



## last time

deadlock: X wait for Y [possibly indirectly] wait for X

deadlock requirements

- hold and wait

- circular dependency

avoiding deadlock:

- lock order

- undo and retry

# anonymous feedback (1)

“As a professional, it’s easy to gloss over things that seem obvious to you, but aren’t obvious to others. I appreciated how Prof. Skadron took his time explaining all the concepts in great detail, often repeating things. Because of his clarity, I also think there weren’t nearly as many (unnecessary or irrelevant) questions taking time away from the content. I also liked how he broke down the code in the examples and walked through it with the class so everyone was on the same page. Sometimes Prof. Reiss will speak very fast and I can’t quite grasp the words he’s saying, even if I slow the recording down. Usually, Prof. Reiss’s lectures feel rushed and are personally stressful to watch. It feels to me that we fly through the material without getting the chance to fully understand it, so while there are in-class examples, we have an incomplete understanding of multiple examples rather than a complete understanding of a few.”

would like to know specifics re: glossing over things  
agree probably should watch for rushing  
selfishly like getting questions, so maybe I have bad incentives...  
some decision re: in-class exercises to not explain code if giving time to  
read exercise — bad choice?

## anonymous feedback (2)

“Quiz 8 is too difficult. I did the lectures, readings, and supplementary readings and none of them, including the examples and exercises we did in class, even approached the complexity of the code snippets and questions in quiz 8. I understand the concepts but thinking through race conditions and deadlock is difficult and error-prone when the provided code is so arduous. The concepts could have been tested with much simpler code.”

“...I want to clarify that I'm not referring to Questions 5 or 6 where the readability and intent of the code are good/clear, but Questions 2-4 where things are much more confusing and complex than examples given and readings.”

# beyond locks

in practice: want more than locks for synchronization

for waiting for arbitrary events (without CPU-hogging-loop):

- monitors

- semaphores

for common synchronization patterns:

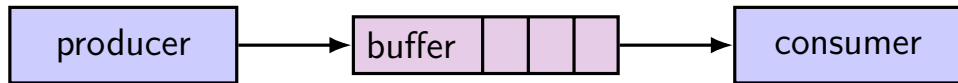
- barriers

- reader-writer locks

higher-level interface:

- transactions

## example: producer/consumer

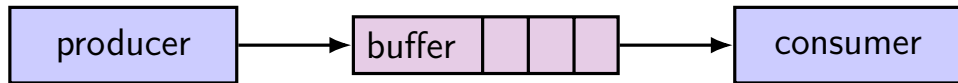


shared buffer (queue) of fixed size

one or more producers inserts into queue

one or more consumers removes from queue

## example: producer/consumer



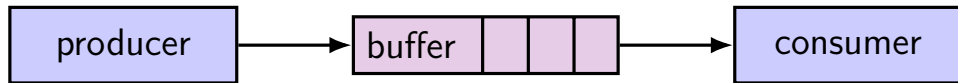
shared buffer (queue) of fixed size

- 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 → compiler → assembler → linker



# monitors/condition variables

**locks** for mutual exclusion

**condition variables** for waiting for event

represents **list of waiting threads**

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)

pthread: 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

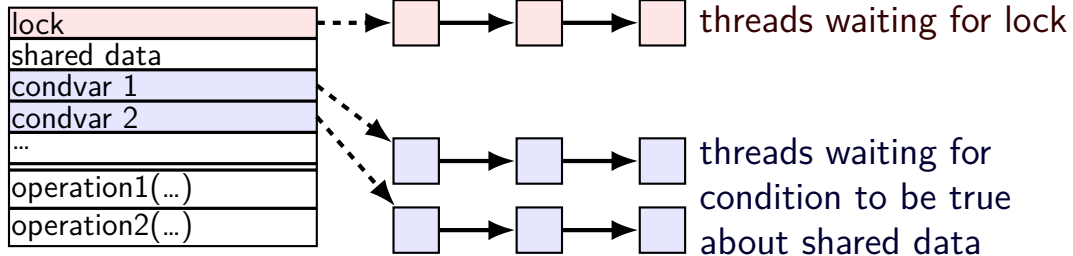
a monitor

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



# monitor idea

a monitor



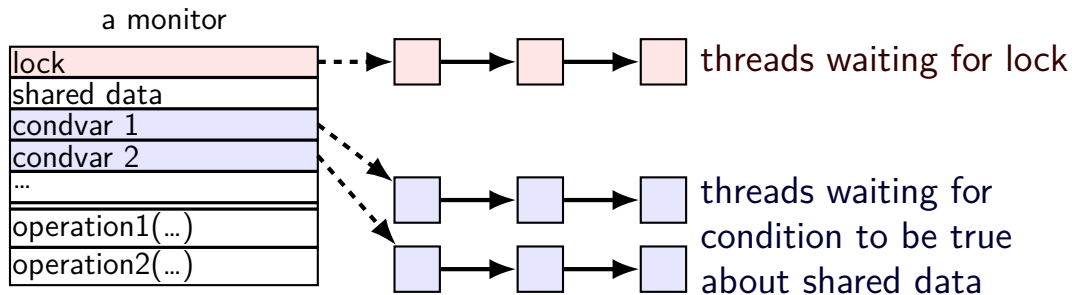
# condvar operations

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

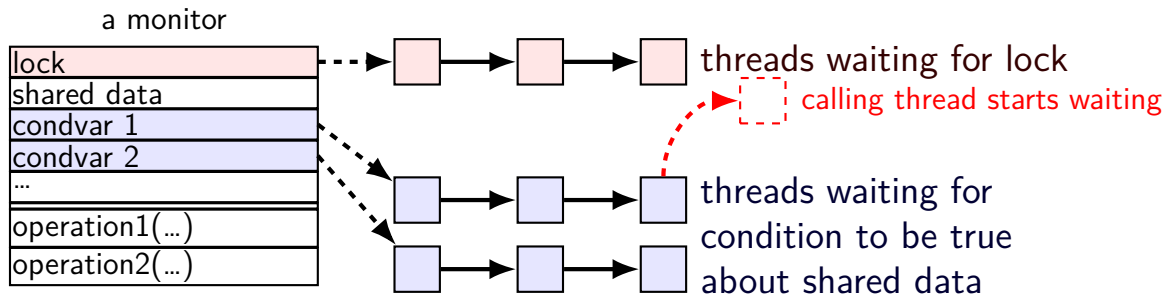
condvar operations:

**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

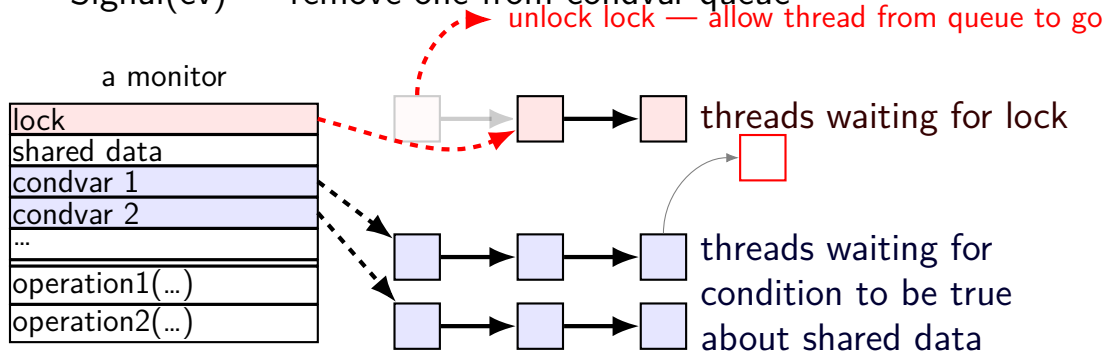
condvar operations:

**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





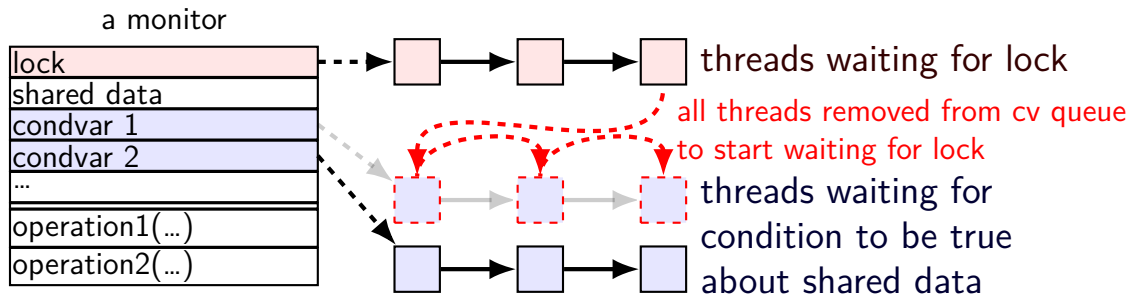
# condvar operations

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



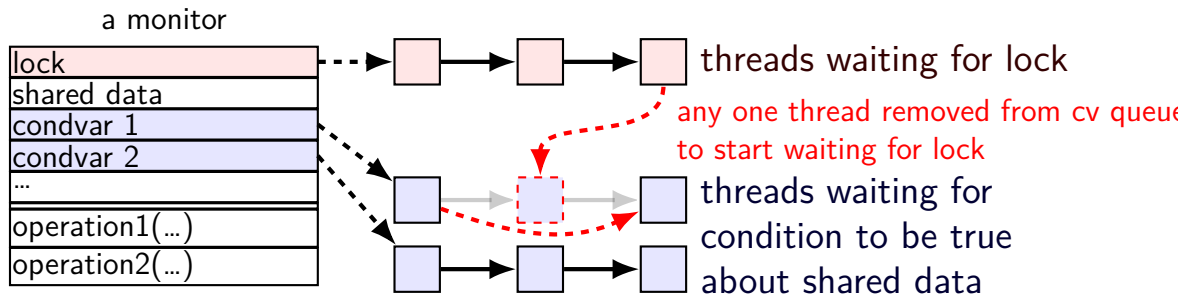
# condvar operations

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);
```

```
}
```

acquire lock before  
reading or writing finished

```
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);  
}
```

check whether we need to wait at all  
(why a loop? we'll explain later)

```
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);
```

```
}
```

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;
```

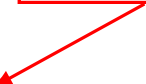
```
bool finished;    // data, only accessed with after acquiring lock
```

```
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
<code>mutex_lock(&amp;lock)</code> (thread has lock)	
	<code>mutex_lock(&amp;lock)</code> (start waiting for lock)
<code>while (!finished) ...</code> <code>cond_wait(&amp;finished_cv, &amp;lock);</code> (start waiting for cv)	(done waiting for lock)
	<code>finished = true</code> <code>cond_broadcast(&amp;finished_cv)</code>
(done waiting for cv) (start waiting for lock)	
	<code>mutex_unlock(&amp;lock)</code>
(done waiting for lock) <code>while (!finished) ...</code> (finished now true, so return) <code>mutex_unlock(&amp;lock)</code>	



## WaitForFinish timeline 2

WaitForFinish thread	Finish thread
	<code>mutex_lock(&amp;lock)</code> <code>finished = true</code> <code>cond_broadcast(&amp;finished_cv)</code> <code>mutex_unlock(&amp;lock)</code>
<code>mutex_lock(&amp;lock)</code> <code>while (!finished) ...</code> (finished now true, so return) <code>mutex_unlock(&amp;lock)</code>	

## 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  
simultaneously 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 cannot 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);  
}
```

wake one Consume thread  
*if any are waiting*

```
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;  
}
```

Thread 1

Produce()
...lock
...enqueue
...signal
...unlock

Thread 2

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_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

Produce()
...lock
...enqueue
...signal
...unlock

Thread 2

Consume()
...lock
...empty? yes
...unlock/start wait
waiting for data_ready
stop wait
waiting for lock
...lock
...empty? yes
...unlock/start wait

Thread 3

Consume()
waiting for lock
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;
```

in pthreads: signalled thread not  
guaranteed to hold lock next

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

```
pthread_cond_wait(&data_ready, &lock);
}
item = buffer.dequeue();
pthread_mutex_unlock(&lock);
return item;
}
```

Thread 1

```
Produce()
...lock
...enqueue
...signal
...unlock
```

Thread 2

```
Consume()
...lock
...empty? yes
...unlock/start wait

waiting for
data_ready

stop wait

waiting for
lock

...lock
...empty? yes
...unlock/start wait
```

Thread 3

```
Consume()
waiting for
lock

lock
...empty? no
...dequeue
...unlock
return
```

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;
}
```

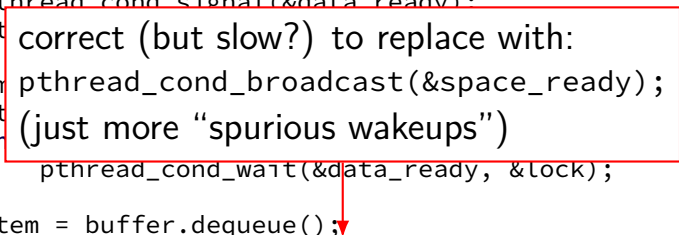
# bounded buffer producer/consumer

```
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);
}
Consume() {
    pthread_cond_broadcast(&space_ready);
    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:  
(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 A) {
    pthread_cond_broadcast(&condvar_for_A);
    /* or signal, if only one thread cares */
}
if (set condition B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* 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);
```

# wait for both finished

```
// MISSING: init calls, etc.
```

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished_cv;
```

```
void WaitForBothFinished() {  
    pthread_mutex_lock(&lock);  
    while (_____) {  
        pthread_cond_wait(&both_finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish(int index) {  
    pthread_mutex_lock(&lock);  
    finished[index] = true;  
    -----  
    pthread_mutex_unlock(&lock);  
}
```

# wait for both finished

*// MISSING: init calls, etc.*

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished_cv;
```

```
void WaitForBothFinished() {  
    pthread_mutex_lock(&lock);  
    while ( ) {  
        pthread_cond_wait(&both_finished_cv, &lock);  
    }  
    pthread_mutex_unlock(&lock);  
}
```

```
void Finish(int index) {  
    pthread_mutex_lock(&lock);  
    finished[index] = true;  
    -----  
    pthread_mutex_unlock(&lock);  
}
```

- A. `finished[0] && finished[1]`
- B. `finished[0] || finished[1]`
- C. `!finished[0] || !finished[1]`
- D. `finished[0] != finished[1]`
- E. something else

# wait for both finished

*// MISSING: init calls, etc.*

```
pthread_mutex_t lock;  
bool finished[2];  
pthread_cond_t both_finished;
```

```
void WaitForBothFinished
```

```
pthread_mutex_lock(&lock);
```

```
while ( _____ )
```

```
pthread_cond_wait(&both_finished_cv, &lock);
```

```
}
```

```
pthread_mutex_unlock(&lock);
```

```
}
```

```
void Finish(int index) {
```

```
pthread_mutex_lock(&lock);
```

```
finished[index] = true;
```

```
pthread_mutex_unlock(&lock);
```

```
}
```

A. pthread\_cond\_signal(&both\_finished\_cv)

B. pthread\_cond\_broadcast(&both\_finished\_cv)

C. if (finished[1-index])

pthread\_cond\_signal(&both\_finished\_cv);

D. if (finished[1-index])

pthread\_cond\_broadcast(&both\_finished\_cv);

E. something else

# monitor exercise: barrier

suppose we want to implement a one-use barrier; fill in blanks:

```
struct BarrierInfo {
    pthread_mutex_t lock;
    int total_threads; // initially total # of threads
    int number_reached; // initially 0
    -----
};
void BarrierWait(BarrierInfo *b) {
    pthread_mutex_lock(&b->lock);
    ++b->number_reached;
    if (b->number_reached == b->total_threads) {
        -----
    } else {
        -----
        -----
    }
    pthread_mutex_unlock(&b->lock);
}
```



# transactions

transaction: set of operations that occurs atomically

idea: something higher-level handles locking, etc.:

```
BeginTransaction();  
int FromOldBalance = GetBalance(FromAccount);  
int ToOldBalance = GetBalance(ToAccount);  
SetBalance(FromAccount, FromOldBalance - 100);  
SetBalance(ToAccount, FromOldBalance + 100);  
EndTransaction();
```

idea: library/database/etc. makes “transaction” happens all at once

# consistency / durability

“happens all at once” = could mean:

locking to make sure no other operations interfere (consistency)

making sure on crash, no partial transaction seen (durability)

(some systems provide both, some provide only one)

we'll just talk about implementing consistency

# implementing consistency: simple

simplest idea: only one run transaction at a time

# implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: undo everything, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

# implementing consistency: locking

everytime something read/written: acquire associated lock

on end transaction: release lock

if deadlock: **undo everything**, go back to BeginTransaction(), retry

how to undo?

one idea: keep list of writes instead of writing

apply writes only at EndTransaction()

# implementing consistency: optimistic

on read: copy version # for value read

on write: record value to be written, but don't write yet

on end transaction:

- acquire locks on everything

- make sure values read haven't been changed since read

if they have changed, just retry transaction

# life homework (pseudocode)

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

# life homework

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

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

parallel version: produce parts of grid in different threads

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



# backup slides

# producer/consumer signal?

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

```
Produce(item) {  
    pthread_mutex_lock(&lock);  
    buffer.enqueue(item);  
    /* GOOD CODE: pthread_cond_signal(&data_ready); */  
    /* BAD CODE: */  
    if (buffer.size() == 1)  
        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;  
}
```

## bad case (setup)

thread 0	1	2	3
Consume(): lock empty? wait on cv	Consume(): lock empty? wait on cv	Produce(): lock	Produce():

# bad case

thread 0	1	2	3
Consume(): lock empty? wait on cv	Consume(): lock empty? wait on cv	Produce(): lock  enqueue size = 1? signal unlock	Produce(): wait for lock  gets lock enqueue size $\neq$ 1: don't signal unlock
wait for lock			
gets lock dequeue	still waiting		

# monitor exercise: ConsumeTwo

suppose we want producer/consumer, but...

but change Consume() to ConsumeTwo() which returns a **pair of values**

and don't want two calls to ConsumeTwo() to wait...  
with each getting one item

what should we change below?

```
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;  
}
```

# monitor exercise: ordering

suppose we want producer/consumer, but...

but want to ensure first call to Consume() **always** returns first

(no matter what ordering cond\_signal/cond\_broadcast use)

```
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;  
}
```



**backup slides**



# using atomic exchange?

example: OS wants something done by whichever core tries first  
does not want it started twice!

if two cores try at once, only one should do it

```
int global_flag = 0;
void DoThingIfFirstToTry() {
    int my_value = 1;
    AtomicExchange(&my_value, &global_flag);
    if (my_value == 0) {
        /* flag was zero before, so I was first!*/
        DoThing();
    } else {
        /* flag was already 1 when we exchanged */
        /* I was second, so some other core is handling it */
    }
}
```

## 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);
```

# life homework even/odd

naive way has an operation that needs locking:

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

but this alternative needs less locking:

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

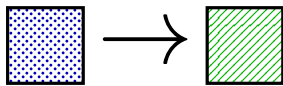
# life homework even/odd

naive way has an operation that needs locking:

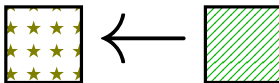
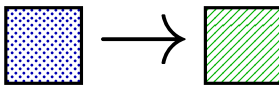
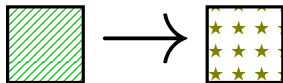
```
for (int time = 0; time < MAX_ITERATIONS; ++time) {  
    ... compute to_grid ...  
    swap(from_grid, to_grid);  
}
```

but this alternative needs less locking:

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



swap



## x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

acquire:

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

release:

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

# x86-64 spinlock with xchg

lock variable in shared memory: the\_lock

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

acquire:

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

test %eax, %eax          // if %eax == 1, then lock is taken
jne acquire              // if not equal, jump to acquire
ret                      // read old value
```

release:

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

# x86-64 spinlock with xchg

lock variable in shared memory: the\_lock

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

acquire:

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

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

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

release:

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



# x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

acquire:

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

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

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

release:

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

# x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

acquire:

```
movl $1, %eax           // %eax <- 1
lock xchg %eax, the_lock // swap %eax and the_lock
                        // sets the_lock to 1 (taken) of t
```

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

Intel's manual says:

no reordering of loads/stores across a `lock` or `mfence` instruction

release:

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

## exercise: spin wait

consider implementing 'waiting' functionality of pthread\_join

thread calls ThreadFinish() when done

complete code below:

finished: .quad 0

ThreadFinish:

-----  
ret

ThreadWaitForFinish:

-----  
lock xchg %eax, finished

cmp \$0, %eax

---- ThreadWaitForFinish

ret

A. mfence; mov \$1, finished

B. mov \$1, finished; mfence

C. mov \$0, %eax

D. mov \$1, %eax

E. je

F. jne

# 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

# 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  
        mark current thread as waiting  
        /* xv6: myproc()->state = SLEEPING; */  
        UnlockSpinlock(&m->guard_spinlock);  
        run scheduler (context switch)  
    } 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  
        mark thread as no longer waiting  
        /* 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  
        mark current thread as waiting  
        /* xv6: myproc()->state = SLEEPING; */  
        UnlockSpinlock(&m->guard_spinlock);  
        run scheduler (context switch)  
    } 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  
        mark thread as no longer waiting  
        /* 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;
};
```

subtly: if UnlockMutex runs here on another core  
need to make sure scheduler on the other core doesn't switch to thread  
while it is still running (would 'clone' thread/mess up registers)

```
LockMutex(Mutex *m) {
    LockSpinlock(&m->guard_spinlock);
    if (m->lock_taken) {
        put current thread on m->wait_queue
        mark current thread as waiting
        /* xv6: myproc()->state = SLEEPING; */
        UnlockSpinlock(&m->guard_spinlock);
        run scheduler (context switch)
    } 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
        mark thread as no longer waiting
        /* 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  
        mark current thread as waiting  
        /* xv6: myproc()->state = SLEEPING; */  
        UnlockSpinlock(&m->guard_spinlock);  
        run scheduler (context switch)  
    } 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  
        mark thread as no longer waiting  
        /* 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)	
start LockMutex		
acquire spinlock		
discover lock taken		
enqueue thread A		
thread A set not runnable		
release spinlock	start UnlockMutex	
	thread A set runnable	
	finish UnlockMutex	
	run scheduler	
	scheduler switches to A	
	...with old version of registers	
thread A runs scheduler		...
...finally saving registers		...

Linux soln.: track 'thread running' separately from 'thread runnable'

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

# mutex and scheduler subtly

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

Linux soln.: track 'thread running' separately from 'thread runnable'

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

# 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

# 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() {	Unlock() {
<b>disable interrupts</b>	<b>enable interrupts</b>
}	}

problem: nested locks

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

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

# C++ containers and locking

can you use a vector from multiple threads?

...question: how is it implemented?

- dynamically allocated array
- reallocated on size changes

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

assuming it's implemented like we expect...

- but can we really depend on that?

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



# C++ standard rules for containers

multiple threads can read anything at the same time

can only read element if no other thread is modifying it

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

(sometimes can safely add/remove in extra cases)

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

might be implemented by putting multiple bools in one int

## a simple race

thread\_A:

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

thread\_B:

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

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

## a simple race

thread\_A:

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

thread\_B:

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

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

if loads/stores atomic, then possible results:

- A:1 B:1 — both moves into x and y, then both moves into eax execute
- A:0 B:1 — thread A executes before thread B
- A:1 B:0 — thread B executes before thread A

# a simple race: results

thread\_A:

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

thread\_B:

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

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

my desktop, 100M trials:

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

# a simple race: results

thread\_A:

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

thread\_B:

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

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

my desktop, 100M trials:

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

# why reorder here?

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
```

thread A: faster to load y right now!

...rather than wait for write of x to finish

# why load/store reordering?

fast processor designs can execute instructions out of order

goal: do something instead of waiting for slow memory accesses, etc.

more on this later in the semester

# 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 forever
```

```
    jne .L3
```

```
    cmpl $0, no_milk
```

```
    ...
```

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

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

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

**solution**

## xv6 spinlock: acquire

```
void
```

```
acquire(struct spinlock *lk)
```

```
{
```

```
    pushcli(); // disable interrupts to avoid deadlock.
```

```
    ...
```

```
    // The xchg is atomic.
```

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

```
        ;
```

```
    // Tell the C compiler and the processor to not move loads or stores
```

```
    // past this point, to ensure that the critical section's memory
```

```
    // references happen after the lock is acquired.
```

```
    __sync_synchronize();
```

```
    ...
```

```
}
```

## xv6 spinlock: acquire

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

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

## xv6 spinlock: acquire

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

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

xchg wraps the lock xchg instruction  
same loop as before

## xv6 spinlock: acquire

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

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

• avoid load store reordering (including by compiler)  
on x86, xchg alone is enough to avoid processor's reordering  
(but compiler may need more hints)

## xv6 spinlock: release

```
void
```

```
release(struct spinlock *lk)
```

```
...
```

```
// Tell the C compiler and the processor to not move loads or stores
```

```
// past this point, to ensure that all the stores in the critical
```

```
// section are visible to other cores before the lock is released
```

```
// Both the C compiler and the hardware may re-order loads and
```

```
// stores; __sync_synchronize() tells them both not to.
```

```
__sync_synchronize();
```

```
// Release the lock, equivalent to lk->locked = 0.
```

```
// This code can't use a C assignment, since it might
```

```
// not be atomic. A real OS would use C atomics here.
```

```
asm volatile("movl $0, %0" : "+m" (lk->locked) : );
```

```
popcli();
```

```
}
```



# xv6 spinlock: release

```
void  
release(struct spinlock *lk)
```

```
...
```

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

```
__sync_synchronize();
```

```
// Release the lock, equivalent to lk->locked = 0.  
// This code can't use a C assignment, since it might  
// not be atomic. A real OS would use C atomics here.
```

```
asm volatile("movl $0, %0" : "+m" (lk->locked) : );
```

```
popcli  
}
```

turns into instruction to tell processor not to reorder  
plus tells compiler not to reorder

# xv6 spinlock: release

```
void
```

```
release(struct spinlock *lk)
```

```
...
```

```
// Tell the C compiler and the processor to not move loads or stores
```

```
// past this point, to ensure that all the stores in the critical
```

```
// section are visible to other cores before the lock is released
```

```
// Both the C compiler and the hardware may re-order loads and
```

```
// stores; __sync_synchronize() tells them both not to.
```

```
__sync_synchronize();
```

```
// Release the lock, equivalent to lk->locked = 0.
```

```
// This code can't use a C assignment, since it might
```

```
// not be atomic. A real OS would use C atomics here.
```

```
asm volatile("movl $0, %0" : "+m" (lk->locked) : );
```

```
popcli();
```

```
}
```

turns into mov of constant 0 into lk->locked

# xv6 spinlock: release

```
void
```

```
release(struct spinlock *lk)
```

```
...
```

```
// Tell the C compiler and the processor to not move loads or stores
```

```
// past this point, to ensure that all the stores in the critical
```

```
// section are visible to other cores before the lock is released
```

```
// Both the C compiler and the hardware may re-order loads and
```

```
// stores; __sync_synchronize() tells them both not to.
```

```
__sync_synchronize();
```

```
// Release the lock, equivalent to lk->locked = 0.
```

```
// This code can't use a C assignment, since it might
```

```
// not be atomic. A real OS would use C atomics here.
```

```
asm volatile("movl $0, %0" : "+m" (lk->locked) : );
```

```
popcli();
```

```
}
```

reenable interrupts (taking nested locks into account)

# fetch-and-add with CAS (1)

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

---

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

implementation sketch:

- fetch value from pointer `old`

- compute in temporary value result of addition `new`

- try to change value at pointer from `old` to `new`

- [compare-and-swap]

- if not successful, repeat

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

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

## exercise: append to singly-linked list

ListNode is a singly-linked list

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

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

- atomically change \*pointer from old to new

- return true if successful

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

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

# some common atomic operations (1)

*// x86: emulate with exchange*

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

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

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

## some common atomic operations (2)

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

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



# common atomic operation pattern

try to do operation, ...

detect if it failed

if so, repeat

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

# cache coherency states

extra information for each cache block

overlaps with/replaces valid, dirty bits

stored in each cache

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

different caches may have different states for same block

# MSI state summary

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

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

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

# MSI scheme

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

blue: transition requires sending message on bus

# MSI scheme

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

blue: transition requires sending message on bus

example: write while Shared

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

# MSI scheme

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

blue: transition requires sending message on bus

example: write while Shared

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

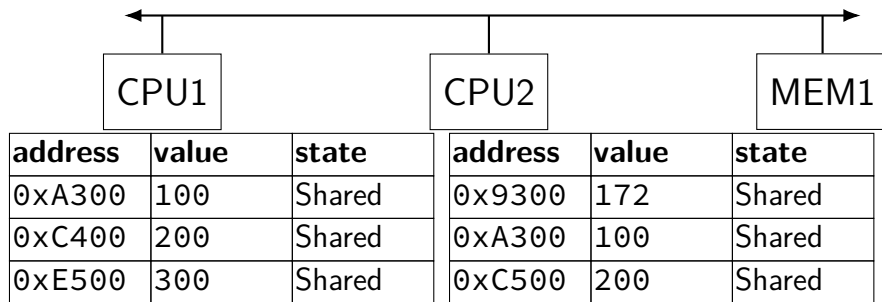
example: hear write while Shared

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

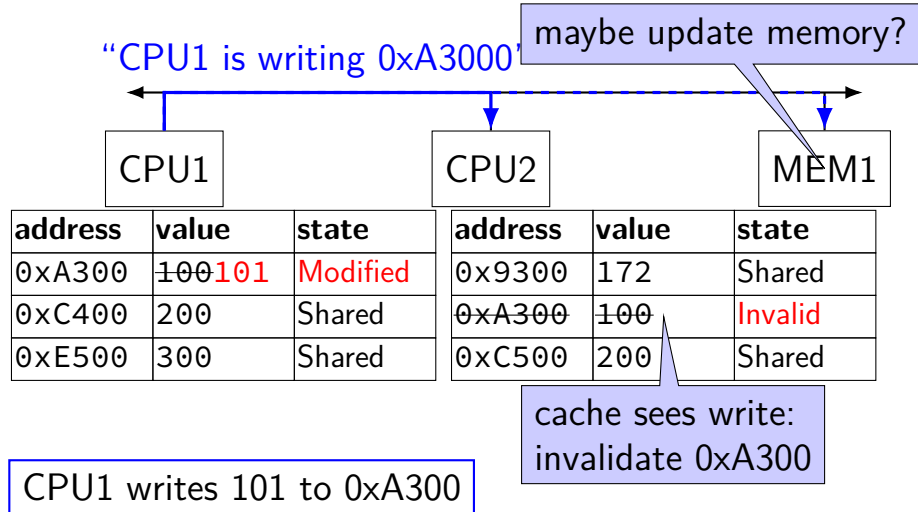
example: write while Modified

nothing to do — no other CPU can have a copy

# MSI example

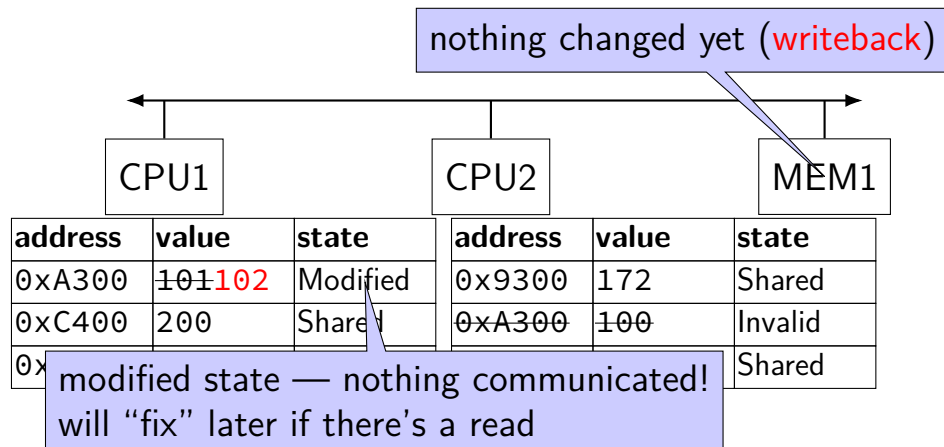


# MSI example

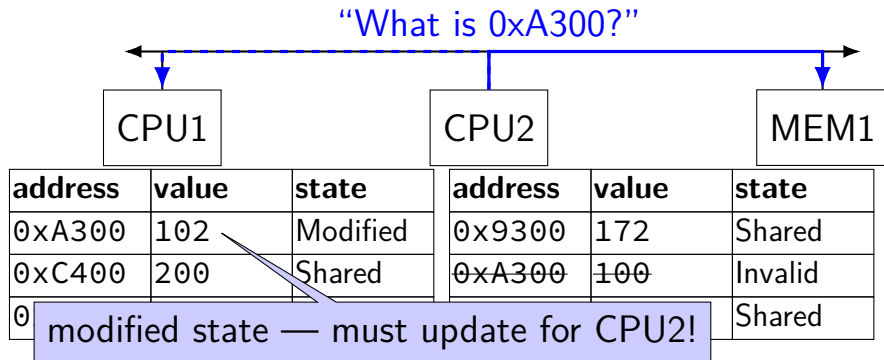




# MSI example



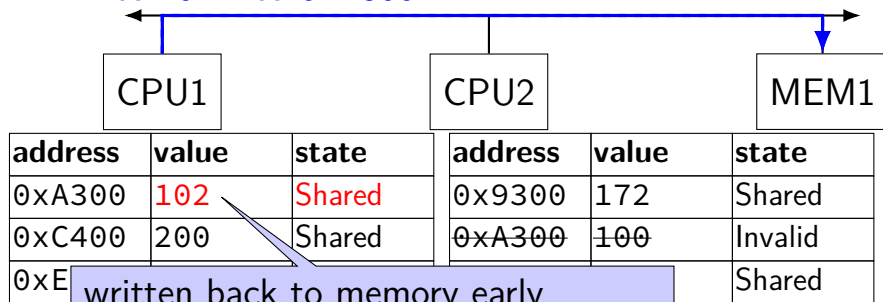
# MSI example



CPU2 reads 0xA300

# MSI example

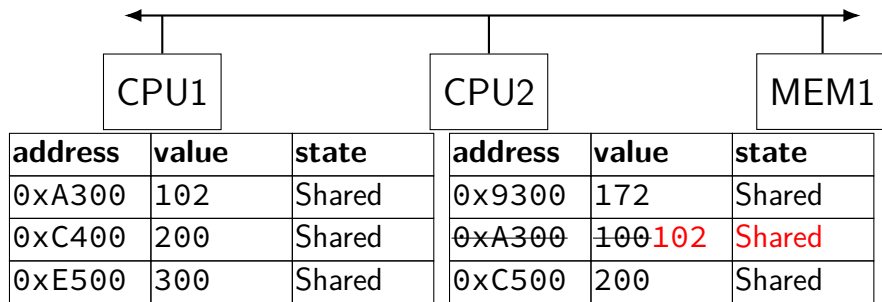
“Write 102 into 0xA300”



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

CPU2 reads 0xA300

# MSI example



# MSI: update memory

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

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

# MSI: on cache replacement/writeback

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

changes state to **invalid**

requires writeback if modified (= dirty bit)

# cache coherency exercise

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

CPU 1: read 0x1000

CPU 2: read 0x1000

CPU 1: write 0x1000

CPU 1: read 0x2000

CPU 2: read 0x1000

CPU 2: write 0x2008

CPU 3: read 0x1008

Q1: final state of 0x1000 in caches?

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

CPU 1:                      CPU 2:                      CPU 3:

Q2: final state of 0x2000 in caches?

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

CPU 1:                      CPU 2:                      CPU 3:

# why load/store reordering?

fast processor designs can execute instructions out of order

goal: do something instead of waiting for slow memory accesses, etc.

more on this later in the semester



# C++: preventing reordering

to help implementing things like `pthread_mutex_lock`

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

prevent CPU reordering *and* prevent compiler reordering

also provide other tools for implementing locks (more later)

could also hand-write assembly code

compiler can't know what assembly code is doing

# C++: preventing reordering example

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

---

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

# C++ atomics: no reordering

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

---

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

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

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

predate `std::atomic`

builtin functions starting with `__sync` and `__atomic`

these are what `xv6` uses

## aside: some x86 reordering rules

each core sees its own loads/stores in order

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

stores *from other cores* appear in a consistent order

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

*causality:*

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

# how do you do anything with this?

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

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

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

# 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

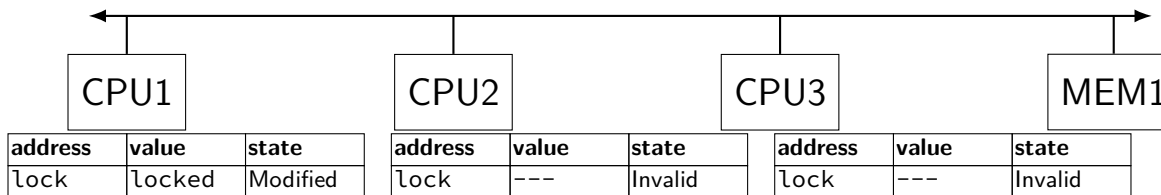
  - want to run another thread instead of infinite loop

  - solution: lock implementation integrated with scheduler

- spinlocks can send a lot of messages on the shared bus

  - more efficient atomic operations to implement locks

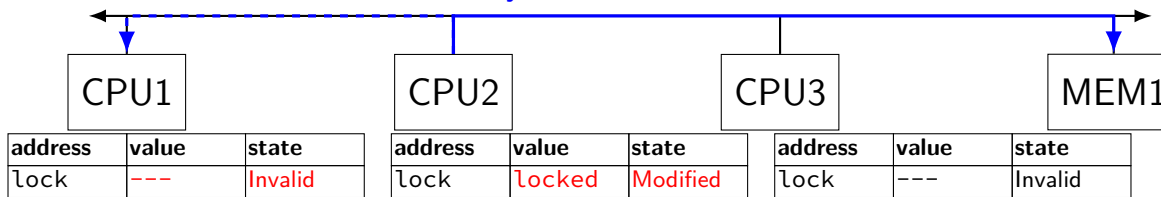
# ping-ponging





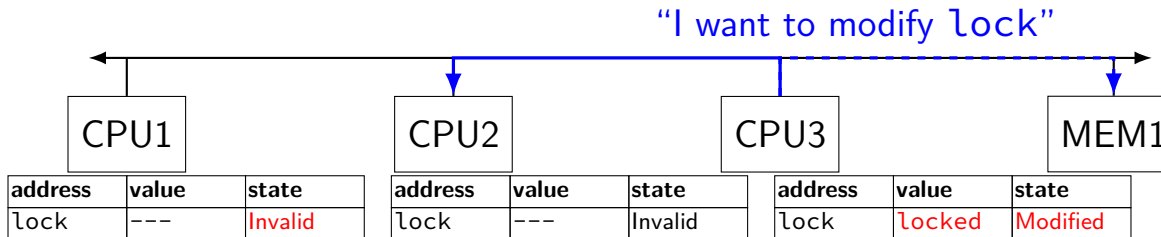
# ping-ponging

"I want to modify lock?"



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

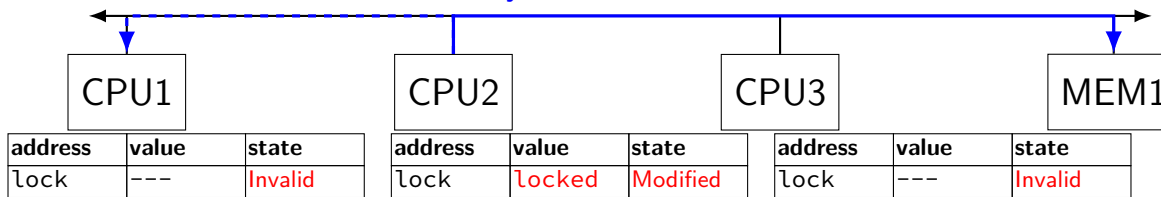
# ping-ponging



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

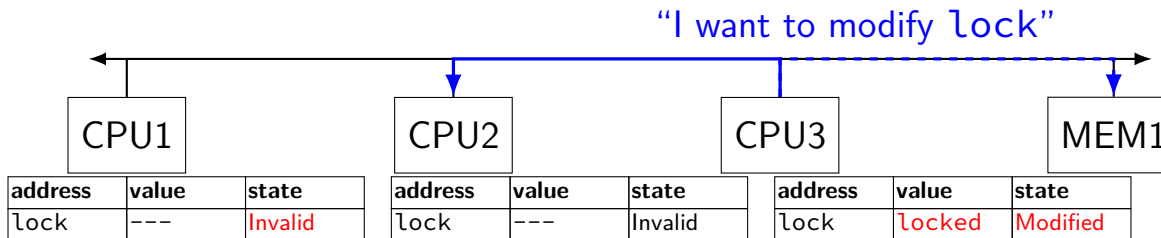
# ping-ponging

"I want to modify lock?"



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

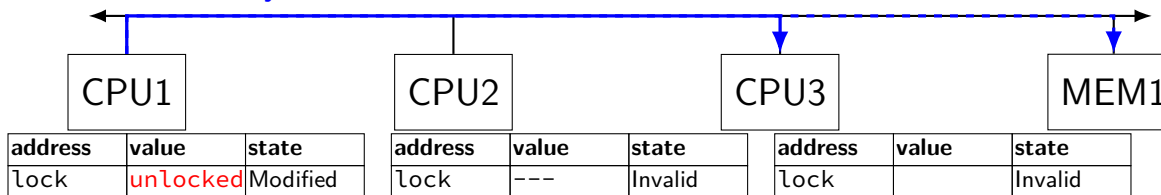
# ping-ponging



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

# ping-ponging

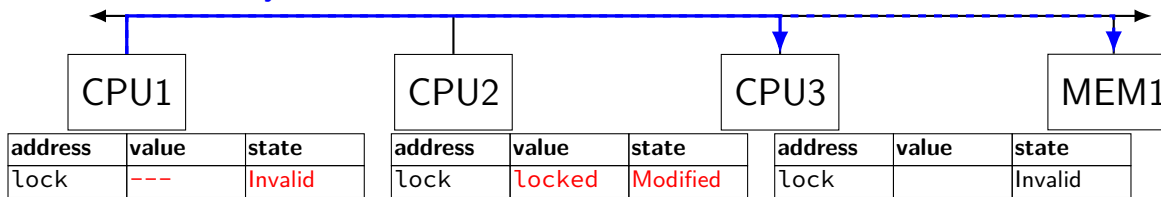
“I want to modify lock”



CPU1 sets lock to unlocked

# ping-ponging

“I want to modify lock”



some CPU (this example: CPU2) acquires lock

# ping-ponging

test-and-set problem: cache block “ping-pongs” between caches  
each waiting processor reserves block to modify  
could maybe wait until it determines modification needed — but not  
typical implementation

each transfer of block sends messages on bus

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

like what the processor with the lock is doing

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

```
acquire(int *the_lock) {  
    do {  
        while (ATOMIC-READ(the_lock) == 0) { /* try again */ }  
    } while (ATOMIC-TEST-AND-SET(the_lock) == ALREADY_SET);  
}
```

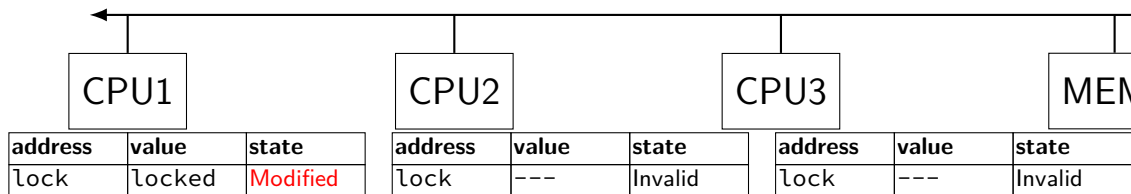


# test-and-test-and-set (assembly)

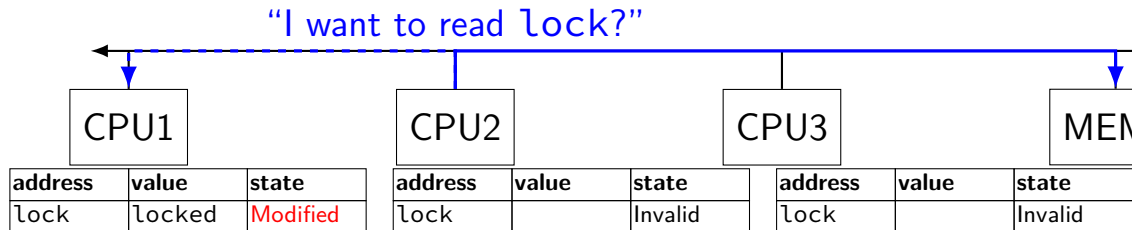
acquire:

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

# less ping-ponging



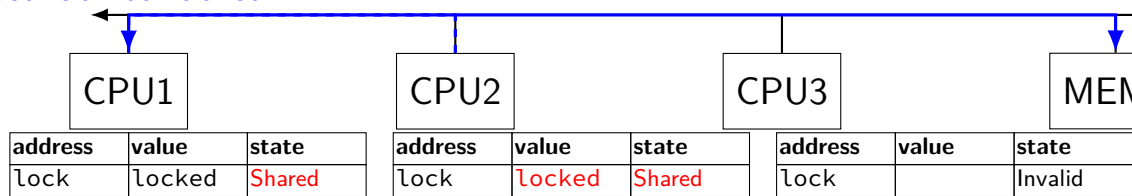
# less ping-ponging



CPU2 reads lock  
(to see it is still locked)

# less ping-ponging

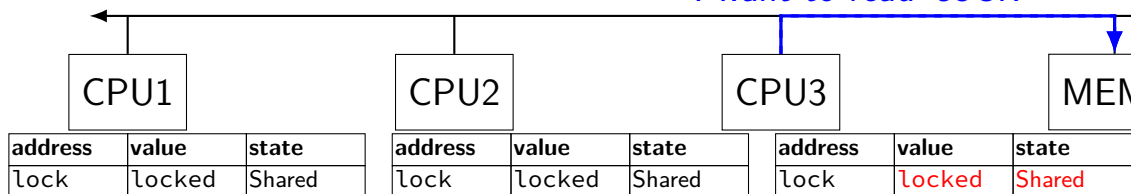
“set lock to locked”



CPU1 writes back lock value,  
then CPU2 reads it

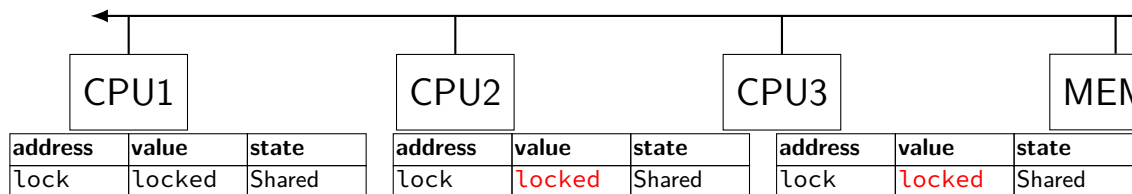
# less ping-ponging

"I want to read lock"



CPU3 reads lock  
(to see it is still locked)

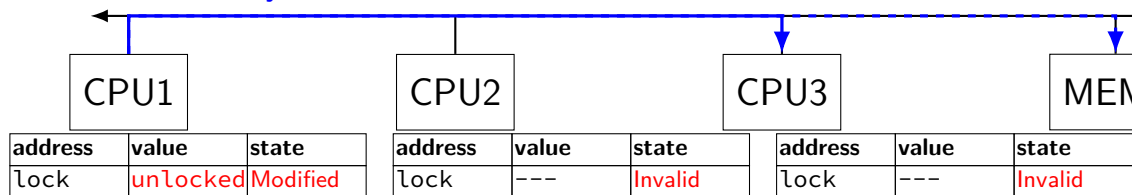
# less ping-ponging



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

# less ping-ponging

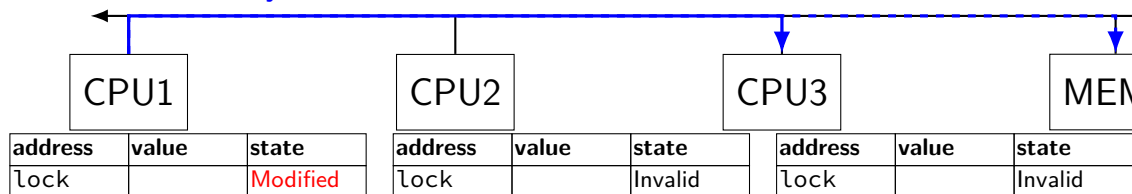
“I want to modify lock”



CPU1 sets lock to unlocked

# less ping-ponging

“I want to modify lock”



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



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

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

and keep it in the shared state

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

# more room for improvement?

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

there other spinlock designs that avoid this

- ticket locks

- MCS locks

- ...

# MSI extensions

real cache coherency protocols sometimes more complex:

separate tracking modifications from whether other caches have copy

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

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

## too much milk

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

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

how can Alice and Bob coordinate better?

# too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

place before buying, remove after buying

don't try buying if there's a note

≈ setting/checking a variable (e.g. “note = 1”)  
with atomic load/store of variable

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

# too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

place before buying, remove after buying

don't try buying if there's a note

≈ setting/checking a variable (e.g. “note = 1”)  
with atomic load/store of variable

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

exercise: why doesn't this work?

# too much milk “solution” 1 (timeline)

**Alice**

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

**Bob**

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

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

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

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



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

### Alice

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

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

**Alice**

```
leave note;
```

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

```
    if (no note) { ← but there's always a note
```

```
        buy milk;
```

```
    }
```

```
}
```

```
remove note;
```

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

**Alice**

```
leave note;
```

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

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

```
        buy milk;
```

```
    }
```

```
}
```

```
remove note;
```

← but there's **always a note**

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

## “solution” 3: algorithm

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

computer equivalent: separate noteFromAlice and noteFromBob

variables

### Alice

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

### Bob

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

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

**Alice**

leave note from Alice

if (no milk) {

    if (no note from Bob) {

~~buy milk~~

    }

}

remove note from Alice

**Bob**

leave note from Bob

if (no milk) {

    if (no note from Alice) {

~~buy milk~~

    }

}

remove note from Bob

## too much milk: is it possible

is there a solutions with writing/reading notes?

≈ loading/storing from shared memory

yes, but it's not very elegant

## too much milk: solution 4 (algorithm)

### Alice

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

### Bob

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

## too much milk: solution 4 (algorithm)

### Alice

leave note from Alice

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

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

```
remove note from Alice
```

exercise (hard): prove (in)correctness

### Bob

leave note from Bob

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

```
remove note from Bob
```



## too much milk: solution 4 (algorithm)

### Alice

leave note from Alice

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

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

```
remove note from Alice
```

exercise (hard): prove (in)correctness

### Bob

leave note from Bob

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

```
remove note from Bob
```

## too much milk: solution 4 (algorithm)

### Alice

leave note from Alice

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

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

```
remove note from Alice
```

### Bob

leave note from Bob

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

```
remove note from Bob
```

exercise (hard): prove (in)correctness

exercise (hard): extend to three people

# Peterson's algorithm

general version of solution

see, e.g., Wikipedia

we'll use special hardware support instead

# mfence

x86 instruction mfence

make sure all loads/stores in progress finish

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

fairly expensive

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

# mfence

x86 instruction mfence

make sure all loads/stores in progress finish

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

fairly expensive

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

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

# modifying cache blocks in parallel

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

typically how caches work — write instructions happen one at a time:

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

processor updates 4 bytes of 64-byte cache block

later, processor might give up cache block

# modifying things in parallel (code)

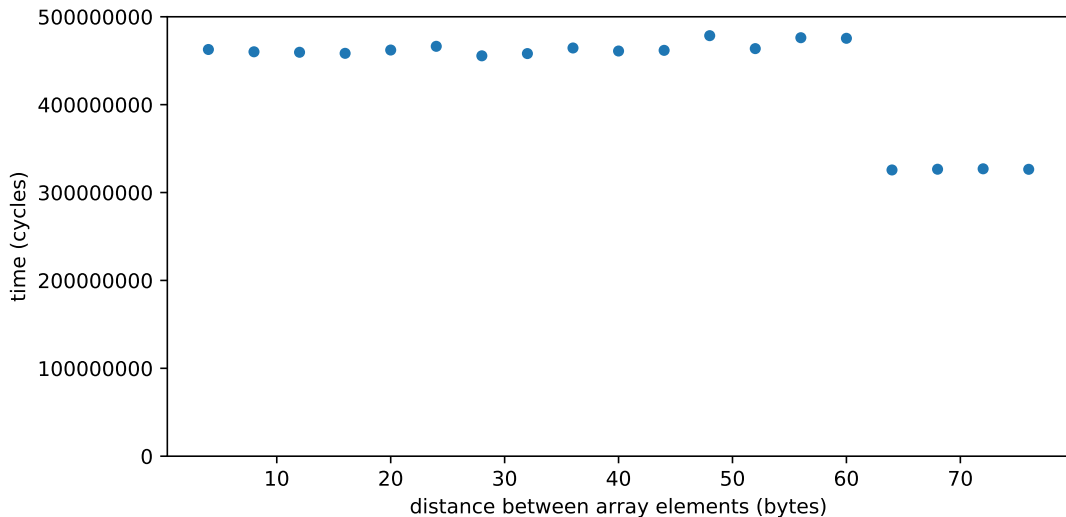
```
void *sum_up(void *raw_dest) {  
    int *dest = (int *) raw_dest;  
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {  
        *dest += data[i];  
    }  
}
```

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

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

# performance v. array element gap

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





# false sharing

synchronizing to access two independent things

two parts of same cache block

solution: separate them

## exercise (1)

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

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

## exercise (2)

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

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

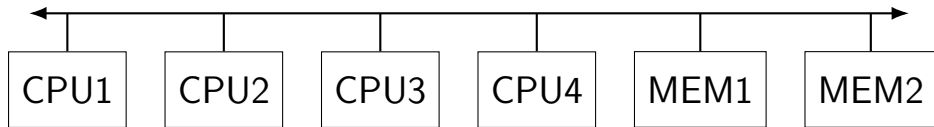
Where is false sharing likely to occur?

# connecting CPUs and memory

multiple processors, common memory

how do processors communicate with memory?

# shared bus



one possible design

we'll revisit later when we talk about I/O

tagged messages — everyone gets everything, filters

contention if multiple communicators

some hardware enforces only one at a time

# shared buses and scaling

shared buses perform poorly with “too many” CPUs

so, there are other designs

we'll gloss over these for now

# shared buses and caches

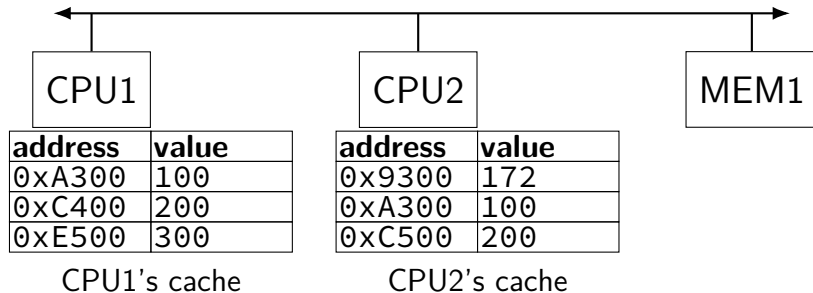
remember caches?

memory is pretty slow

each CPU wants to keep local copies of memory

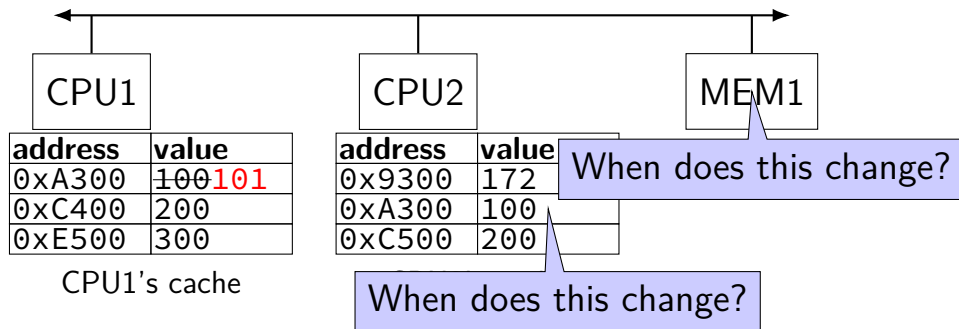
what happens when multiple CPUs cache same memory?

# the cache coherency problem





# the cache coherency problem



CPU1 writes 101 to 0xA300?