synchronization 2

changelog

changes since lecture version:

- 2 March 2022: life homework even/odd (not shown in lecture) fix precedence issue in pseudocode

last time

atomic operation concept

- all of it happens or none of it happens
- can't observe in-between state

atomic load/stores not really enough

lock abstraction:

- lock/acquire — wait for lock to be available
- unlock/release — allow another to use lock
- pattern: lock before using shared resource, unlock after

pthread_mutex, xv6 spinlock

exercise

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

possible values of one/two after A+B run?

exercise (alternate 1)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
}
```

possible values of one/two after A+B run?

exercise (alternate 2)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;
string one = "init one", two = "init two";
void ThreadA() {
    pthread_mutex_lock(&lock2);
    two = "two in ThreadA";  // (A2)
    pthread_mutex_unlock(&lock2);
    pthread_mutex_lock(&lock1);
    one = "one in ThreadA";  // (A1)
    pthread_mutex_unlock(&lock1);
}
void ThreadB() {
    pthread_mutex_lock(&lock1);
    one = "one in ThreadB";  // (B1)
    pthread_mutex_unlock(&lock1);
    pthread_mutex_lock(&lock2);
    two = "two in ThreadB";  // (B2)
    pthread_mutex_unlock(&lock2);
}
```

possible values of one/two after A+B run?

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

are locks enough?

do we need more than locks?

example 1: pipes?

suppose we want to implement a pipe with threads

`read` sometimes needs to wait for a `write`

don't want busy-wait

(and trick of having writer unlock() so reader can finish a lock() is illegal)

more synchronization primitives

need other ways to wait for threads to finish

we'll introduce several synchronization ideas beyond locks:

- barriers — (today)
- condition variables / monitors
- counting semaphores
- reader/writer locks

barriers

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

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);
```

life homework (pseudocode)

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    for (int y = 0; y < size; ++y) {
        for (int x = 0; x < size; ++x) {
            to_grid(x, y) = computeValue(from_grid, x, y);
        }
    }
    swap(from_grid, to_grid);
}
```

life homework

compute grid of values for time t from grid for time $t-1$

- compute new value at i, j based on surrounding values

parallel version: produce parts of grid in different threads

use barriers to finish time t before going to time $t+1$

- avoid trying to read things that aren't computed

CoA2 (pilot new curriculum) students: additional requirement

- also additional on next pool assignment — start early!

life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    ... compute to_grid ...
    swap(from_grid, to_grid);
}
```

but this alternative needs less locking:

```
Grid grids[2];
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    from_grid = &grids[time % 2];
    to_grid = &grids[(time % 2) + 1];
    ... compute to_grid ...
}
```

life homework even/odd

naive way has an operation that needs locking:

```
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    ... compute to_grid ...
    swap(from_grid, to_grid);
}
```

but this alternative needs less locking:

```
Grid grids[2];
for (int time = 0; time < MAX_ITERATIONS; ++time) {
    from_grid = &grids[time % 2];
    to_grid = &grids[(time % 2) + 1];
    ... compute to_grid ...
}
```

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() {
    disable interrupts
}
```

```
Unlock() {
    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

compilers move loads/stores (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 move loads/stores (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 move 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 move 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 move 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
```

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	???

pthreads 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

implementation part 1: prevent compiler reordering

implementation part 2: use special instructions

- example: x86 `mfence` instruction

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

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

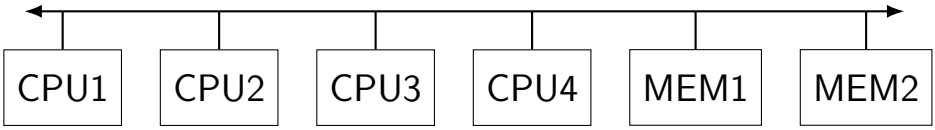
aside: this instruction is did not exist in the original x86
so xv6 uses something older that's equivalent

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

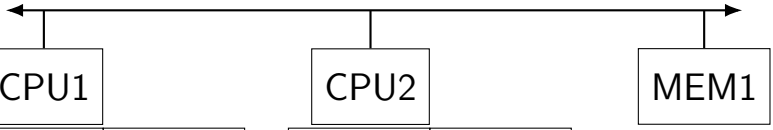
remember caches?

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



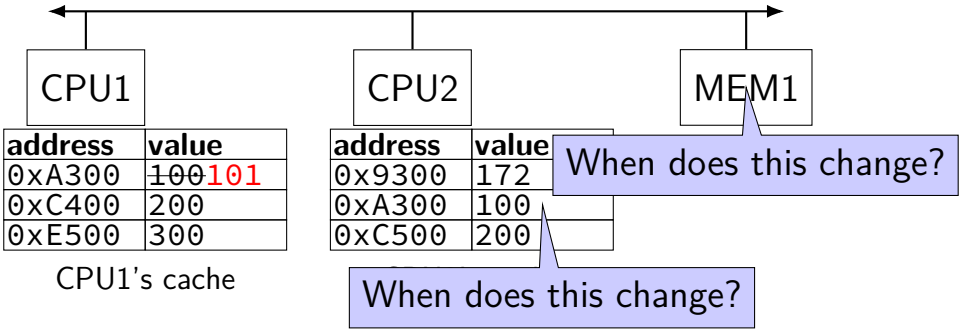
address	value
0xA300	100
0xC400	200
0xE500	300

CPU1's cache

address	value
0x9300	172
0xA300	100
0xC500	200

CPU2's cache

the cache coherency problem



CPU1 writes 101 to 0xA300?

using a shared the bus

want to change a value other processors might have?

use bus to tell them “get rid of your copy”

want to start using value other processor might have reserved?

use bus to say “I’d like to use this value now”

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)

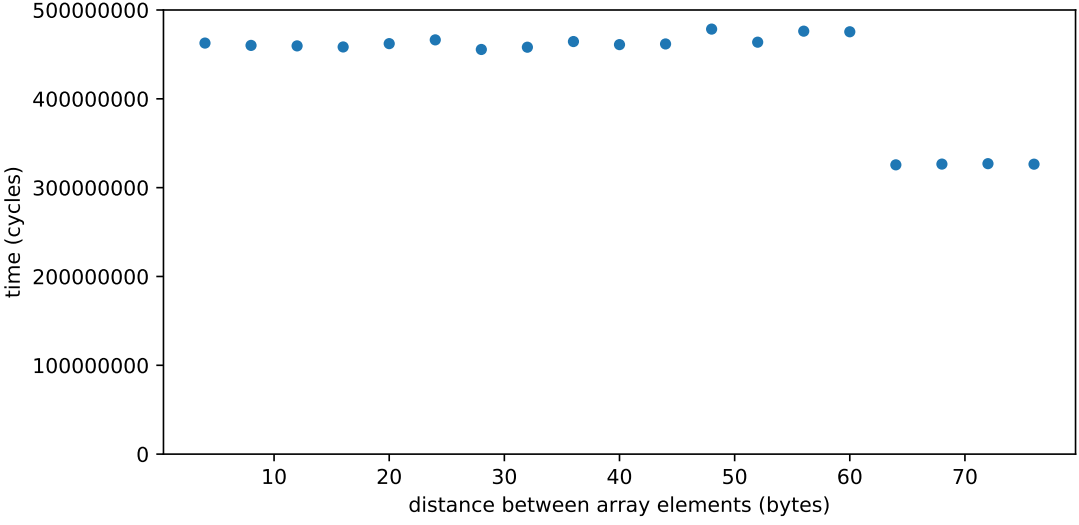
```
void *sum_up(void *raw_dest) {
    int *dest = (int *) raw_dest;
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {
        *dest += data[i];
    }
}

__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

exercise (1)

```
int values[1024];
int results[2];
void *sum_front(void *ignored_argument) {
    results[0] = 0;
    for (int i = 0; i < 512; ++i)
        results[0] += values[i];
    return NULL;
}
void *sum_back(void *ignored_argument) {
    results[1] = 0;
    for (int i = 512; i < 1024; ++i)
        results[1] += values[i];
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL);
    pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

Where is false sharing likely to occur? How to fix?

exercise (2)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        my_info->result += my_info->values[i];
    }
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

Where is false sharing likely to occur?

atomic read-modfiy-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

temp ← M[ECX]

M[ECX] ← EAX

EAX ← temp

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

implementing atomic exchange

make sure other processors don't have cache block

do read+modify+write operation

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 th
    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 th

    test %eax, %eax               // if
    jne acquire                   //   t
    ret

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

set lock variable to 1 (taken)
read old value

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 th
    test %eax, %eax
    jne acquire
    ret

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

if lock was already locked retry
"spin" until lock is released elsewhere

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 th

    test %eax, %eax
    jne acquire
    ret

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

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

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)
                                                              of th
    test %eax, %eax
    jne acquire
    ret

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

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

exercise: spin wait

consider implementing 'waiting' functionality of pthread_join

thread calls ThreadFinish() when done

complete code below:

```
finished: .quad 0
ThreadFinish:
    ________________________
    ret
ThreadWaitForFinish:
    ________________________
    lock xchg %eax, finished
    cmp $0, %eax
    ____ ThreadWaitForFinish
    ret
```

A. `mfence; mov $1, finished`
B. `mov $1, finished; mfence`
C. `mov $0, %eax`
D. `mov $1, %eax`
E. `je`
F. `jne`

exercise: spin wait

```
finished: .quad 0
ThreadFinish:
    __________A_____________
    ret
ThreadWaitForFinish:
    _________B_____________
    lock xchg %eax, finished
    cmp $0, %eax
    __C_ ThreadWaitForFinish
    ret
```

```
/* or without using a writing instru
mov %eax, finished
mfence
cmp $0, %eax
je ThreadWaitForFinish
ret
```

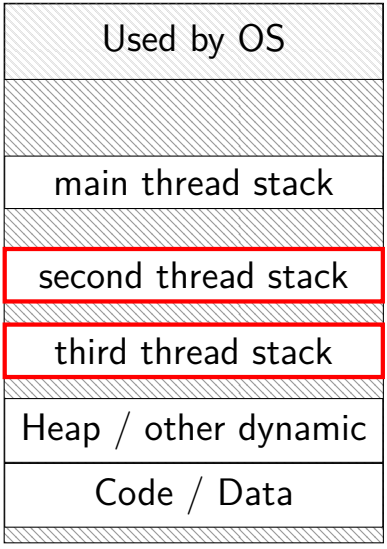
A. `mfence; mov $1, finished`
B. `mov $1, finished; mfence`
C. `mov $0, %eax`
D. `mov $1, %eax`
E. `je`
F. `jne`

backup slides

what's wrong with this?

```
/* omitted: headers */
#include <string>
using std::string;
void *create_string(void *ignored_argument) {
  string result;
  result = ComputeString();
  return &result;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  string *string_ptr;
  pthread_join(the_thread, (void*) &string_ptr);
  cout << "string is " << *string_ptr;
}
```

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

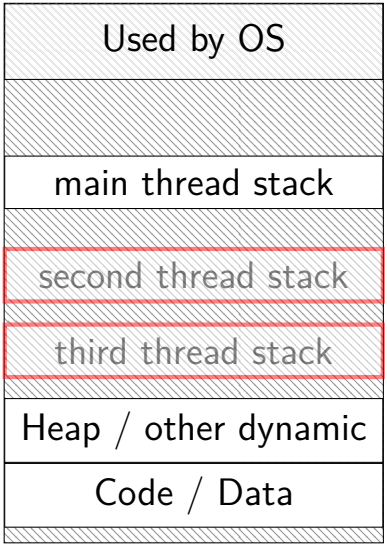
`string result` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

program memory



0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

`string result` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

load/store reordering

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

recall?: out-of-order processors

processor optimization: sometimes execute instructions in non-program order

- hide delays from slow caches, variable computation rates, etc.
- documneted limits on when this is/is not allowed

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

want to avoid this *special instructions ensure strict ordering*

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

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

```
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;
}
```

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 stor
  // 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 stor
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __s
  ..
}
```

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 stor
  // 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 stor
  // 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 stor
  // 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 stor
  // 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 stor
  // 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 stor
  // 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)

mutex efficiency

'normal' mutex **uncontended** case:

lock: acquire + release spinlock, see lock is free

unlock: acquire + release spinlock, see queue is empty

not much slower than spinlock

pthread mutexes: addt'l features

mutex attributes (`pthread_mutexattr_t`) allow:

(reference: `man pthread.h`)

error-checking mutexes

locking mutex twice in same thread?

unlocking already unlocked mutex?

...

mutexes shared between processes

otherwise: must be only threads of same process

(unanswered question: where to store mutex?)

...

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) {
    ...
}
```

append to singly-linked list

```
/* assumption: other threads may be appending to list,
 *             but nodes are not being removed, reordered, etc.
 */
void append_to_list(ListNode *head, ListNode *new_last_node) {
  memory_ordering_fence();
  ListNode *current_last_node;
  do {
    current_last_node = head;
    while (current_last_node->next) {
      current_last_node = current_last_node->next;
    }
  } while (
    !compare-and-swap(&current_last_node->next,
                      NULL, 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

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

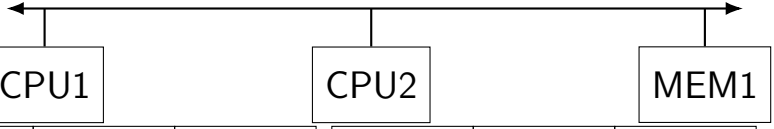
example: hear write while Shared

- 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

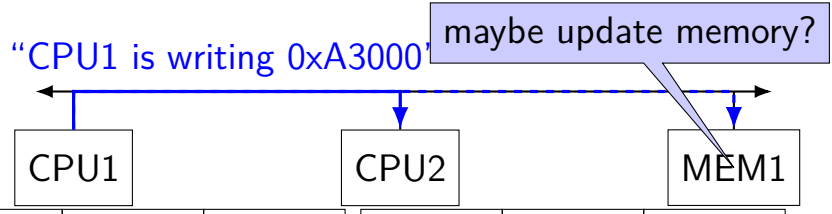
MSI example



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



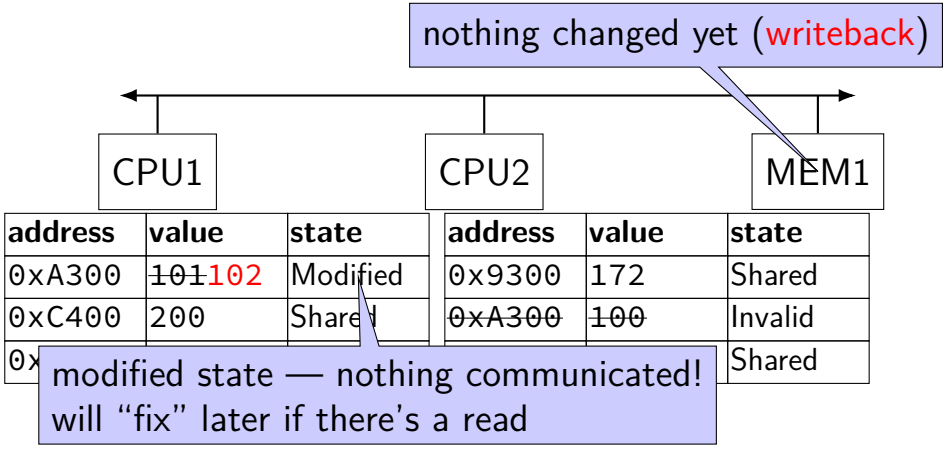
address	value	state
0xA300	~~100~~101	Modified
0xC400	200	Shared
0xE500	300	Shared

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

cache sees write: invalidate 0xA300

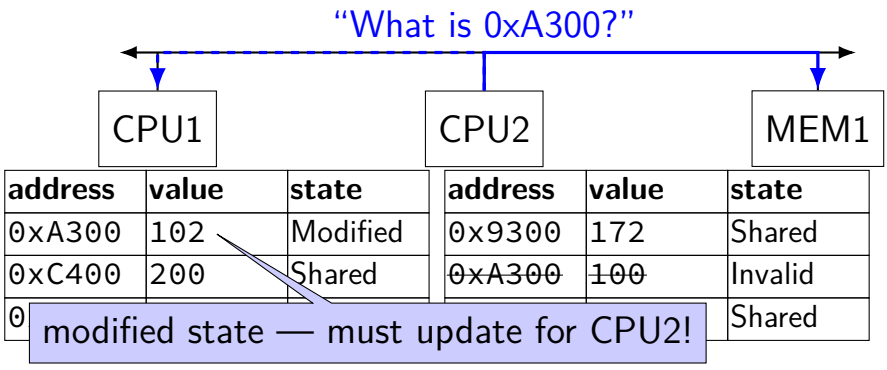
CPU1 writes 101 to 0xA300

MSI example



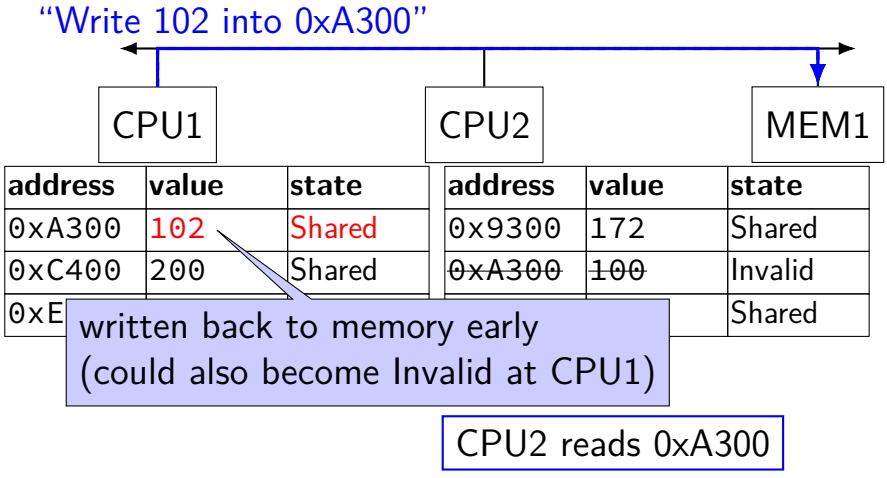
CPU1 writes 102 to 0xA300

MSI example

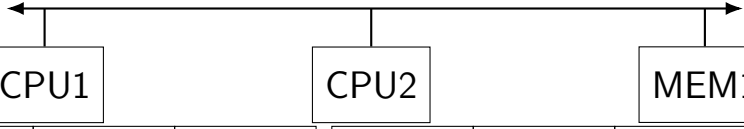


CPU2 reads 0xA300

MSI example


"Write 102 into 0xA300"
CPU1
CPU2
MEM1
address
value
state
0xA300
102
Shared
0xC400
200
Shared
0xE
address
value
state
0x9300
172
Shared
0xA300
100
Invalid
Shared
written back to memory early
(could also become Invalid at CPU1)
CPU2 reads 0xA300

MSI example



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

address	value	state
0x9300	172	Shared
~~0xA300~~	~~100~~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:

cache coherency exercise solution

action	0x1000–0x101f			0x2000–0x201f		
	CPU 1	CPU 2	CPU 3	CPU 1	CPU 2	CPU 3
	I	I	I	I	I	I
CPU 1: read 0x1000	S	I	I	I	I	I
CPU 2: read 0x1000	S	S	I	I	I	I
CPU 1: write 0x1000	M	I	I	I	I	I
CPU 1: read 0x2000	M	I	I	S	I	I
CPU 2: read 0x1000	S	S	I	S	I	I
CPU 2: write 0x2008	S	S	I	I	M	I
CPU 3: read 0x1008	S	S	S	I	M	I

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
  ...
```

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

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

Source: Intel 64 and IA-32 Software Developer's Manual, Volume 3A, Chapter 8

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:

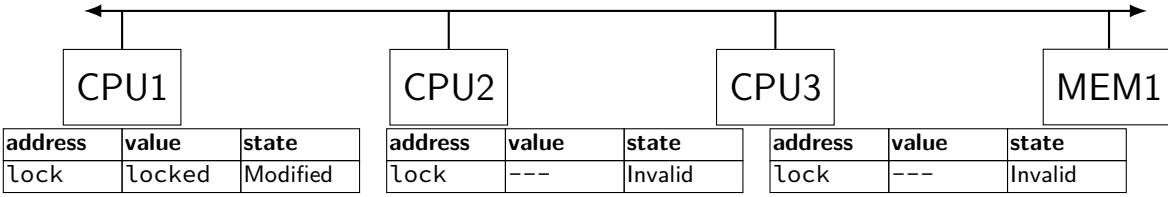
special instructions with stronger (and simpler) ordering rules

- 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

ping-ponging

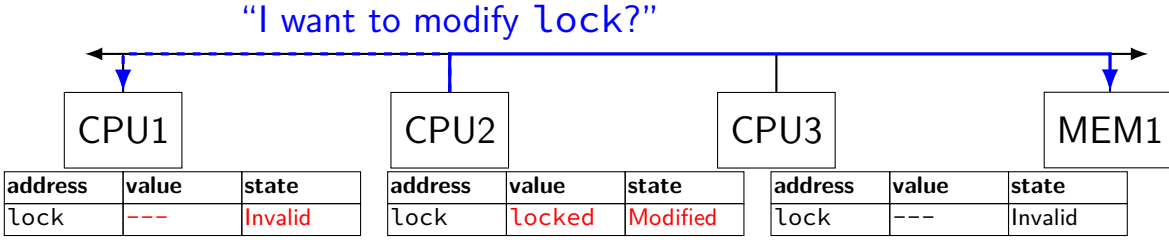


address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

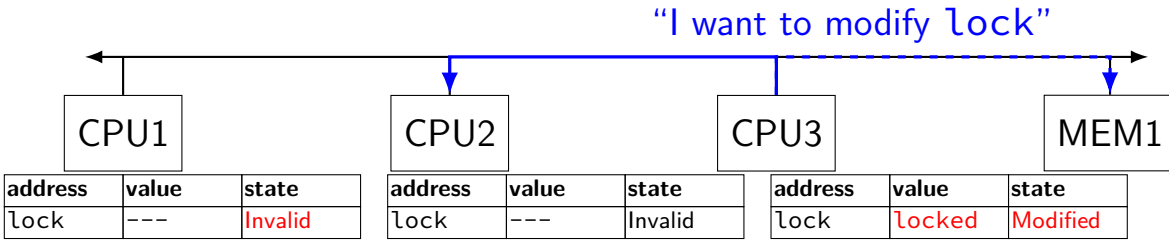
address	value	state
lock	---	Invalid

ping-ponging



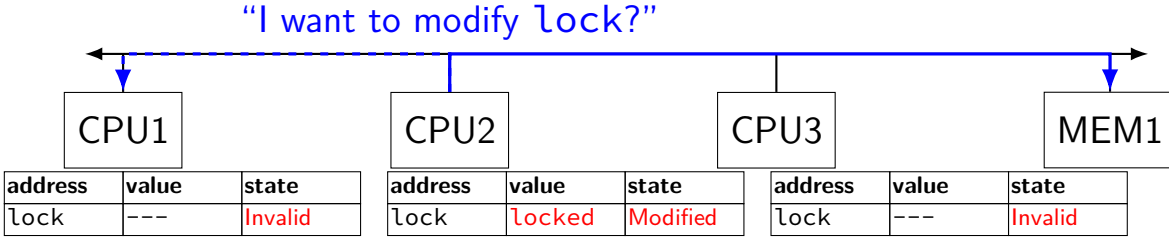
CPU2 read-modify-writes lock
(to see it is still locked)

ping-ponging



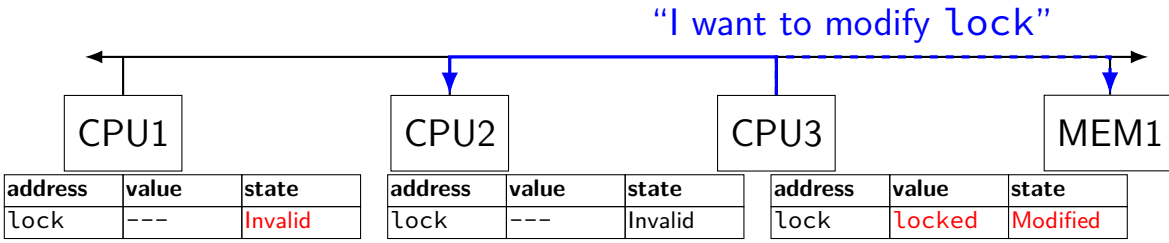
CPU3 read-modify-writes lock
(to see it is still locked)

ping-ponging



CPU2 read-modify-writes lock
(to see it is still locked)

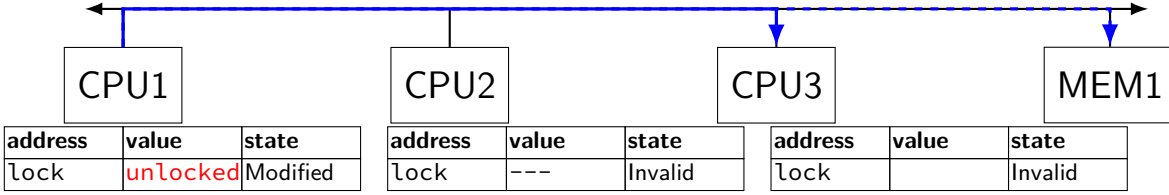
ping-ponging



CPU3 read-modify-writes lock
(to see it is still locked)

ping-ponging

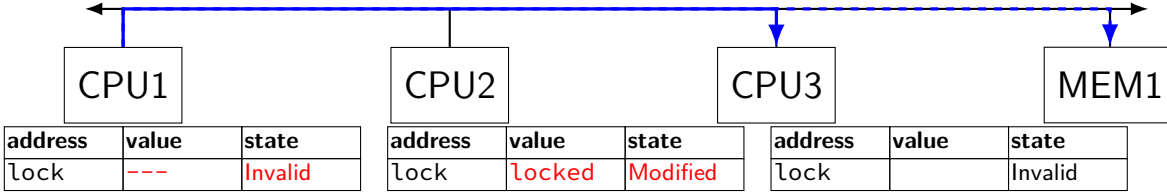
"I want to modify `lock`"



CPU1 sets lock to unlocked

ping-ponging

"I want to modify `lock`"



some CPU (this example: CPU2) acquires lock

ping-ponging

test-and-set problem: cache block "ping-pongs" between caches

- each waiting processor reserves block to modify
- could maybe wait until it determines modification needed — but not typical implementation

each transfer of block sends messages on bus

...so bus can't be used for real work

- like what the processor with the lock is doing

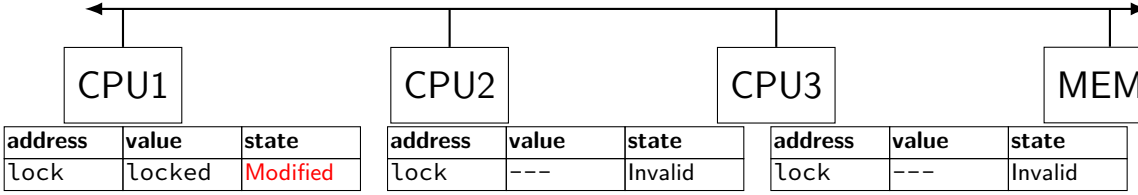
test-and-test-and-set (pseudo-C)

```
acquire(int *the_lock) {
    do {
        while (ATOMIC_READ(the_lock) == 0) { /* try again */ }
    } while (ATOMIC_TEST_AND_SET(the_lock) == ALREADY_SET);
}
```

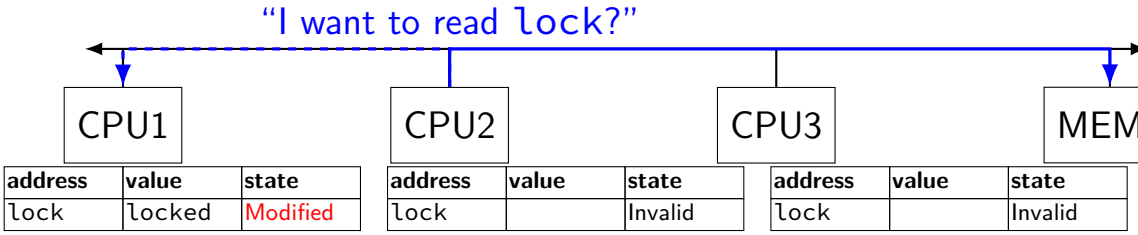
test-and-test-and-set (assembly)

```
acquire:
    cmp $0, the_lock          // test the lock non-atomically
            // unlike lock xchg --- keeps lock in Shared state!
    jne acquire               // try again (still locked)
    // lock possibly free
    // but another processor might lock
    // before we get a chance to
    // ... so try wtih atomic swap:
    movl $1, %eax             // %eax ← 1
    lock xchg %eax, the_lock  // swap %eax and the_lock
           // sets the_lock to 1
           // sets %eax to prior value of the_lock
    test %eax, %eax           // if the_lock wasn't 0 (someone else
    jne acquire               //   try again
    ret
```

less ping-ponging



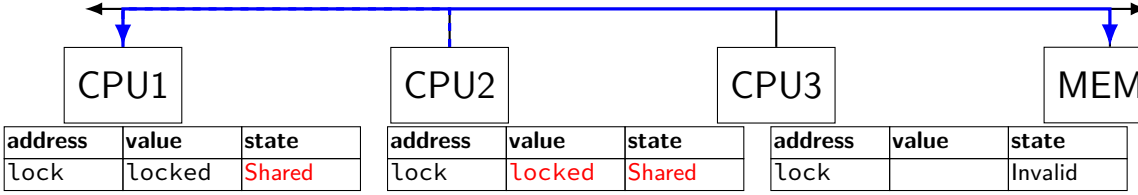
less ping-ponging



CPU2 reads lock
(to see it is still locked)

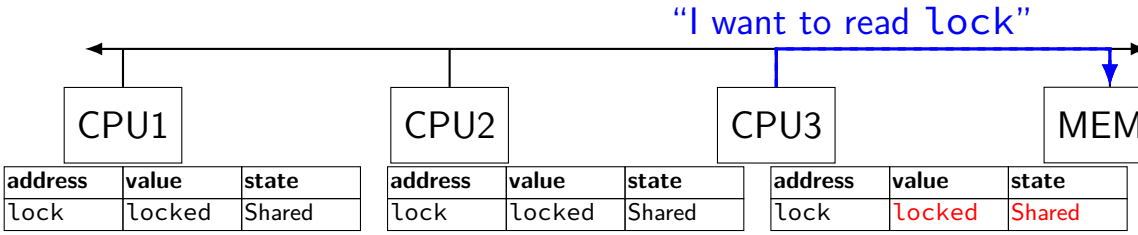
less ping-ponging

"set lock to locked"



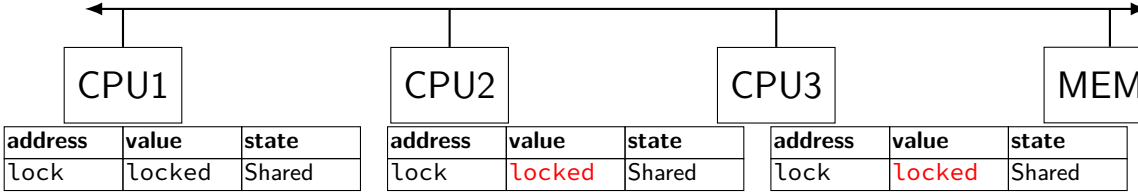
CPU1 writes back lock value,
then CPU2 reads it

less ping-ponging



CPU3 reads lock
(to see it is still locked)

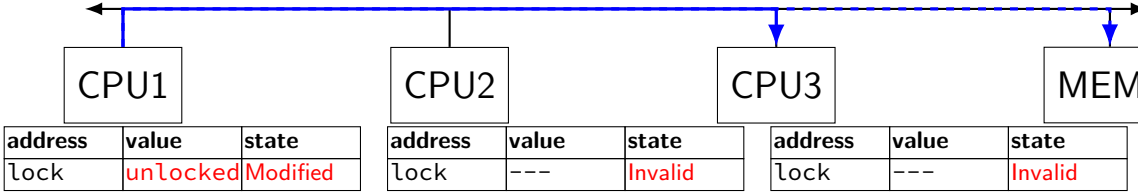
less ping-ponging



CPU2, CPU3 continue to read lock from cache
no messages on the bus

less ping-ponging

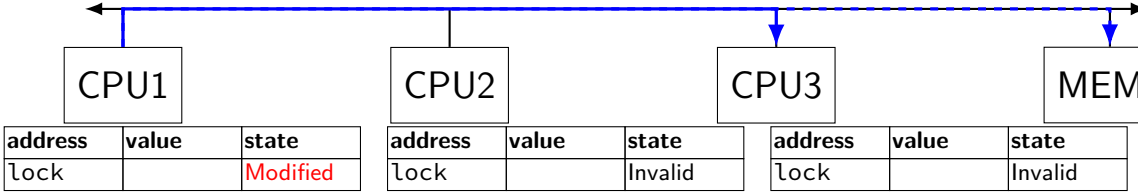
"I want to modify `lock`"



CPU1 sets lock to unlocked

less ping-ponging

"I want to modify `lock`"



some CPU (this example: CPU2) acquires lock
(CPU1 writes back value, then CPU2 reads + modifies it)

couldn't the read-modify-write instruction...

notice that the value of the lock isn't changing...

and keep it in the shared state

maybe — but extra step in "common" case
(swapping different values)

more room for improvement?

can still have a lot of attempts to modify locks after unlocked

there other spinlock designs that avoid this

- ticket locks
- MCS locks
- ...

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)

monitors with semaphores: locks

```
sem_t semaphore;  // initial value 1

Lock() {
    sem_wait(&semaphore);
}

Unlock() {
    sem_post(&semaphore);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

monitors with semaphores: [broken] cvs

start with only wait/signal:

```
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
    lock.Unlock();
    sem_wait(&threads_to_wakeup);
    lock.Lock();
}
Signal() {
    sem_post(&threads_to_wakeup);
}
```

problem: signal wakes up non-waiting threads (in the far future)

monitors with semaphores: cvs (better)

start with only wait/signal:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Signal() {
  sem_wait(&private_lock);
  if (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

monitors with semaphores: broadcast

now allows broadcast:

```
sem_t private_lock;  // initially 1
int num_waiters;
sem_t threads_to_wakeup;  // initially 0
Wait(Lock lock) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Broadcast() {
  sem_wait(&private_lock);
  while (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  sem_post(&private_lock);
}
```

building semaphore with monitors

```
pthread_mutex_t lock;
```

lock to protect shared state

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
```

lock to protect shared state

shared state: semaphore tracks a count

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

building semaphore with monitors

```
pthread_mutex_t lock;
unsigned int count;
/* condition, broadcast when becomes count > 0 */
pthread_cond_t count_is_positive_cv;
void down() {
    pthread_mutex_lock(&lock);
    while (!(count > 0)) {
        pthread_cond_wait(
            &count_is_positive_cv,
            &lock);
    }
    count -= 1;
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    /* count must now be
       positive, and at most
       one thread can go per
       call to Up() */
    pthread_cond_signal(
        &count_is_positive_cv
    );
    pthread_mutex_unlock(&lock);
}
```

lock to protect shared state

shared state: semaphore tracks a count

add cond var for each reason we wait

semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes