synchronization 2: locks / memory ordering

last time

pthread create/join

racing

where data is stored in stacks/etc.

(not) making locks from atomic load/store

making locks (started)

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

problem: user can hang the system:

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

naive interrupt enable/disable (1)

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

a simple race

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

recall?: out-of-order processors

processors execute instructons in different order

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

convenient optimization: execute loads/stores in different order

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

some x86 reordering restrictions

each core sees its own loads/stores in order

(if a core store 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 that 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?

special instructions with stronger ordering rules

special instructions that restirct 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;
    do {} while (note_from_bob);
    if (no_milk) {++milk;}
    note_from_alice = 2;
}
```

```
Alice:
  // don't set note_from_alice to 1,
  // since 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;
    do {} while (note_from_bob);
    if (no_milk) {++milk;}
    note_from_alice = 2;
}
```

```
Alice:
  // don't set note_from_alice to 1,
  // since 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;
    do {} while (note_from_bob);
    if (no_milk) {++milk;}
    note_from_alice = 2;
}
```

```
Alice:
  // don't set note_from_alice to 1,
  // since 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
```

pthreads and reordering

synchronizing pthreads functions prevent reordering

everything before function call actually happens before everything after

includes preventing some optimizations

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

not just pthread_mutex_lock/unlock!

includes pthread_create, pthread_join, ...

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 (1)

```
#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 is visible to other cores before loadir
  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 atmoic functions

used to implement std::atomic, etc.

predate std::atomic

builtin functions starting with `__sync` and `__atomic`

these are what xv6 uses

GCC: preventing reordering example (1)

```
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
  ...
```

GCC: preventing reordering example (2)

```
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;}
}
```

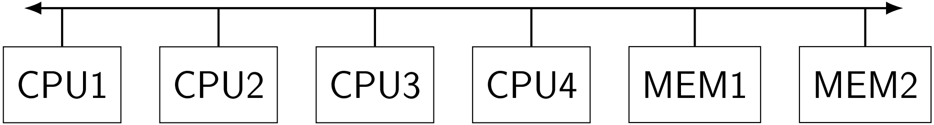
```
Alice:
  movl $1, note_from_alice
  mfence
.L2:
  movl note_from_bob, %eax
  testl %eax, %eax
  jne .L2
  ...
```

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

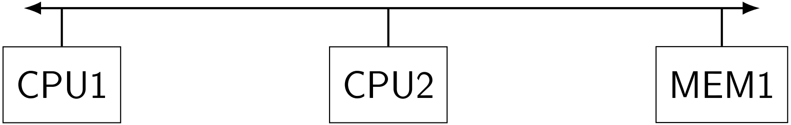
remember caches?

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



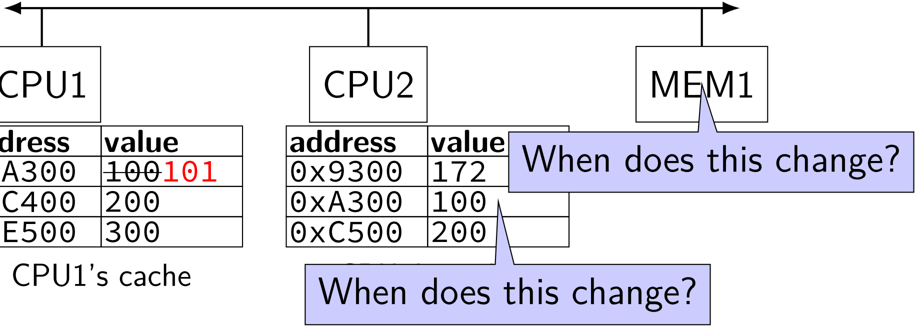
address	value
0xA300	100
0xC400	200
0xE500	300

CPU1's cache

address	value
0x9300	172
0xA300	100
0xC500	200

CPU2's cache

the cache coherency problem



address	value
0xA300	~~100~~101
0xC400	200
0xE500	300

CPU1's cache

address	value
0x9300	172
0xA300	100
0xC500	200

When does this change?

When does this change?

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

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

sample states:

- Modified: cache has updated value
- Shared: cache is only reading, has same as memory/others
- Invalid

scheme 1: MSI

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

scheme 1: MSI

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

scheme 1: MSI

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

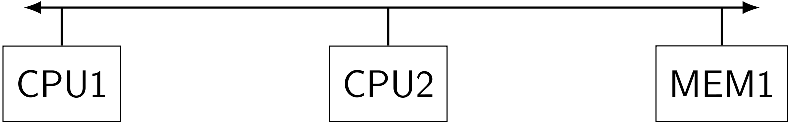
blue: transition requires sending message on bus

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

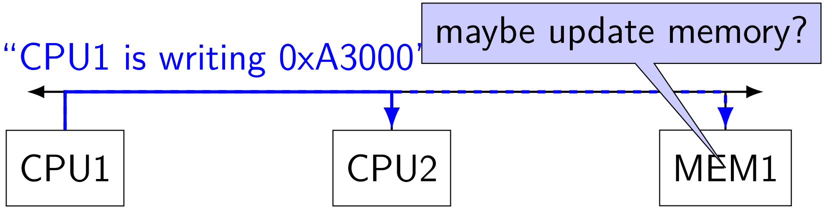
MSI example



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



maybe update memory?

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

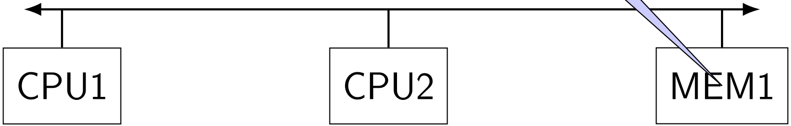
address	value	state
0x9300	172	Shared
~~0xA300~~	~~100~~	Invalid
0xC500	200	Shared

cache sees write: invalidate 0xA300

CPU1 writes 101 to 0xA300

MSI example

nothing changed yet (writeback)



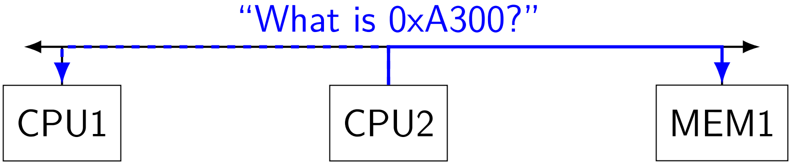
address	value	state
0xA300	~~101~~102	Modified
0xC400	200	Shared
0x		

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

modified state — nothing communicated!
will “fix” later if there’s a read

CPU1 writes 102 to 0xA300

MSI example



address	value	state
0xA300	102	Modified
0xC400	200	Shared
0		

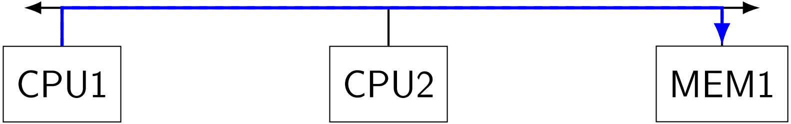
address	value	state
0x9300	172	Shared
~~0xA300~~	~~100~~	Invalid
		Shared

modified state — must update for CPU2!

CPU2 reads 0xA300

MSI example

“Write 102 into 0xA300”



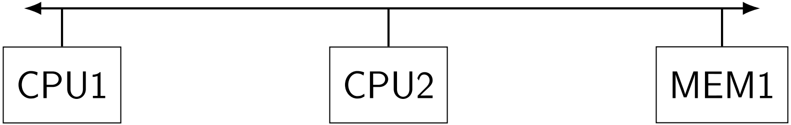
address	value	state
0xA300	102	Shared
0xC400	200	Shared
0xE		

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)

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 extensions

extra states for *unmodified* values where no other cache has a copy

- avoid sending "I am writing" message later

allow values to be sent directly between caches

- (MSI: value needs to go to memory first)

support not sending invalidate/etc. messages to *all* cores

- requires some tracking of what cores have each address
- only makes sense with non-shared-bus design

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.

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
                                       // sets %eax to prior value of t
    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
    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
                                                                of t

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

release:
    mfence                    // for memory order reasons
    movl $0, the_lock         // then, set the_lock to 0
    ret
```

set lock variable to 1 (locked)

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
                                                                of t

    test %eax, %eax
    jne acquire
    ret

release:
    mfence                        // for memory order reasons
    movl $0, the_lock             // then, set the_lock to 0
    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
                                                                  of t

    test %eax, %eax
    jne acquire
    ret

release:
    mfence                           // for memory order reasons
    movl $0, the_lock                // then, set the_lock to 0
    ret
```

release lock by setting it to 0 (unlocked)

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
                                                                of t
    test %eax, %eax
    jne acquire
    ret

release:
    mfence                       // for memory order reasons
    movl $0, the_lock            // then, set the_lock to 0
    ret
```

Intel's manual says:

no reordering of loads/stores across a `lock`

or `mfence` instruction

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;
}
```

append to singly-linked list

```c
/*
    assumption 1: other threads may be appending to list,
    but nodes are not being removed, reordered, etc.

    assumption 2: the processor will not previous reoreder stores
                  into *new_last_node to take place after the
                  store for the compare_and_swap
*/
void append_to_list(ListNode *head, ListNode *new_last_node) {
  ListNode *current_last_node = head;
  do {
    while (current_last_node->next) {
      current_last_node = current_last_node->next;
    }
  } while (
    !compare_and_swap(&current_last_node->next,
                      NULL, new_last_node)
  );
}
```

common atomic operation pattern

try to acquire lock, or update next pointer, or …

detect if try failed

if so, repeat

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.
  __sync_synchronize();
  ...
}
```

don't want to be waiting for lock held by non-running thread

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 as loop above

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                                                oads or stor
  // past                                                n's memory
  // refer
  __sync_synchronize();
  ...
}
```

avoid load store reordering (including by compiler)

on x86, `xchg` alone avoids processor's reordering

(but compiler might 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();
}
```

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();
}
```

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

```c
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;
  ...
}
```

spinlock problems

spinlocks can send a lot of messages on the shared bus

- makes every non-cached memory access slower...

wasting CPU time waiting for another thread

- could we do something useful instead?

spinlock problems

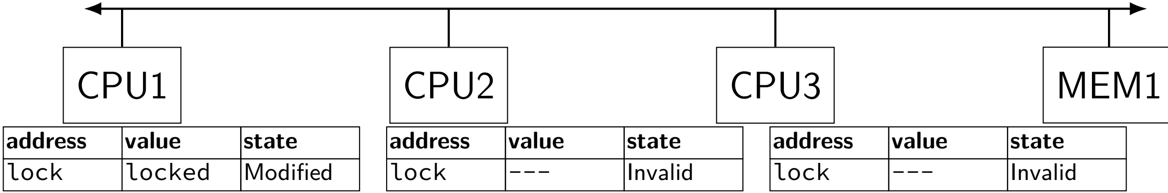
spinlocks can send a lot of messages on the shared bus

- makes every non-cached memory access slower…

wasting CPU time waiting for another thread

- could we do something useful instead?

ping-ponging

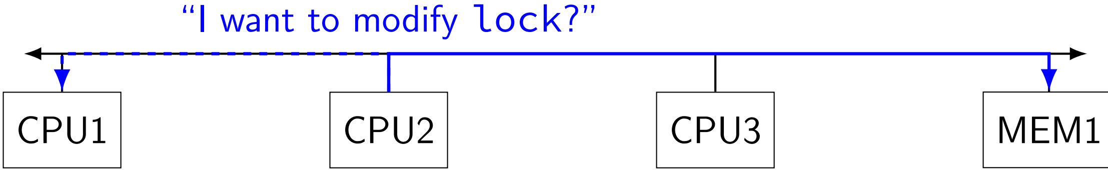


address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

ping-ponging



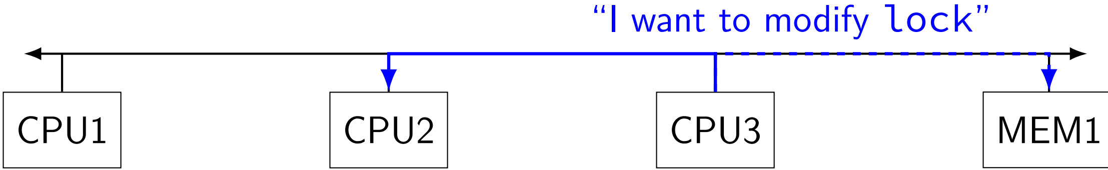
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

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

ping-ponging



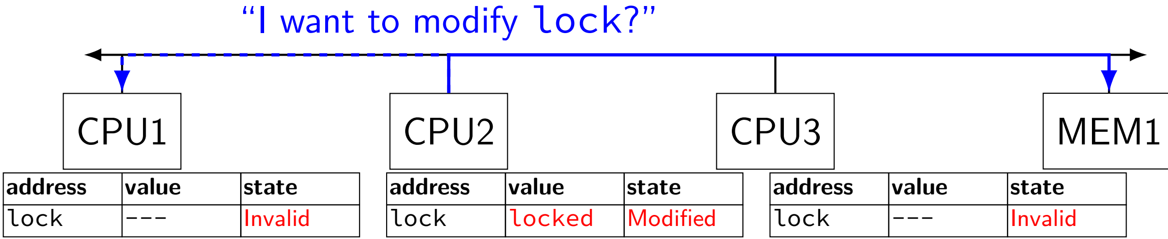
address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

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

ping-ponging



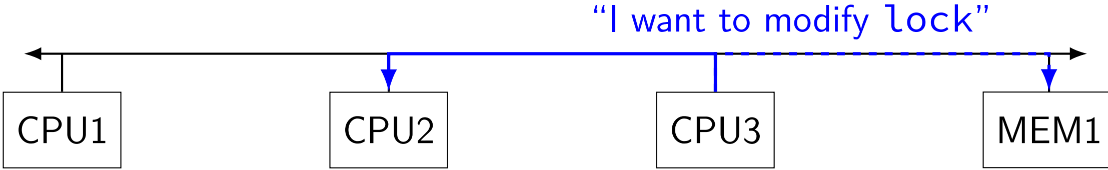
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

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

ping-ponging



address	value	state
lock	---	Invalid

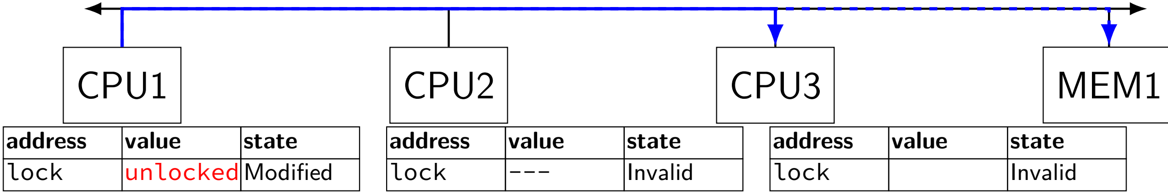
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

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

ping-ponging

“I want to modify `lock`”



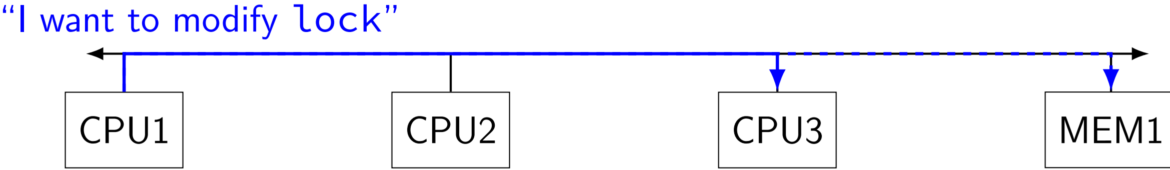
address	value	state
`lock`	`unlocked`	Modified

address	value	state
`lock`	`---`	Invalid

address	value	state
`lock`		Invalid

CPU1 sets lock to unlocked

ping-ponging


“I want to modify lock”

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

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

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

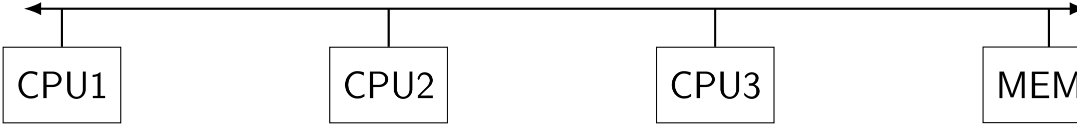
test-and-test-and-set (pseudo-C)

```
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



CPU1

CPU2

CPU3

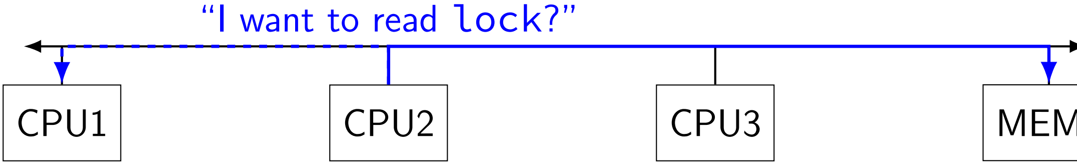
MEM

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

less ping-ponging


“I want to read lock?”

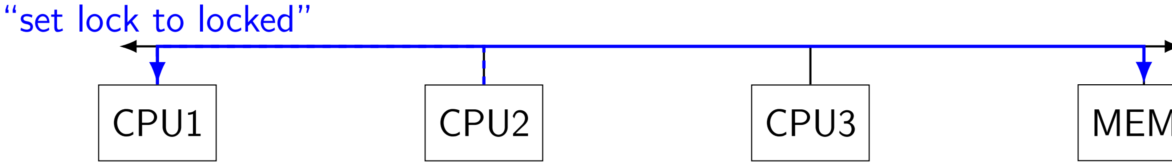
address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

address	value	state
lock		Invalid

CPU2 reads lock
(to see it is still locked)

less ping-ponging


“set lock to locked”

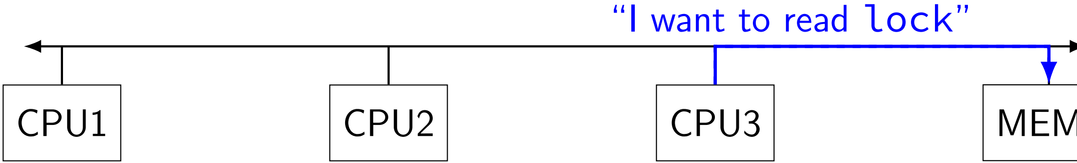
address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock		Invalid

CPU1 writes back lock value,
then CPU2 reads it

less ping-ponging



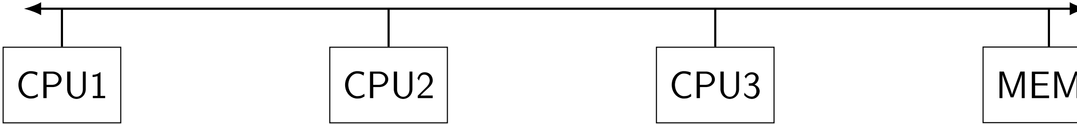
address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

CPU3 reads lock
(to see it is still locked)

less ping-ponging



address	value	state
lock	locked	Shared

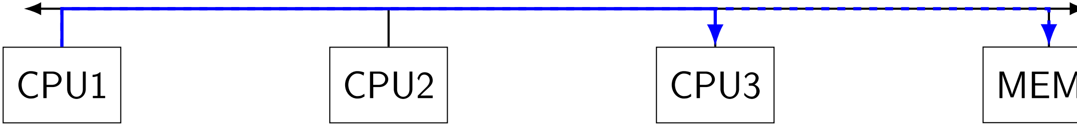
address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

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

less ping-ponging

“I want to modify `lock`”



address	value	state
lock	lockedunlocked	Modified

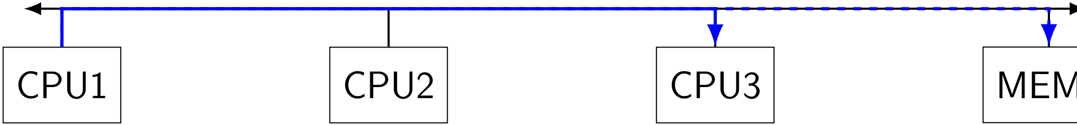
address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

CPU1 sets lock to unlocked

less ping-ponging

“I want to modify lock”



address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

address	value	state
lock		Invalid

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
- …

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)

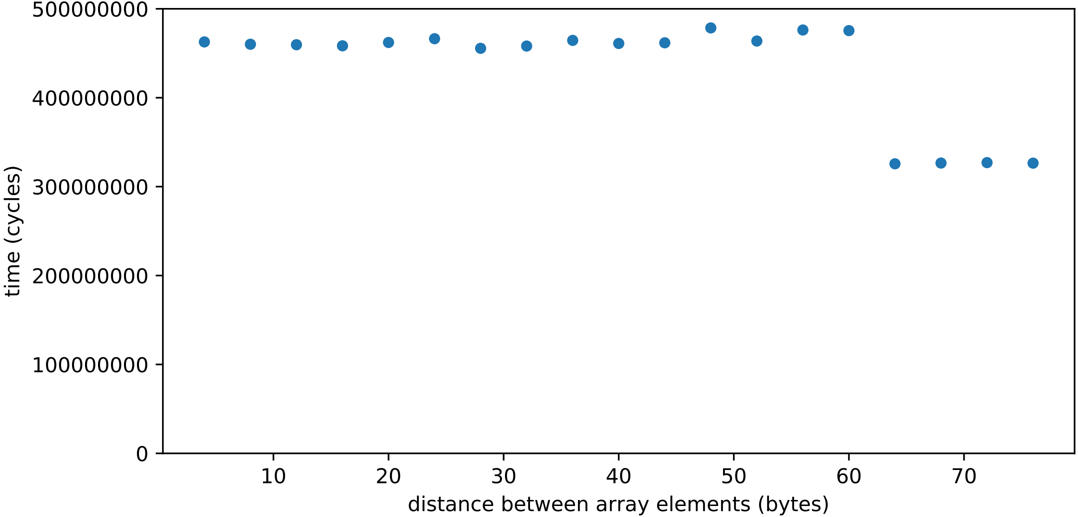
```
void *sum_up(void *raw_dest) {
    int *dest = (int *) raw_dest;
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {
        *dest += data[i];
    }
}

__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

spinlock problems

spinlocks can send a lot of messages on the shared bus

- makes every non-cached memory access slower...

wasting CPU time waiting for another thread

- could we do something useful instead?

problem: busy waits

```
  while(xchg(&lk->locked, 1) != 0)
    ;
```

what if it's going to be a while?

waiting for process that's waiting for I/O?

really would like to do something else with CPU instead...

mutexes: intelligent waiting

mutexes — locks that wait better

instead of running infinite loop, give away CPU

lock = go to sleep, add self to list

- sleep = scheduler runs something else

unlock = wake up sleeping thread

mutexes: intelligent waiting

mutexes — locks that wait better

instead of running infinite loop, give away CPU

lock = go to sleep, add self to list

sleep = scheduler runs something else

unlock = wake up sleeping thread

mutex implementation idea

shared list of waiters

spinlock protects list of waiters from concurrent modification

lock = use spinlock to add self to list, then wait without spinlock

unlock = use spinlock to remove item from list

mutex implementation idea

shared list of waiters

spinlock protects list of waiters from concurrent modification

lock = use spinlock to add self to list, then wait without spinlock

unlock = use spinlock to remove item from list

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

spinlock protecting `lock_taken` and `wait_queue`
only held for very short amount of time (compared to mutex itself)

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

tracks whether any thread has locked and not unlocked

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

list of threads that discovered lock is taken
and are waiting for it be free
these threads are not runnable

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    make current thread not runnable
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    make that thread runnable
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
     m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

instead of setting lock_taken to false
choose thread to hand-off lock to

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    make current thread not runnable
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    make that thread runnable
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
     m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

subtle: what if UnlockMutex() runs in between these lines?
reason why we make thread not runnable before releasing guard spinlock

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    make current thread not runnable
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
}
```

if woken up here, need to make sure schedul
doesn't run us on another core until we
switch to the scheduler (and save our regs)
xv6 solution: acquire ptable lock
Linux solution: seperate 'on cpu' flags

mutex: one possible implementation

```
struct Mutex {
    SpinLock guard_spinlock;
    bool lock_taken = false;
    WaitQueue wait_queue;
};
```

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    make current thread not runnable
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    make that thread runnable
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
     m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

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

recall: pthread mutex

```
#include <pthread.h>

pthread_mutex_t some_lock;
pthread_mutex_init(&some_lock, NULL);
// or: pthread_mutex_t some_lock = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&some_lock);
...
pthread_mutex_unlock(&some_lock);
pthread_mutex_destroy(&some_lock);
```

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?)

…

POSIX mutex restrictions

pthread_mutex rule: unlock from same thread you lock in

implementation I gave before — not a problem

...but there other ways to implement mutexes

- e.g. might involve comparing with "holding" thread ID

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 three extensions of locks for this:

- barriers
- counting semaphores
- condition variables

all implemented with read/modify/write instructions
+ queues of waiting threads

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);
```

generalizing locks

barriers are very useful

do things locks can't do

but can't do things locks can do

semaphores and condition variables are more general

can implement locks *and* barriers *and* …

generalizing locks: semaphores

semaphore has a non-negative integer **value** and two operations:

P() or **down** or **wait**:
wait for semaphore to become positive (> 0),
then decerement by 1

V() or **up** or **signal** or **post**:
increment semaphore by 1 (waking up thread if needed)

P, V from Dutch: *proberen* (test), *verhogen* (increment)

semaphores are kinda integers

semaphore like an integer, but...

cannot read/write directly

- down/up operaion only way to access (typically)
- exception: initialization

never negative — wait instead

- down operation wants to make negative? thread waits

reserving books

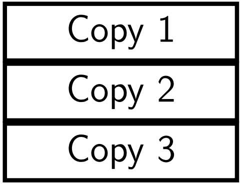
suppose tracking copies of library book...

```
Semaphore free_copies = Semaphore(3);
void ReserveBook() {
    // wait for copy to be free
    free_copies.down();
    ... // ... then take reserved copy
}

void ReturnBook() {
    ... // return reserved copy
    free_copies.up();
    // ... then wakeup waiting thread
}
```

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



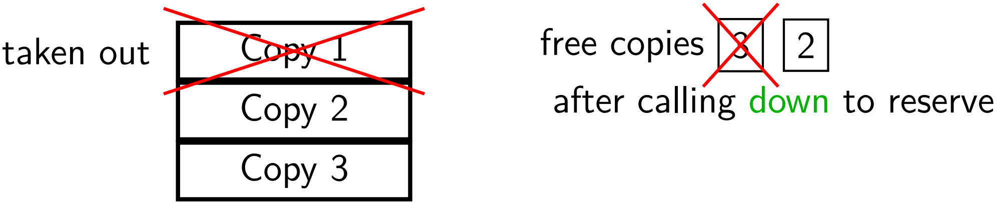
free copies $\boxed{3}$

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book



taken out

~~Copy 1~~

Copy 2

Copy 3

free copies $\boxed{\cancel{3}}$ $\boxed{2}$

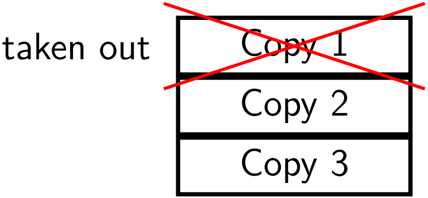
after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book



taken out

~~Copy 1~~

Copy 2

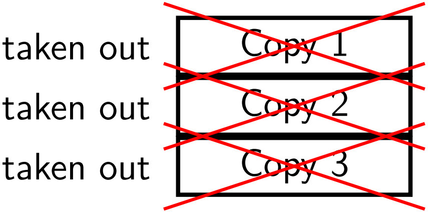
Copy 3

free copies $\boxed{2}$

after calling down to reserve

counting resources: reserving books

suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



free copies $\boxed{0}$

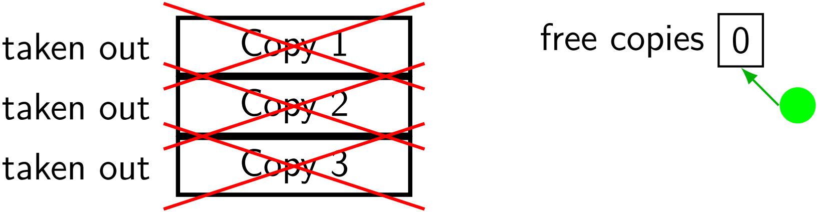
after calling down three times
to reserve all copies

counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book



taken out

taken out

taken out

~~Copy 1~~

~~Copy 2~~

~~Copy 3~~

free copies $\boxed{0}$

reserve book

call *down* again

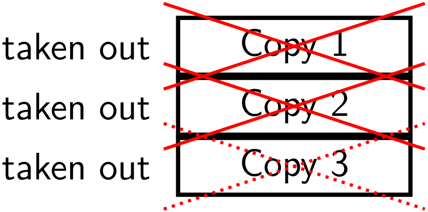
start waiting...

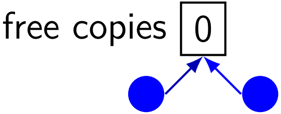
counting resources: reserving books

suppose tracking copies of same library book

non-negative integer count = # how many books used?

up = give back book; down = take book





return book

call *up*

release waiter

reserve book

call *down*

~~waiting~~

done waiting

implementing mutexes with semaphores

```
struct Mutex {
    Semaphore s; /* with inital value 1 */
    /* value = 1 --> mutex if free */
    /* value = 0 --> mutex is busy */
}

MutexLock(Mutex *m) {
    m->s.down();
}
MutexUnlock(Mutex *m) {
    m->s.up();
}
```

implementing join with semaphores

```
struct Thread {
    ...
    Semaphore finish_semaphore; /* with initial value 0 */
    /* value = 0: either thread not finished OR already joined */
    /* value = 1: thread finished AND not joined */
};
thread_join(Thread *t) {
    t->finish_semaphore->down();
}

/* assume called when thread finishes */
thread_exit(Thread *t) {
    t->finish_semaphore->up();
    /* tricky part: deallocating struct Thread safely? */
}
```

POSIX semaphores

```
#include <semaphore.h>
...
sem_t my_semaphore;
int process_shared = /* 1 if sharing between processes */;
sem_init(&my_semaphore, process_shared, initial_value);
...
sem_wait(&my_semaphore);  /* down */
sem_post(&my_semaphore);  /* up */
...
sem_destroy(&my_semaphore);
```

semaphore intuition

What do you need to wait for?

- critical section to be finished
- queue to be non-empty
- array to have space for new items

what can you count that will be 0 when you need to wait?

- # of threads that can start critical section now
- # of threads that can join another thread without waiting
- # of items in queue
- # of empty spaces in array

use up/down operations to maintain count