last time

reordering: processors and compilers

avoiding reordering: special instructions, compiler directives

- memory fence idea: everything before fence, then everything after

cache coherency (keeping caches in sync)

- baseline idea: write-through + snooping
- better than write-through: only one cache with modified version
- monitor reads/writes to keep in sync

false sharing

read/modify/write atomic instructions

spinlocks

spinlock problems

lock abstraction is not powerful enough

- lock/unlock operations don't handle "wait for event"
- common thing we want to do with threads
- solution: other synchronization abstractions

spinlocks waste CPU time more than needed

- want to run another thread instead of infinite loop
- solution: lock implementation integrated with scheduler

spinlocks can send a lot of messages on the shared bus

- more efficient atomic operations to implement locks

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
- condition variables / monitors
- counting semaphores
- reader/writer locks

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);
```

spinlock problems

lock abstraction is not powerful enough

- lock/unlock operations don't handle "wait for event"
- common thing we want to do with threads
- solution: other synchronization abstractions

spinlocks waste CPU time more than needed

- want to run another thread instead of infinite loop
- solution: lock implementation integrated with scheduler

spinlocks can send a lot of messages on the shared bus

- more efficient atomic operations to implement locks

mutexes: intelligent waiting

want: locks that wait better

- example: POSIX mutexes

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

want: locks that wait better

- example: POSIX mutexes

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

better lock 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

better lock 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

one possible implementation

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

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)

one possible implementation

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

tracks whether any thread has locked and not unlocked

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

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);
}
```

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);
}
```

one possible implementation

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

subtle: what if UnlockMutex runs on another core between these lines?
scheduler on another core might want to switch to it before it saves registers
issue to handle when marking threads not runnable for any reason
need to work with scheduler to prevent this

```
  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);
}
```

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 and scheduler subtly

core 0 (thread A)	core 1 (thread B)	core 2
start LockMutex		
acquire spinlock		
discover lock taken		
enqueue thread A		
thread A set not runnable		
release spinlock	start UnlockMutex	
	dequeue thread A	
	thread A set runnable	
		run scheduler
		scheduler switches to A
		…with old verison of registers
thread A runs scheduler		…
…finally saving registers		…

xv6 soln.: hold scheduler lock until thread A saves registers

Linux soln.: track that/check if thread A is still on core 0

mutex and scheduler subtly

core 0 (thread A)	core 1 (thread B)	core 2
start LockMutex		
acquire spinlock		
discover lock taken		
enqueue thread A		
thread A set not runnable		
release spinlock	start UnlockMutex	
	dequeue thread A	
	thread A set runnable	
		run scheduler
		scheduler switches to A
		…with old verison of registers
thread A runs scheduler		…
…finally saving registers		…

xv6 soln.: hold scheduler lock until thread A saves registers

Linux soln.: track that/check if thread A is still on core 0

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

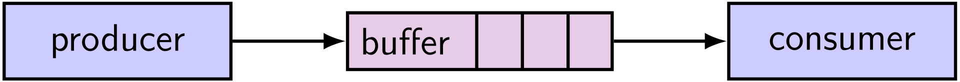
pthread_mutex rule: unlock from same thread you lock in

implementation I gave before — not a problem

...but there other ways to implement mutexes

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

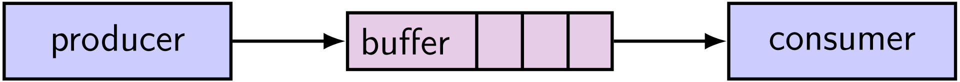
example: producer/consumer



shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

example: producer/consumer



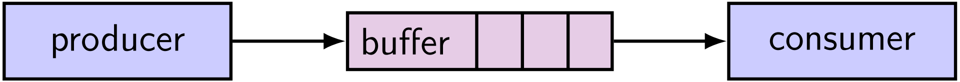
shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

- (might need to wait for each other to catch up)

example: producer/consumer



shared buffer (queue) of fixed size

- one or more producers inserts into queue
- one or more consumers removes from queue

producer(s) and consumer(s) don't work in lockstep

- (might need to wait for each other to catch up)

example: C compiler

- preprocessor $\rightarrow$ compiler $\rightarrow$ assembler $\rightarrow$ linker

monitors/condition variables

locks for mutual exclusion

condition variables for waiting for event

- operations: wait (for event); signal/broadcast (that event happened)

related data structures

monitor = lock + 0 or more condition variables + shared data

- Java: every object is a monitor (has instance variables, built-in lock, cond. var)
- pthreads: build your own: provides you locks + condition variables

monitor idea

a monitor

lock
shared data
condvar 1
condvar 2
…
operation1(…)
operation2(…)

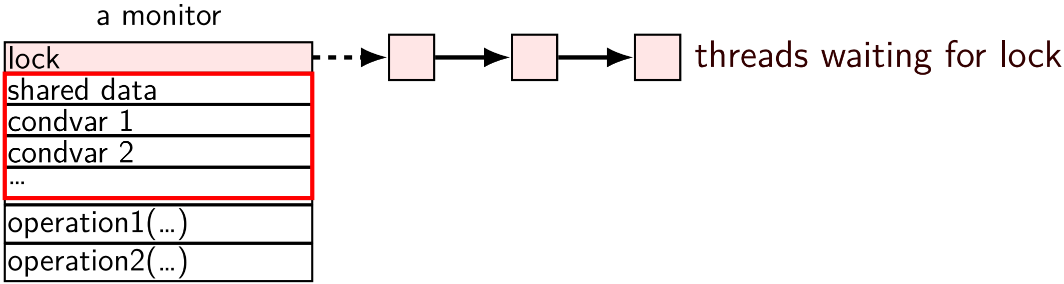
monitor idea

a monitor

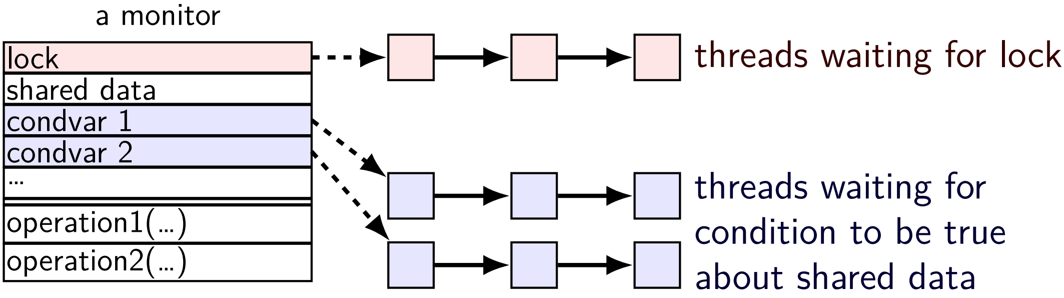
lock
shared data
condvar 1
condvar 2
…
operation1(…)
operation2(…)

lock must be acquired
before accessing
any part of monitor's stuff

monitor idea



monitor idea



condvar operations

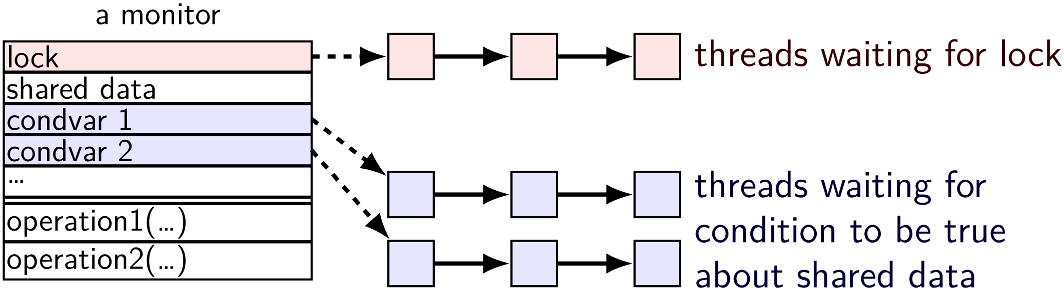
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

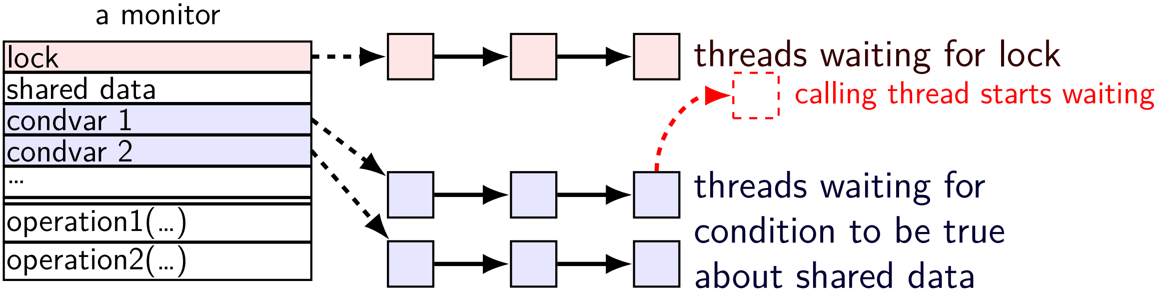
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

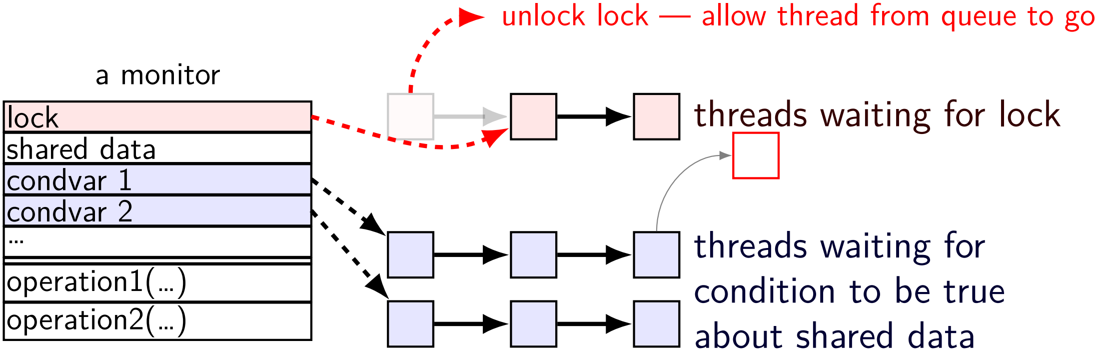
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

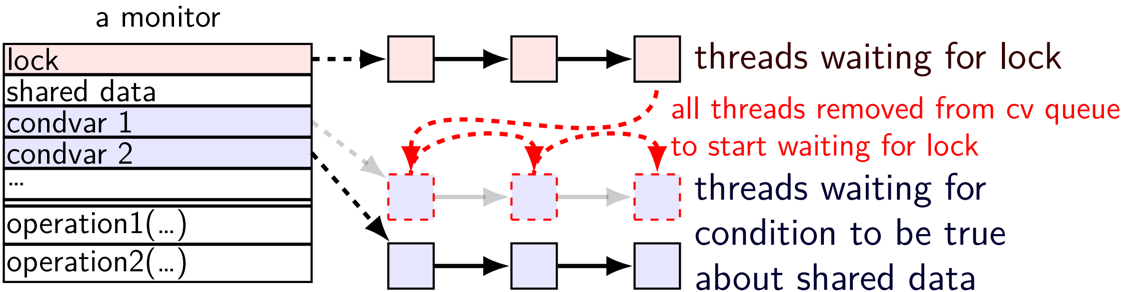
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

…and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

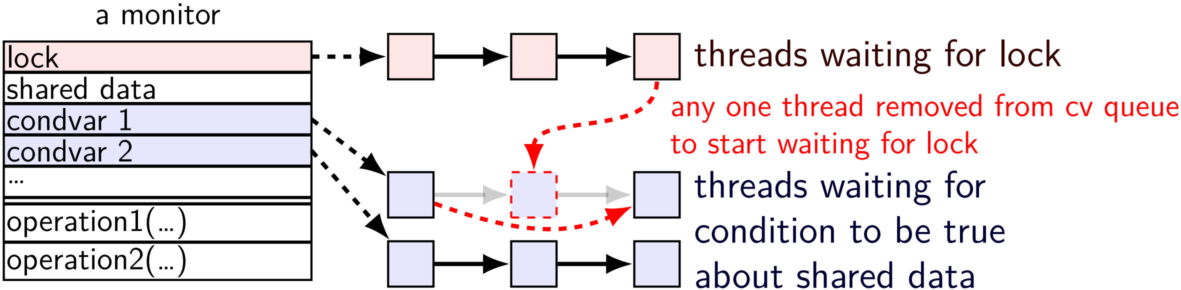
condvar operations:

Wait(cv, lock) — unlock lock, add current thread to cv queue

…and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;    // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

acquire lock before reading or writing `finished`

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;    // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

check whether we need to wait at all

(why a loop? we'll explain later)

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;    // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

know we need to wait
(finished can't change while we have lock)
so wait, releasing lock...

pthread cv usage

```
// MISSING: init calls, etc.
pthread_mutex_t lock;
bool finished;   // data, only accessed with after acquiring lock
pthread_cond_t finished_cv;  // to wait for 'finished' to be true

void WaitForFinished() {
  pthread_mutex_lock(&lock);
  while (!finished) {
    pthread_cond_wait(&finished_cv, &lock);
  }
  pthread_mutex_unlock(&lock);
}

void Finish() {
  pthread_mutex_lock(&lock);
  finished = true;
  pthread_cond_broadcast(&finished_cv);
  pthread_mutex_unlock(&lock);
}
```

allow all waiters to proceed
(once we unlock the lock)

WaitForFinish timeline 1

WaitForFinish thread	Finish thread
`mutex_lock(&lock)` (thread has lock)	
	`mutex_lock(&lock)` (start waiting for lock)

`while (!finished) ...`
`cond_wait(&finished_cv, &lock);`
(start waiting for cv)
	(done waiting for lock)

	`finished = true` `cond_broadcast(&finished_cv)`
(done waiting for cv) (start waiting for lock)	
	`mutex_unlock(&lock)`

(done waiting for lock)
`while (!finished) ...`
(finished now true, so return)
`mutex_unlock(&lock)`	

WaitForFinish timeline 2

WaitForFinish thread	Finish thread
	`mutex_lock(&lock)`
	`finished = true`
	`cond_broadcast(&finished_cv)`
	`mutex_unlock(&lock)`
`mutex_lock(&lock)`	
`while (!finished) ...`	
(finished now true, so return)	
`mutex_unlock(&lock)`	

why the loop

```
while (!finished) {
  pthread_cond_wait(&finished_cv, &lock);
}
```

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

why the loop

```
while (!finished) {
  pthread_cond_wait(&finished_cv, &lock);
}
```

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

pthread_cond_wait manual page:

> "Spurious wakeups ... may occur."

spurious wakeup = `wait` returns even though nothing happened

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

rule: never touch `buffer` without acquiring lock

otherwise: what if two threads simulatenously en/dequeue?

(both use same array/linked list entry?)

(both reallocate array?)

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

check if empty
if so, dequeue

okay because have lock

other threads can**not** dequeue here

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

wake one Consume thread
if any are waiting

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1	Thread 2
Produce()	
...lock	
...enqueue	
...signal	
...unlock	
	Consume()
	...lock
	...empty? no
	...dequeue
	...unlock
	return

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1	Thread 2
	Consume()
	…lock
	…empty? yes
	…unlock/start wait
Produce()	waiting for data_ready
…lock	
…enqueue	
…signal	stop wait
…unlock	lock
	…empty? no
	…dequeue
	…unlock
	return

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or …?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_rea
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_r
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

Thread 1	Thread 2	Thread 3
	Consume()	
	...lock	
	...empty? yes	
	...unlock/start wait	
Produce()	waiting for data_ready	
...lock		Consume()
...enqueue		waiting for lock
...signal	stop wait	
...unlock	waiting for lock	lock
		...empty? no
		...dequeue
		...unlock
	...lock	return
	...empty? yes	
	...unlock/start wait	

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or ...?

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

                                              );

                                              a_rea
                                              ck);

                                              );
    while (buffer.empty()) {
        pthread_cond_wait(&data_r
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

in pthreads: signalled thread not gaurenteed to hold lock next

alternate design: signalled thread gets lock next called "Hoare scheduling" not done by pthreads, Java, …

Thread 1	Thread 2	Thread 3
	Consume()	
	…lock	
	…empty? yes	
	…unlock/start wait	
Produce()	waiting for data_ready	
…lock		Consume()
…enqueue		waiting for lock
…signal	stop wait	
…unlock	waiting for lock	lock
		…empty? no
		…dequeue
		…unlock
	…lock	return
	…empty? yes	
	…unlock/start wait	

0 iterations: Produce() called before Consume()
1 iteration: Produce() signalled, probably
2+ iterations: spurious wakeup or …?

Hoare versus Mesa monitors

Hoare-style monitors

signal 'hands off' lock to awoken thread

Mesa-style monitors

any eligible thread gets lock next
(maybe some other idea of priority?)

every current threading library I know of does Mesa-style

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consum
    pt
    wh
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

correct (but slow?) to replace with:

`pthread_cond_broadcast(&space_ready);`

(just more "spurious wakeups")

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}

Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

correct but slow to replace `data_ready` and `space_ready` with 'combined' condvar `ready` and use broadcast
(just more "spurious wakeups")

monitor pattern

```
pthread_mutex_lock(&lock);
while (!condition A) {
    pthread_cond_wait(&condvar_for_A, &lock);
}
... /* manipulate shared data, changing other conditions */
if (set condition B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* or signal, if only one thread cares */
}
if (set condition C) {
    pthread_cond_broadcast(&condvar_for_C);
    /* or signal, if only one thread cares */
}
...
pthread_mutex_unlock(&lock)
```

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for entire operation:

verifying condition (e.g. buffer not full) *up to and including* manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write loop calling cond_wait to wait for condition X

broadcast/signal condition variable every time you change X

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for entire operation:

- verifying condition (e.g. buffer not full) *up to and including* manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write loop calling cond_wait to wait for condition X

broadcast/signal condition variable every time you change X

correct but slow to...

- broadcast when just signal would work
- broadcast or signal when nothing changed
- use one condvar for multiple conditions

mutex/cond var init/destroy

```
pthread_mutex_t mutex;
pthread_cond_t cv;
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cv, NULL);
// --OR--
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

// and when done:
...
pthread_cond_destroy(&cv);
pthread_mutex_destroy(&mutex);
```

backup slides

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

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;
  ...
}
```

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

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) {
    ...
}
```

spinlock problems

lock abstraction is not powerful enough

- lock/unlock operations don't handle "wait for event"
- common thing we want to do with threads
- solution: other synchronization abstractions

spinlocks waste CPU time more than needed

- want to run another thread instead of infinite loop
- solution: lock implementation integrated with scheduler

spinlocks can send a lot of messages on the shared bus

- more efficient atomic operations to implement locks

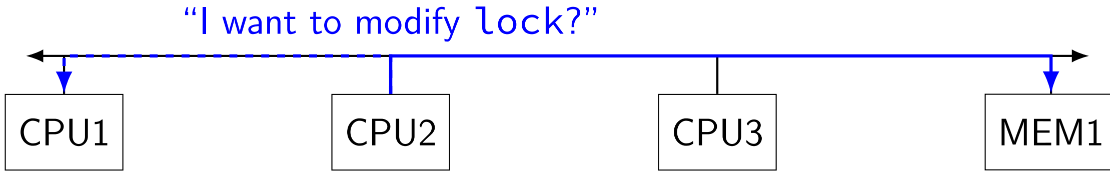
ping-ponging

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

ping-ponging



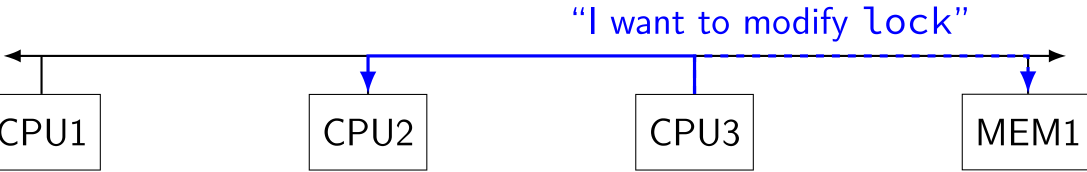
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

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

ping-ponging



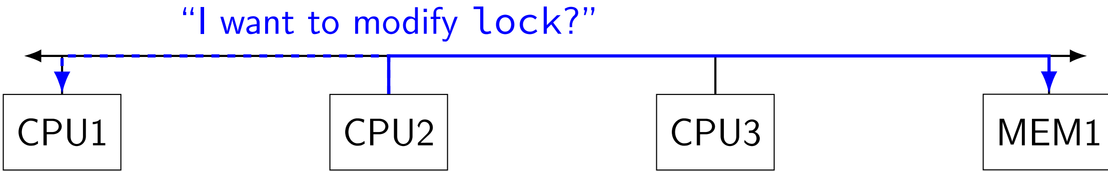
address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

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

ping-ponging



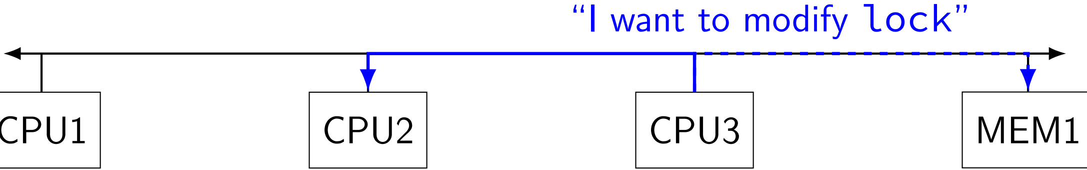
address	value	state
lock	---	Invalid

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
- 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
```

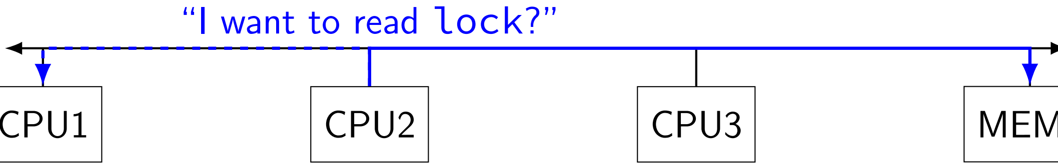
less ping-ponging

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

less ping-ponging



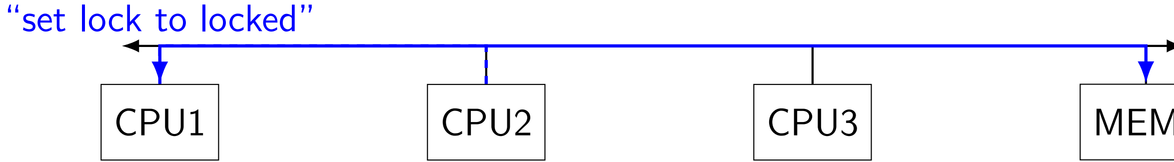
address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

address	value	state
lock		Invalid

CPU2 reads lock
(to see it is still locked)

less ping-ponging



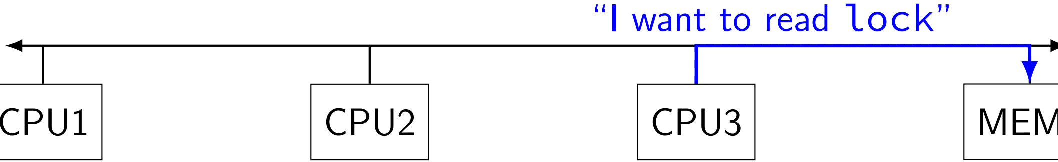
address	value	state
lock	locked	Shared

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`	`unlocked`	Modified

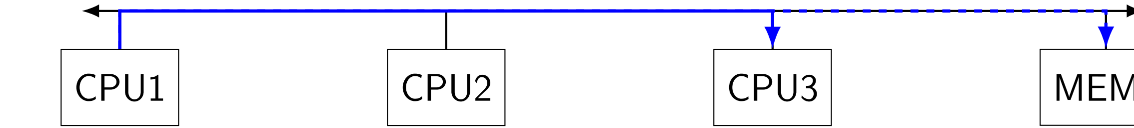
address	value	state
`lock`	`---`	Invalid

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

CPU1 sets lock to unlocked

less ping-ponging

"I want to modify `lock`"



address	value	state
`lock`		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
- ...