
394	A:0 B:0	???

# a simple race: results

thread\_A:

```
movl $1, x /* x ← 1 */  
movl y, %eax /* return y */  
ret
```

thread\_B:

```
movl $1, y /* y ← 1 */  
movl x, %eax /* return x */  
ret
```

```
x = y = 0;  
pthread_create(&A, NULL, thread_A, NULL);  
pthread_create(&B, NULL, thread_B, NULL);  
pthread_join(A, &A_result); pthread_join(B, &B_result);  
printf("A:%d B:%d\n", (int) A_result, (int) B_result);
```

my desktop, 100M trials:

frequency	result	
99 823 739	A:0 B:1	('A executes before B')
171 161	A:1 B:0	('B executes before A')
4 706	A:1 B:1	('execute moves into x+y first')
394	A:0 B:0	???

# load/store reordering

load/stores atomic, but run *out of order*

recall?: out-of-order processors

processor optimization: execute instructions in non-program order

hide delays from slow caches, variable computation rates, etc.

track side-effects *within a thread* to make as if in-order

but common choice: don't worry as much between cores/threads

design decision: if programmer cares, they worry about it

# why load/store reordering?

prior example: load of x executing before store of y

why do this? otherwise delay the load

if x and y unrelated — no benefit to waiting

## aside: some x86 reordering rules

each core sees its own loads/stores in order

(if a core stores something, it can always load it back)

stores *from other cores* appear in a consistent order

(but a core might observe its own stores too early)

*causality:*

*if* a core reads  $X=a$  and (after reading  $X=a$ ) writes  $Y=b$ ,  
*then* a core that reads  $Y=b$  cannot later read  $X$ =older value than  $a$

# how do you do anything with this?

difficult to reason about what modern CPU's reordering rules do  
typically: don't depend on details, instead:

special instructions with stronger (and simpler) ordering rules  
often same instructions that help with implementing locks in other ways

special instructions that restrict ordering of instructions around  
them (“fences”)  
loads/stores can't cross the fence



# compilers changes loads/stores too (1)

```
void Alice() {  
    note_from_alice = 1;  
    do {} while (note_from_bob);  
    if (no_milk) {++milk;}  
}
```

---

Alice:

```
    movl $1, note_from_alice // note_from_alice ← 1  
    movl note_from_bob, %eax // eax ← note_from_bob
```

```
.L2:  
    testl %eax, %eax  
    jne .L2 // while (eax == 0) repeat  
    cmpl $0, no_milk // if (no_milk != 0) ...  
    ...
```

# compilers changes loads/stores too (1)

```
void Alice() {  
    note_from_alice = 1;  
    do {} while (note_from_bob);  
    if (no_milk) {++milk;}  
}
```

---

Alice:

```
    movl $1, note_from_alice // note_from_alice ← 1  
    movl note_from_bob, %eax // eax ← note_from_bob
```

```
.L2:  
    testl %eax, %eax  
    jne .L2 // while (eax == 0) repeat  
    cmpl $0, no_milk // if (no_milk != 0) ...  
    ...
```

## compilers changes loads/stores too (2)

```
void Alice() {  
    note_from_alice = 1; // "Alice waiting" signal for Bob()  
    do {} while (note_from_bob);  
    if (no_milk) {++milk;}  
    note_from_alice = 2;  
}
```

---

Alice:

```
// compiler optimization: don't set note_from_alice to 1,  
// (why? it will be set to 2 anyway)  
movl note_from_bob, %eax // eax ← note_from_bob  
.L2:  
testl %eax, %eax  
jne .L2 // while (eax == 0) repeat  
...  
movl $2, note_from_alice // note_from_alice ← 2
```

## compilers changes loads/stores too (2)

```
void Alice() {  
    note_from_alice = 1; // "Alice waiting" signal for Bob()  
    do {} while (note_from_bob);  
    if (no_milk) {++milk;}  
    note_from_alice = 2;  
}
```

---

Alice:

```
// compiler optimization: don't set note_from_alice to 1,  
// (why? it will be set to 2 anyway)
```

```
movl note_from_bob, %eax // eax ← note_from_bob
```

.L2:

```
testl %eax, %eax
```

```
jne .L2 // while (eax == 0) repeat
```

```
...
```

```
movl $2, note_from_alice // note_from_alice ← 2
```

## compilers changes loads/stores too (2)

```
void Alice() {  
    note_from_alice = 1; // "Alice waiting" signal for Bob()  
    do {} while (note_from_bob);  
    if (no_milk) {++milk;}  
    note_from_alice = 2;  
}
```

---

Alice:

```
// compiler optimization: don't set note_from_alice to 1,  
// (why? it will be set to 2 anyway)
```

```
movl note_from_bob, %eax // eax ← note_from_bob
```

.L2:

```
testl %eax, %eax
```

```
jne .L2 // while (eax == 0) repeat
```

```
...
```

```
movl $2, note_from_alice // note_from_alice ← 2
```

# pthread and reordering

many pthreads functions **prevent reordering**

everything before function call actually happens before

includes **preventing some optimizations**

e.g. keeping global variable in register for too long

pthread\_mutex\_lock/unlock, pthread\_create, pthread\_join, ...

basically: if pthreads is waiting for/starting something, no weird ordering

# C++: preventing reordering

to help implementing things like `pthread_mutex_lock`

C++ 2011 standard: *atomic* header, *std::atomic* class

prevent CPU reordering *and* prevent compiler reordering

also provide other tools for implementing locks (more later)

could also hand-write assembly code

    compiler can't know what assembly code is doing

# C++: preventing reordering example

```
#include <atomic>
void Alice() {
    note_from_alice = 1;
    do {
        std::atomic_thread_fence(std::memory_order_seq_cst);
    } while (note_from_bob);
    if (no_milk) {++milk;}
}
```

---

```
Alice:
    movl $1, note_from_alice // note_from_alice ← 1
.L2:
    mfence // make sure store visible on/from other cores
    cmpl $0, note_from_bob // if (note_from_bob == 0) repeat fence
    jne .L2
    cmpl $0, no_milk
    ...
```



# C++ atomics: no reordering

```
std::atomic<int> note_from_alice, note_from_bob;  
void Alice() {  
    note_from_alice.store(1);  
    do {  
    } while (note_from_bob.load());  
    if (no_milk) {++milk;}  
}
```

---

```
Alice:  
    movl $1, note_from_alice  
    mfence  
.L2:  
    movl note_from_bob, %eax  
    testl %eax, %eax  
    jne .L2  
    ...
```

# mfence

x86 instruction mfence

make sure all loads/stores in progress finish

...and make sure no loads/stores were started early

fairly expensive

Intel 'Skylake': order 33 cycles + time waiting for pending stores/loads

# **GCC: built-in atomic functions**

used to implement `std::atomic`, etc.

predate `std::atomic`

builtin functions starting with `__sync` and `__atomic`

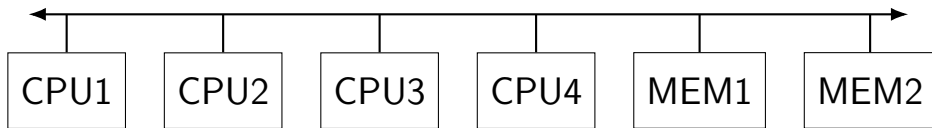
these are what xv6 uses

# connecting CPUs and memory

multiple processors, common memory

how do processors communicate with memory?

# shared bus



tagged messages — everyone gets everything, filters

contention if multiple communicators

some hardware enforces only one at a time

# shared buses and scaling

shared buses perform poorly with “too many” CPUs

so, there are other designs

we'll gloss over these for now

# shared buses and caches

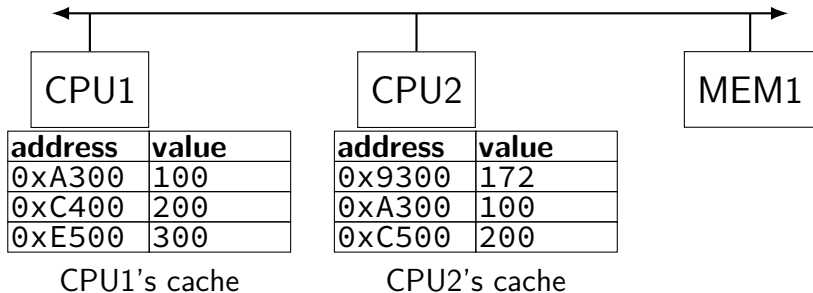
remember caches?

memory is **pretty slow**

each CPU wants to keep local copies of memory

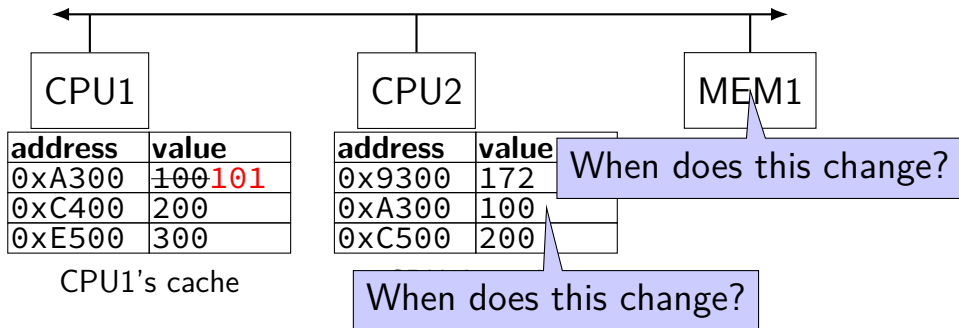
what happens when multiple CPUs cache same memory?

# the cache coherency problem





# the cache coherency problem



CPU1 writes 101 to 0xA300?

# “snooping” the bus

every processor already receives every read/write to memory

take advantage of this to update caches

idea: use messages to clean up “bad” cache entries

# cache coherency states

extra information for **each cache block**

overlaps with/replaces valid, dirty bits

stored in **each cache**

update states based on reads, writes **and heard messages on bus**

different caches may have different states for same block

# MSI state summary

**Modified** value may be **different than memory** *and* I am the only one who has it

**Shared** value is the **same as memory**

**Invalid** I don't have the value; I will need to ask for it

# MSI scheme

from state	hear read	hear write	read	write
Invalid	—	—	to Shared	to Modified
Shared	—	to Invalid	—	to Modified
Modified	to Shared	to Invalid	—	—

blue: transition requires sending message on bus

# MSI scheme

from state	hear read	hear write	read	write
Invalid	—	—	to Shared	to Modified
Shared	—	to Invalid	—	to Modified
Modified	to Shared	to Invalid	—	—

blue: transition requires sending message on bus

example: write while Shared

must send write — inform others with Shared state  
then change to Modified

# MSI scheme

from state	hear read	hear write	read	write
Invalid	—	—	to Shared	to Modified
Shared	—	to Invalid	—	to Modified
Modified	to Shared	to Invalid	—	—

blue: transition requires sending message on bus

example: write while Shared

must send write — inform others with Shared state  
then change to Modified

example: hear write while Shared

change to Invalid  
can send read later to get value from writer

example: write while Modified

nothing to do — no other CPU can have a copy

# MSI example



CPU1

CPU2

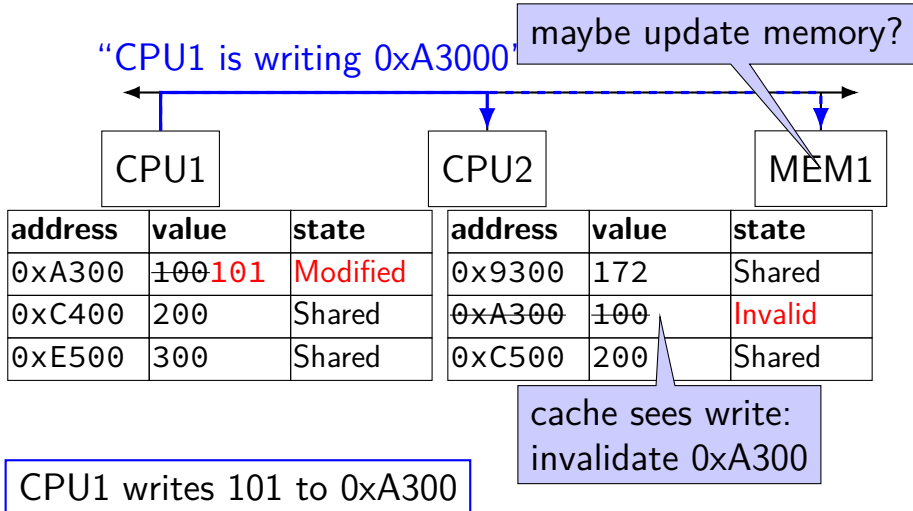
MEM1

address	value	state
0xA300	100	Shared
0xC400	200	Shared
0xE500	300	Shared

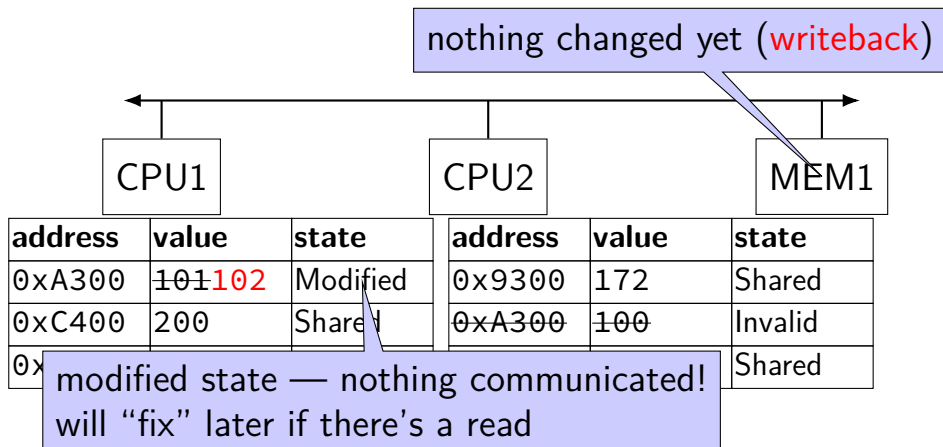
address	value	state
0x9300	172	Shared
0xA300	100	Shared
0xC500	200	Shared



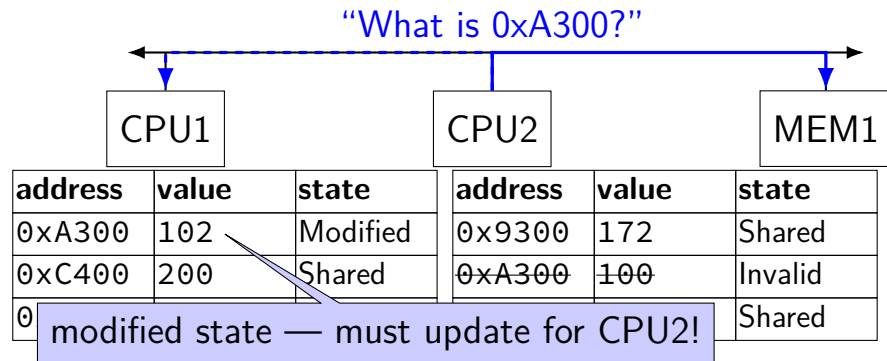
# MSI example



# MSI example



# MSI example



CPU2 reads 0xA300

# MSI example

“Write 102 into 0xA300”



CPU1

CPU2

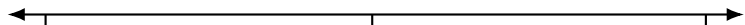
MEM1

address	value	state	address	value	state
0xA300	102	Shared	0x9300	172	Shared
0xC400	200	Shared	0xA300	100	Invalid
0xE					Shared

written back to memory early  
(could also become Invalid at CPU1)

CPU2 reads 0xA300

# MSI example



CPU1

CPU2

MEM1

address	value	state
0xA300	102	Shared
0xC400	200	Shared
0xE500	300	Shared

address	value	state
0x9300	172	Shared
0xA300	<del>100</del> 102	Shared
0xC500	200	Shared

## MSI: update memory

to write value (enter modified state), need to **invalidate** others  
can avoid sending actual value (shorter message/faster)

“I am writing address  $X$ ” versus “I am writing  $Y$  to address  $X$ ”

# MSI: on cache replacement/writeback

still happens — e.g. want to store something else

changes state to **invalid**

requires writeback if modified (= dirty bit)

# cache coherency exercise

modified/shared/invalid; all initially invalid; 32B blocks, 8B read/writes

CPU 1: read 0x1000

CPU 2: read 0x1000

CPU 1: write 0x1000

CPU 1: read 0x2000

CPU 2: read 0x1000

CPU 2: write 0x2008

CPU 3: read 0x1008

Q1: final state of 0x1000 in caches?

Modified/Shared/Invalid for CPU 1/2/3

CPU 1:

CPU 2:

CPU 3:

Q2: final state of 0x2000 in caches?

Modified/Shared/Invalid for CPU 1/2/3

CPU 1:

CPU 2:

CPU 3:



# MSI extensions

real cache coherency protocols sometimes more complex:

separate tracking modifications from whether other caches have copy

send values directly between caches (maybe skip write to memory)

send messages only to cores which might care (no shared bus)

# modifying cache blocks in parallel

cache coherency works on **cache blocks**

but typical memory access — less than cache block

e.g. one 4-byte array element in 64-byte cache block

what if two processors modify different parts same cache block?

4-byte writes to 64-byte cache block

cache coherency — write instructions happen one at a time:

processor 'locks' 64-byte cache block, fetching latest version

processor updates 4 bytes of 64-byte cache block

later, processor might give up cache block

# modifying things in parallel (code)

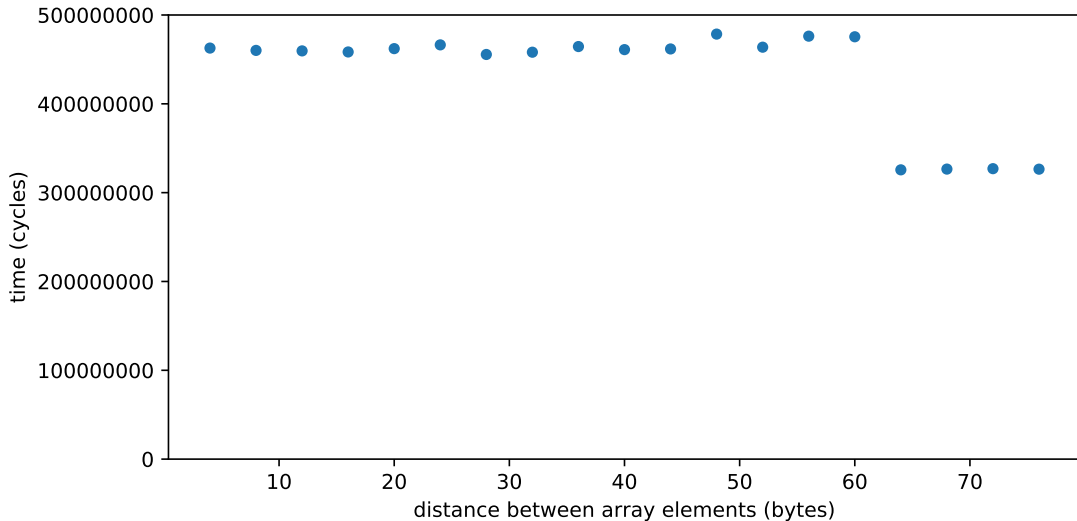
```
void *sum_up(void *raw_dest) {
    int *dest = (int *) raw_dest;
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {
        *dest += data[i];
    }
}
```

```
__attribute__((aligned(4096)))
int array[1024]; /* aligned = address is mult. of 4096 */
```

```
void sum_twice(int distance) {
    pthread_t threads[2];
    pthread_create(&threads[0], NULL, sum_up, &array[0]);
    pthread_create(&threads[1], NULL, sum_up, &array[distance]);
    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
}
```

# performance v. array element gap

(assuming `sum_up` compiled to not omit memory accesses)



# false sharing

synchronizing to access two independent things

two parts of same cache block

solution: separate them

# atomic read-modify-write

really hard to build locks for atomic load store  
and normal load/stores aren't even atomic...

...so processors provide **read/modify/write** operations

one instruction that  
*atomically*

reads *and* modifies *and* writes back a value

## x86 atomic exchange

```
lock xchg (%ecx), %eax
```

atomic exchange

$$\text{temp} \leftarrow M[\text{ECX}]$$
$$M[\text{ECX}] \leftarrow \text{EAX}$$
$$\text{EAX} \leftarrow \text{temp}$$

...without being interrupted by other processors, etc.

# test-and-set: using atomic exchange

one instruction that...

writes a fixed new value

and reads the old value



# test-and-set: using atomic exchange

one instruction that...

writes a fixed new value

and reads the old value

write: mark a locked as TAKEN (no matter what)

read: see if it was already TAKEN (if so, only us)

# implementing atomic exchange

get cache block into *Modified* state

do read+modify+write operation while state doesn't change

recall: Modified state = "I am the only one with a copy"

## x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

if 1: someone has the lock; if 0: lock is free to take

acquire:

```
    movl $1, %eax           // %eax ← 1
    lock xchg %eax, the_lock // swap %eax and the_lock
                             // sets the_lock to 1 (taken)
                             // sets %eax to prior val. of the_lock
    test %eax, %eax         // if the_lock wasn't 0 before:
    jne acquire             //   try again
    ret
```

release:

```
    mfence                 // for memory order reasons
    movl $0, the_lock      // then, set the_lock to 0 (not taken)
    ret
```

## x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

if 1: someone has the lock; if 0: lock is free to take

acquire:

```
movl $1, %eax           // %eax ← 1
lock xchg %eax, the_lock // swap %eax and the_lock
                        // sets the_lock to 1 (taken)
                        // sets %eax to prior val. of the_lock

test %eax, %eax         // if the_lock == 1
jne acquire            // then, set lock variable to 1 (taken)
ret                    // read old value
```

release:

```
mfence                // for memory order reasons
movl $0, the_lock    // then, set the_lock to 0 (not taken)
ret
```

# x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

if 1: someone has the lock; if 0: lock is free to take

acquire:

```
movl $1, %eax           // %eax ← 1
lock xchg %eax, the_lock // swap %eax and the_lock
                        // sets the_lock to 1 (taken)
                        // sets %eax to prior val of the_lock
```

```
test %eax, %eax
jne acquire
ret
```

if lock was already locked retry  
“spin” until lock is released elsewhere

release:

```
mfence                 // for memory order reasons
movl $0, the_lock     // then, set the_lock to 0 (not taken)
ret
```

# x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

if 1: someone has the lock; if 0: lock is free to take

acquire:

```
    movl $1, %eax           // %eax ← 1
    lock xchg %eax, the_lock // swap %eax and the_lock
                               // sets the_lock to 1 (taken)
                               // sets %eax to prior val of the_lock

    test %eax, %eax
    jne acquire
    ret
```

release lock by setting it to 0 (not taken)  
allows looping acquire to finish

release:

```
    mfence                 // for memory order reasons
    movl $0, the_lock      // then, set the_lock to 0 (not taken)
    ret
```

# x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

if 1: someone has the lock; if 0: lock is free to take

acquire:

```
movl $1, %eax           // %eax ← 1
lock xchg %eax, the_lock // swap %eax and the_lock
                        // sets the_lock to 1 (taken)
```

```
test %eax, %eax
jne acquire
ret
```

Intel's manual says:  
no reordering of loads/stores across a `lock`  
or `mfence` instruction

release:

```
mfence                 // for memory order reasons
movl $0, the_lock     // then, set the_lock to 0 (not taken)
ret
```

# backup slides



# GCC: preventing reordering example (1)

```
void Alice() {  
    int one = 1;  
    __atomic_store(&note_from_alice, &one, __ATOMIC_SEQ_CST);  
    do {  
    } while (__atomic_load_n(&note_from_bob, __ATOMIC_SEQ_CST));  
    if (no_milk) {++milk;}  
}
```

---

```
Alice:  
    movl $1, note_from_alice  
    mfence  
.L2:  
    movl note_from_bob, %eax  
    testl %eax, %eax  
    jne .L2  
    ...
```

## GCC: preventing reordering example (2)

```
void Alice() {
    note_from_alice = 1;
    do {
        __atomic_thread_fence(__ATOMIC_SEQ_CST);
    } while (note_from_bob);
    if (no_milk) {++milk;}
}
```

---

Alice:

```
    movl $1, note_from_alice // note_from_alice ← 1
```

.L3:

```
    mfence // make sure store is visible to other cores before  
           // on x86: not needed on second+ iteration of loop
```

```
    cmpl $0, note_from_bob // if (note_from_bob == 0) repeat fe
```

```
    jne .L3
```

```
    cmpl $0, no_milk
```

```
    ...
```

# xv6 spinlock: debugging stuff

```
void acquire(struct spinlock *lk) {
    ...
    if(holding(lk))
        panic("acquire")
    ...
    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
void release(struct spinlock *lk) {
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;
    ...
}
```

# xv6 spinlock: debugging stuff

```
void acquire(struct spinlock *lk) {  
    ...  
    if(holding(lk))  
        panic("acquire")  
    ...  
    // Record info about lock acquisition for debugging.  
    lk->cpu = mycpu();  
    getcallerpcs(&lk, lk->pcs);  
}  
void release(struct spinlock *lk) {  
    if(!holding(lk))  
        panic("release");  
  
    lk->pcs[0] = 0;  
    lk->cpu = 0;  
    ...  
}
```

# xv6 spinlock: debugging stuff

```
void acquire(struct spinlock *lk) {
    ...
    if(holding(lk))
        panic("acquire")
    ...
    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}
void release(struct spinlock *lk) {
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;
    ...
}
```

# xv6 spinlock: debugging stuff

```
void acquire(struct spinlock *lk) {
    ...
    if(holding(lk))
        panic("acquire")
    ...
    // Record info about lock acquisition for debugging.
    lk->cpu = mycpu();
    getcallerpcs(&lk, lk->pcs);
}

void release(struct spinlock *lk) {
    if(!holding(lk))
        panic("release");

    lk->pcs[0] = 0;
    lk->cpu = 0;
    ...
}
```

# exercise: fetch-and-add with compare-and-swap

exercise: implement fetch-and-add with compare-and-swap

```
compare_and_swap(address, old_value, new_value) {  
    if (memory[address] == old_value) {  
        memory[address] = new_value;  
        return true;    // x86: set ZF flag  
    } else {  
        return false;  // x86: clear ZF flag  
    }  
}
```

**solution**



## xv6 spinlock: acquire

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    ...
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();
    ...
}
```

# xv6 spinlock: acquire

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    ...
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __asm__ volatile "fence" : : : ;
    // don't let us be interrupted after while have the lock
    // problem: interruption might try to do something with the lock
    // ...but that can never succeed until we release the lock
    // ...but we won't release the lock until interruption finishes
}
```

# xv6 spinlock: acquire

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    ...
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();
    ...
}
```

xchg wraps the lock xchg instruction  
same loop as before

# xv6 spinlock: acquire

```
void
acquire(struct spinlock *lk)
{
    pushcli(); // disable interrupts to avoid deadlock.
    ...
    // The xchg is atomic.
    while(xchg(&lk->locked, 1) != 0)
        ;

    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that the critical section's memory
    // references happen after the lock is acquired.
    __sync_synchronize();
    ... avoid load store reordering (including by compiler)
    on x86, xchg alone is enough to avoid processor's reordering
    (but compiler may need more hints)
}
```

## xv6 spinlock: release

```
void
release(struct spinlock *lk)
{
    ...
    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

# xv6 spinlock: release

```
void
release(struct spinlock *lk)
{
    ...
    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli( turns into instruction to tell processor not to reorder
            plus tells compiler not to reorder
    }
}
```

# xv6 spinlock: release

```
void
release(struct spinlock *lk)
{
    ...
    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

turns into mov of constant 0 into lk->locked

# xv6 spinlock: release

```
void
release(struct spinlock *lk)
{
    ...
    // Tell the C compiler and the processor to not move loads or stores
    // past this point, to ensure that all the stores in the critical
    // section are visible to other cores before the lock is released.
    // Both the C compiler and the hardware may re-order loads and
    // stores; __sync_synchronize() tells them both not to.
    __sync_synchronize();

    // Release the lock, equivalent to lk->locked = 0.
    // This code can't use a C assignment, since it might
    // not be atomic. A real OS would use C atomics here.
    asm volatile("movl $0, %0" : "+m" (lk->locked) : );

    popcli();
}
```

reenable interrupts (taking nested locks into account)



# fetch-and-add with CAS (1)

```
compare-and-swap(address, old_value, new_value) {  
    if (memory[address] == old_value) {  
        memory[address] = new_value;  
        return true;  
    } else {  
        return false;  
    }  
}
```

---

```
long my_fetch_and_add(long *pointer, long amount) { ... }
```

implementation sketch:

- fetch value from pointer `old`
- compute in temporary value result of addition `new`
- try to change value at pointer from `old` to `new`  
[compare-and-swap]
- if not successful, repeat

## fetch-and-add with CAS (2)

```
long my_fetch_and_add(long *p, long amount) {  
    long old_value;  
    do {  
        old_value = *p;  
    } while (!compare_and_swap(p, old_value, old_value + amount));  
    return old_value;  
}
```

## exercise: append to singly-linked list

ListNode is a singly-linked list

assume: threads *only* append to list (no deletions, reordering)

use compare-and-swap(pointer, old, new):

- atomically change \*pointer from old to new

- return true if successful

- return false (and change nothing) if \*pointer is not old

```
void append_to_list(ListNode *head, ListNode *new_last_node) {  
    ...  
}
```

# some common atomic operations (1)

*// x86: emulate with exchange*

```
test_and_set(address) {  
    old_value = memory[address];  
    memory[address] = 1;  
    return old_value != 0; // e.g. set ZF flag  
}
```

*// x86: xchg REGISTER, (ADDRESS)*

```
exchange(register, address) {  
    temp = memory[address];  
    memory[address] = register;  
    register = temp;  
}
```

## some common atomic operations (2)

```
// x86: mov OLD_VALUE, %eax; lock cmpxchg NEW_VALUE, (ADDRESS)
compare-and-swap(address, old_value, new_value) {
    if (memory[address] == old_value) {
        memory[address] = new_value;
        return true;    // x86: set ZF flag
    } else {
        return false;  // x86: clear ZF flag
    }
}
```

```
// x86: lock xaddl REGISTER, (ADDRESS)
fetch-and-add(address, register) {
    old_value = memory[address];
    memory[address] += register;
    register = old_value;
}
```

# common atomic operation pattern

try to do operation, ...

detect if it failed

if so, repeat

atomic operation does “try and see if it failed” part