synchronization

changelog

24 Oct 2024 (after lecture): add explanation slide after first lock exercise, corrected from version shown in Reiss's lecture

a threading race

```c
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n");
    return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    /* assume does not fail */
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```
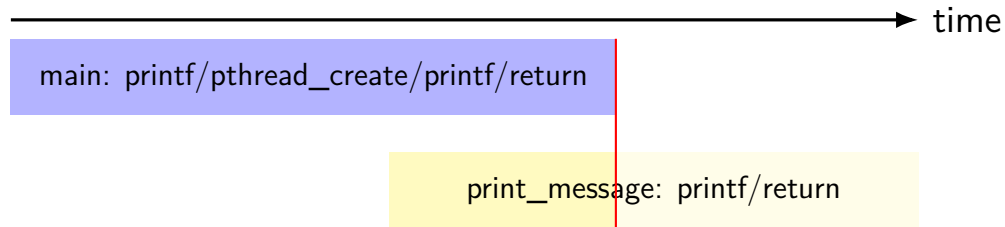
My machine: outputs In the thread about 4% of the time.

a race

returning from main exits the entire process (all its threads)
   same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?

main: printf/pthread_create/printf/return

print_message: printf/return

return from main
ends all threads
in the process

time

the correctness problem

two threads?

introduces *non-determinism*

which one runs first?

allows for "race condition" bugs

...to be avoided with synchronization constructs

example application: ATM server

commands: withdraw, deposit

one correctness goal: don't lose money

ATM server
(pseudocode)

```
ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
Deposit(accountNumber, amount) {
    account = GetAccount(accountNumber);
    account->balance += amount;
    SaveAccountUpdates(account);
}
```

a threaded server?

```
Deposit(accountNumber, amount) {
    account = GetAccount(accountId);
    account->balance += amount;
    SaveAccountUpdates(account);
}
```

maybe GetAccount/SaveAccountUpdates can be slow?
   read/write disk sometimes? contact another server sometimes?

maybe lots of requests to process?
   maybe real logic has more checks than Deposit()
   …

all reasons to handle multiple requests at once

$\rightarrow$ many threads all running the server loop

multiple threads

```
main() {
    for (int i = 0; i < NumberOfThreads; ++i) {
        pthread_create(&server_loop_threads[i], NULL,
                       ServerLoop, NULL);
    }
    ...
}

ServerLoop() {
    while (true) {
        ReceiveRequest(&operation, &accountNumber, &amount);
        if (operation == DEPOSIT) {
            Deposit(accountNumber, amount);
        } else ...
    }
}
```

the lost write

```
account->balance += amount;  (in two threads, same account)
```

Thread A	Thread B
`mov account->balance, %rax`	
`add amount, %rax`	

——————————————— context switch ———————————————

| | `mov account->balance, %rax` |
| | `add amount, %rax` |

——————————————— context switch ———————————————

| `mov %rax, account->balance` | |

——————————————— context switch ———————————————

| | `mov %rax, account->balance` |

the lost write

`account->balance += amount;` (in two threads, same account)

Thread A	Thread B

```
mov account->balance, %rax
add amount, %rax
```
─────────── context switch ───────────
```
                          mov account->balance, %rax
                          add amount, %rax
```
─────────── context switch ───────────
```
mov %rax, account->balance
```
─────────── context switch ───────────
```
                          mov %rax, account->balance
```

lost write to balance

"winner" of the race

the lost write

`account->balance += amount;` (in two threads, same account)

....

| Thread A | Thread B |

```
mov account->balance, %rax
add amount, %rax
```
———————————————— context switch ————————————————
```
                              mov account->balance, %rax
                              add amount, %rax
```
———————————————— context switch ————————————————
```
mov %rax, account->balance
```
———————————————— context switch ————————————————
```
                              mov %rax, account->balance
```

lost write to balance

"winner" of the race

lost track of thread A's more

thinking about race conditions (1)

what are the possible values of x? (initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow 1$	$y \leftarrow 2$

thinking about race conditions (1)

what are the possible values of x? (initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow 1$	$y \leftarrow 2$

must be 1. Thread B can't do anything

thinking about race conditions (2)

possible values of x? (initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

thinking about race conditions (2)

possible values of x? (initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

if A goes first, then B: 1

if B goes first, then A: 5

if B line one, then A, then B line two: 3

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow 1$	$x \leftarrow 2$

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

thinking about race conditions (3)

what are the possible values of x?

(initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow 1$	$x \leftarrow 2$

1 or 2

…but why not 3?

    B: x bit $0 \leftarrow 0$
    A: x bit $0 \leftarrow 1$
    A: x bit $1 \leftarrow 0$
    B: x bit $1 \leftarrow 1$

thinking about race conditions (2)

possible values of x? (initially $x = y = 0$)

Thread A	Thread B
$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

if A goes first, then B: 1

if B goes first, then A: 5

if B line one, then A, then B line two: 3

...and why not 7:

    B (start): $y \leftarrow 2 = 0010_{\text{TWO}}$; then y bit 3 $\leftarrow$ 0; y bit 2 $\leftarrow$ 1; then
    A: x $\leftarrow 110_{\text{TWO}} + 1 = 7$; then
    B (finish): y bit 1 $\leftarrow$ 0; y bit 0 $\leftarrow$ 0

atomic operation

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

we will use these to let threads work together

most machines: loading/storing (aligned) words is atomic
    so can't get 3 from $x \leftarrow 1$ and $x \leftarrow 2$ running in parallel
    aligned $\approx$ address of word is multiple of word size (typically done by compilers)

but some instructions are not atomic; examples:
    x86: integer add constant to memory location
    many CPUs: loading/storing values that cross cache blocks
        e.g. if cache blocks 0x40 bytes, load/store 4 byte from addr. 0x3E is not atomic

lost adds (program)

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

```
int the_value;
extern void *update_loop(void *);
int main(void) {
    the_value = 0;
    pthread_t A, B;
    pthread_create(&A, NULL, update_loop, (void*) 1000000);
    pthread_create(&B, NULL, update_loop, (void*) 1000000);
    pthread_join(A, NULL); pthread_join(B, NULL);
    // expected result: 1000000 + 1000000 = 2000000
    printf("the_value␣=␣%d\n", the_value);
}
```
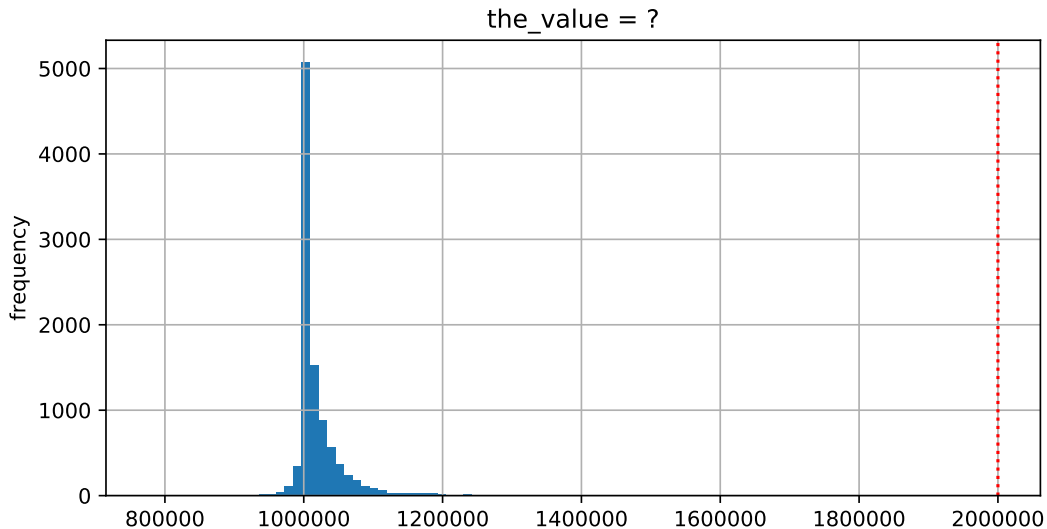
16

lost adds (results)

the_value = ?

but how?

probably not possible on single core
    exceptions can't occur in the middle of add instruction

…but 'add to memory' implemented with multiple steps
    still needs to load, add, store internally
    can be interleaved with what other cores do

but how?

probably not possible on single core
  exceptions can't occur in the middle of add instruction

…but 'add to memory' implemented with multiple steps
  still needs to load, add, store internally
  can be interleaved with what other cores do

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

so, what is actually atomic

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

in general: processor designer will tell you

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

compilers move loads/stores (1)

```
void WaitForReady() {
    do {} while (!ready);
}
```

```
WaitForOther:
  movl ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2              // while (eax == 0) repeat
  ...
```

compilers move loads/stores (1)

```
void WaitForReady() {
    do {} while (!ready);
}
```

```
WaitForOther:
  movl ready, %eax   // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                      // while (eax == 0) repeat
  ...
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                       // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // is_waiting <- 0
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax  // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                       // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // is_waiting <- 0
```

compilers move loads/stores (2)

```
void WaitForOther() {
    is_waiting = 1;
    do {} while (!other_ready);
    is_waiting = 0;
}
```

```
WaitForOther:
  // compiler optimization: don't set is_waiting to 1,
  // (why? it will be set to 0 anyway)
  movl other_ready, %eax   // eax <- other_ready
.L2:
  testl %eax, %eax
  je .L2                       // while (eax == 0) repeat
  ...
  movl $0, is_waiting  // is_waiting <- 0
```

fixing compiler reordering?

isn't there a way to tell compiler not to do these optimizations?

yes, but that is still not enough!

processors sometimes do this kind of reordering too (between cores)

pthreads and reordering

many pthreads functions prevent reordering
  everything before function call actually happens before

includes preventing some optimizations
  e.g. keeping global variable in register for too long

pthread_create, pthread_join, other tools we'll talk about …
  basically: if pthreads is waiting for/starting something, no weird ordering

implementation part 1: prevent compiler reordering

implementation part 2: use special instructions
  example: x86 `mfence` instruction

some definitions

mutual exclusion: ensuring only one thread does a particular thing at a time
 like updating shared balance

some definitions

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

> like updating shared balance

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

> result of mutual exclusion

some definitions

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

  like updating shared balance

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

  result of mutual exclusion

lock: object only one thread can hold at a time

  interface for creating critical sections

lock analogy

agreement: only change account balances while wearing this hat

normally hat kept on table

put on hat when editing balance

hopefully, only one person ($=$ thread) can wear hat a time

need to wait for them to remove hat to put it on

lock analogy

agreement: only change account balances while wearing this hat

normally hat kept on table

put on hat when editing balance

hopefully, only one person ($=$ thread) can wear hat a time

need to wait for them to remove hat to put it on

"lock (or acquire) the lock" $=$ get and put on hat

"unlock (or release) the lock" $=$ put hat back on table

the lock primitive

locks: an object with (at least) two operations:
    acquire or *lock* — wait until lock is free, then "grab" it
    release or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource
    forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

the lock primitive

locks: an object with (at least) two operations:
    acquire or *lock* — wait until lock is free, then "grab" it
    release or *unlock* — let others use lock, wakeup waiters

typical usage: everyone acquires lock before using shared resource
    forget to acquire lock? weird things happen

```
Lock(account_lock);
balance += ...;
Unlock(account_lock);
```

waiting for lock?

when waiting — ideally:

not using processor (at least if waiting a while)

OS can context switch to other programs

pthread mutex

```c
#include <pthread.h>

pthread_mutex_t account_lock;
pthread_mutex_init(&account_lock, NULL);
    // or: pthread_mutex_t account_lock =
    //                 PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&account_lock);
balance += ...;
pthread_mutex_unlock(&account_lock);
```

exercise

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

solution

B1+A2

    A: L(1) A1 U(1) L
    B: L(1) B1 L(2) B2 U(2) U(1)
    A: L(2) A2 U(2)

NOT A1+B2

    would need to run B1 before A1 before A2 before B2
    not possible because Lock1 held for entire B1+B2 operation
    so cannot fit A1+A2 in between B1 and B2

exercise (alternate 1)

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

exercise (alternate 2)

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

POSIX mutex restrictions

pthread_mutex rule: unlock from same thread you lock in

does this actually matter?

depends on how pthread_mutex is implemented

preview: general sync

lots of coordinating threads beyond locks

will talk about two general tools later:k
    monitors/condition variables
    semaphores [if time]

big added feature: wait for arbitrary thing to happen

also some less general tools: barriers

a bad idea

one bad idea to wait for an event:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; bool ready = false;
void WaitForReady() {
    pthread_mutex_lock(&lock);
    do {
        pthread_mutex_unlock(&lock);
        /* only time MarkReady() can run */
        pthread_mutex_lock(&lock);
    } while (!ready);
    pthread_mutex_unlock(&lock);
}
void MarkReady() {
    pthread_mutex_lock(&lock);
    ready = true;
    pthread_mutex_unlock(&lock);
}
```

wastes processor time; MarkReady can stall waiting for unlock

beyond locks

in practice: want more than locks for synchronization

for waiting for arbtirary events (without CPU-hogging-loop):
    monitors
    semaphores

for common synchornization patterns:
    barriers
    reader-writer locks

higher-level interface:
    transactions

barriers

compute minimum of 100M element array with 2 processors

algorithm:

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

wait for all computations to finish

take minimum of all the minimums

barriers

compute minimum of 100M element array with 2 processors

algorithm:

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

wait for all computations to finish

take minimum of all the minimums

barriers API

barrier.Initialize(NumberOfThreads)

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

idea: multiple threads perform computations in parallel

threads wait for all other threads to call Wait()

barrier: waiting for finish

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

Thread 0	Thread 1

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

barrier.Wait();


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

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

barriers: reuse

	Thread 0		Thread 1

Thread 0

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

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

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

Thread 1

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

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

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

barriers: reuse

Thread 0

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

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

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

Thread 1

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

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

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

barriers: reuse

Thread 0

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

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

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

Thread 1

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

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

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

pthread barriers

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

exercise

```
pthread_barrier_t barrier; int x = 0, y = 0;
void thread_one() {
    y = 10;
    pthread_barrier_wait(&barrier);
    y = x + y;
    pthread_barrier_wait(&barrier);
    pthread_barrier_wait(&barrier);
    printf("%d %d\n", x, y);
}
void thread_two() {
    x = 20;
    pthread_barrier_wait(&barrier);
    pthread_barrier_wait(&barrier);
    x = x + y;
    pthread_barrier_wait(&barrier);
}
```

output? (if both run at once, barrier set for 2 threads)

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);
}
```
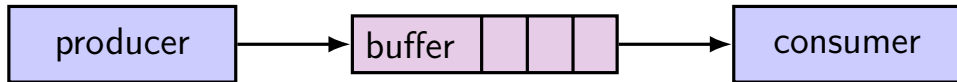
life homework

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 $t+1$
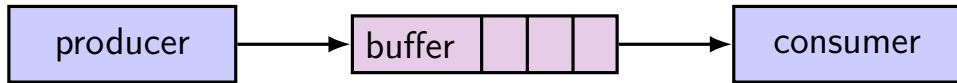
example: producer/consumer

shared buffer (queue) of fixed size
  one or more producers inserts into queue
  one or more consumers removes from queue

example: producer/consumer
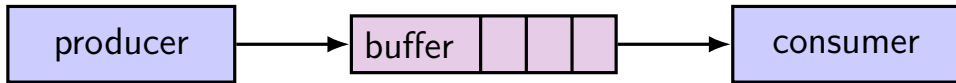

shared buffer (queue) of fixed size
    one or more producers inserts into queue
    one or more consumers removes from queue

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

monitor idea

a monitor

lock	----▶ ☐ ──▶ ☐ ──▶ ☐ threads waiting for lock
shared data	
condvar 1	
condvar 2	
…	
operation1(…)	
operation2(…)	

monitor idea

a monitor

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

threads waiting for lock

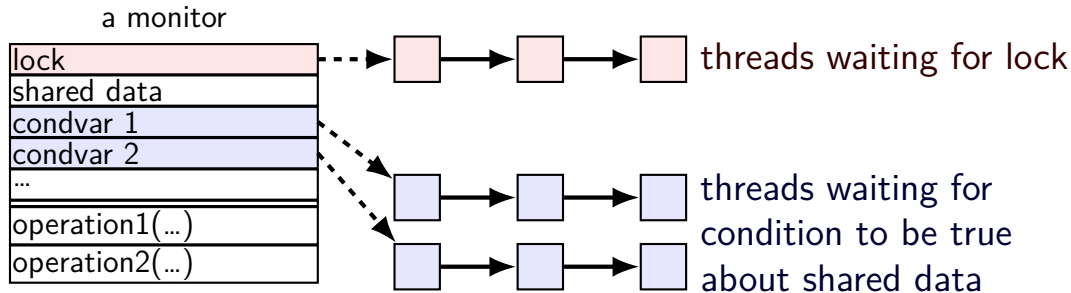threads waiting for condition to be true about shared data

condvar operations
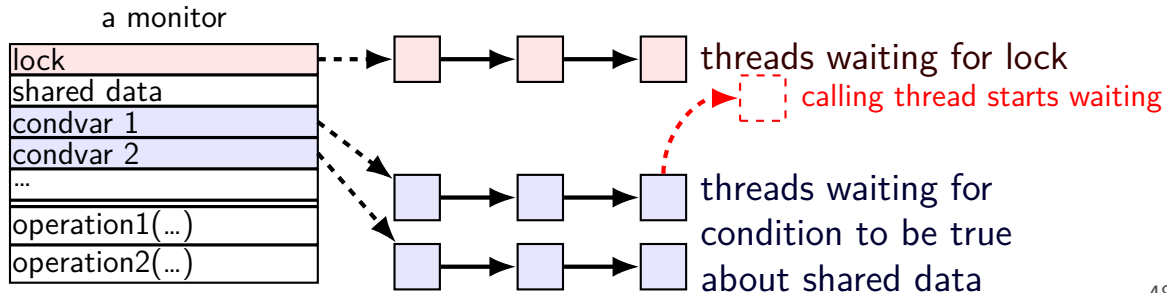
condvar operations:

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

a monitor

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

threads waiting for lock

threads waiting for
condition to be true
about shared data

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
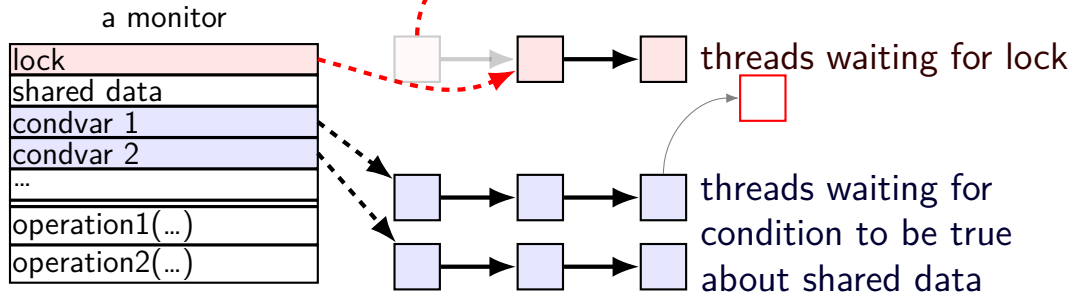
condvar operations

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

unlock lock — allow thread from queue to go

a monitor

threads waiting for lock

threads waiting for
condition to be true
about shared data

lock
shared data
condvar 1
condvar 2
…
operation1(…)
operation2(…)
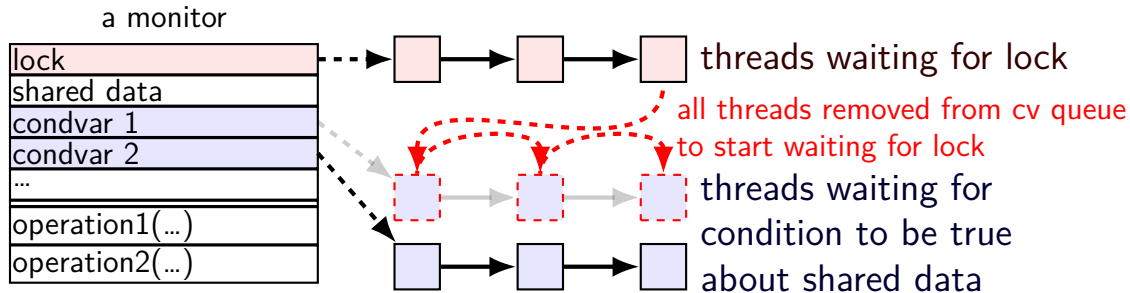
48

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

a monitor

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

threads waiting for lock

all threads removed from cv queue
to start waiting for lock

threads waiting for
condition to be true
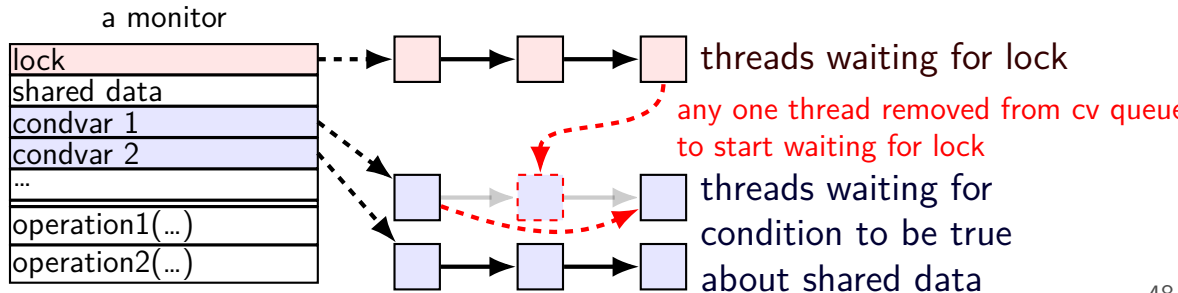about shared data

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

a monitor

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

threads waiting for lock

any one thread removed from cv queue
to start waiting for lock

threads waiting for
condition to be true
about shared data

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)	

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

53

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

53

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 …?

53

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

Thread 1	Thread 2
	Consume()
	…lock
	…empty? yes
	…unlock/start wait
Produce()	waiting for
…lock	data_ready
…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 …?

53

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	lock
…unlock	waiting for 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 …?

53

unbounded buffer producer/consumer

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

in pthreads: signalled thread not
    gaurenteed to hold lock next);

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

            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	lock
...unlock	waiting for 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 ...?

53

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;
}
```
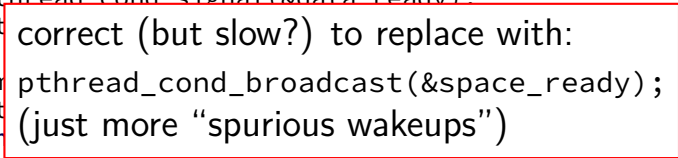
bounded buffer producer/consumer

```
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);
    pt
}
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 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_fini

void WaitForBothFinished
  pthread_mutex_lock(&lo
  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:

```c
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);
}
```

monitor exercise: barrier

```
struct BarrierInfo {
    pthread_mutex_t lock;
    int total_threads;  // initially total # of threads
    int number_reached; // initially 0
    pthread_cond_t cv;
};

void BarrierWait(BarrierInfo *b) {
    pthread_mutex_lock(&b->lock);
    ++b->number_reached;
    if (b->number_reached == b->total_threads) {
        pthread_cond_broadcast(&b->cv);
    } else {
        while (b->number_reached < b->total_threads)
            pthread_cond_wait(&b->cv, &b->lock);
    }
    pthread_mutex_unlock(&b->lock);
}
```

generalizing locks: semaphores

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

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

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

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

semaphores are kinda integers

semaphore like an integer, but...

cannot read/write directly
     down/up operaion only way to access (typically)
     exception: initialization

never negative — wait instead
     down operation wants to make negative? thread waits

reserving books

suppose tracking copies of library book…

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

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

counting resources: reserving books

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

Copy 1
Copy 2
Copy 3

free copies 3

counting resources: reserving books

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

taken out

| Copy 1 |
| Copy 2 |
| Copy 3 |

free copies 3 2

after calling down to reserve

counting resources: reserving books

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

taken out

| Copy 1 |
| Copy 2 |
| Copy 3 |

free copies $\boxed{2}$

after calling down to reserve

counting resources: reserving books

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

taken out | Copy 1
taken out | Copy 2
taken out | Copy 3
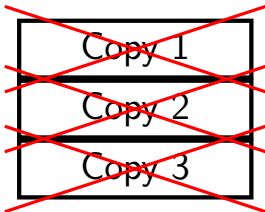
free copies | 0 |

after calling down three times
    to reserve all copies

counting resources: reserving books

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

taken out | Copy 1
taken out | Copy 2
taken out | Copy 3

free copies | 0
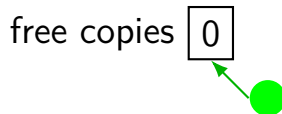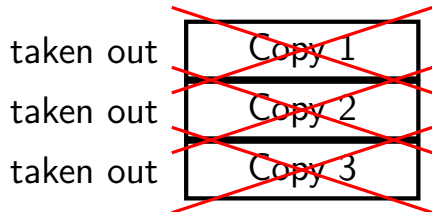
reserve book
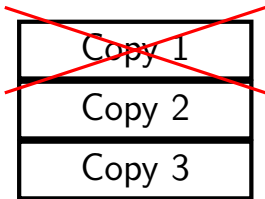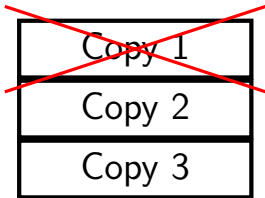call *down* again
start waiting...

counting resources: reserving books

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

taken out | Copy 1
taken out | Copy 2
taken out | Copy 3

free copies | 0

return book

call *up*
release waiter

reserve book

call *down*
~~waiting~~
done waiting
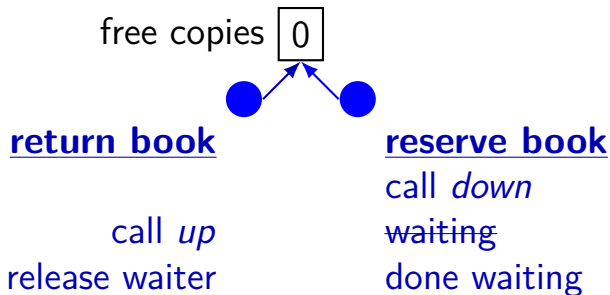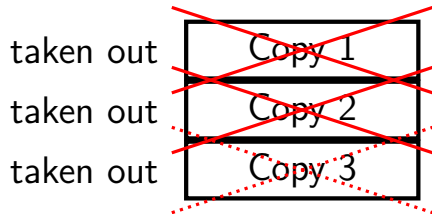
implementing mutexes with semaphores

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

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

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

implementing join with semaphores

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

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

POSIX semaphores

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

semaphore exercise

```
int value;  sem_t empty, ready;  // with some initial values

void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}

int GetValue() {
    int result;

    _____
    result = value;

    _____
    return result;
}
```

What goes in the blanks?
A: `sem_post(&empty)` / `sem_wait(&ready)`
B: `sem_wait(&ready)` / `sem_post(&empty)`
C: `sem_post(&ready)` / `sem_wait(&empty)`
D: `sem_post(&ready)` / `sem_post(&empty)`
E: `sem_wait(&empty)` / `sem_post(&ready)`
F: something else

GetValue() waits for PutValue() to happen, retrieves value, then allows next PutValue().

semaphore exercise [solution]

```
int value;
sem_t empty, ready;
void PutValue(int argument) {
    sem_wait(&empty);
    value = argument;
    sem_post(&ready);
}
int GetValue() {
    int result;
    sem_wait(&ready);
    result = value;
    sem_post(&empty);
    return result;
}
```

semaphore intuition

What do you need to wait for?
> critical section to be finished
> queue to be non-empty
> array to have space for new items

what can you count that will be 0 when you need to wait?
> # of threads that can start critical section now
> # of threads that can join another thread without waiting
> # of items in queue
> # of empty spaces in array

use up/down operations to maintain count

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

one semaphore per constraint:

```
sem_t full_slots;    // consumer waits if empty
sem_t empty_slots;   // producer waits if full
sem_t mutex;         // either waits if anyone changing buffer
FixedSizedQueue buffer;
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots); // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots); // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots); // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots); // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots);    // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                           data
}

Consume() {
    sem_wait(&full_slots);  // wait until queued item, reserve it
    sem_wait(&mutex);
    item = buffer.dequeue();
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
```

Can we do
  sem_wait(&mutex);
  sem_wait(&empty_slots);
instead?

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots);   // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                              data
}

Consume() {
    sem_wait(&full_slots);
    sem_wait(&mutex);
    item = buffer.dequeue()
    sem_post(&mutex);
    sem_post(&empty_slots);
    return item;
}
```

Can we do
    sem_wait(&mutex);
    sem_wait(&empty_slots);
instead?

No. Consumer waits on sem_wait(&mutex)
so can't sem_post(&empty_slots)
(result: producer waits forever
problem called *deadlock*)

73

producer/consumer: cannot reorder mutex/empty

```
ProducerReordered() {
  // BROKEN: WRONG ORDER
  sem_wait(&mutex);
  sem_wait(&empty_slots);

  ...

  sem_post(&mutex);
}
```

```
Consumer() {
  sem_wait(&full_slots);

  // can't finish until
  // Producer's sem_post(&mutex):
  sem_wait(&mutex);

  ...

  // so this is not reached
  sem_post(&full_slots);
}
```

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...
Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots                             more data
}

Consume() {
    sem_wait(&full_slots                             reserve it
    sem_wait(&mutex);
    item = buffer.dequeu
    sem_post(&mutex);
    sem_post(&empty_slots);  // let producer reuse item slot
    return item;
}
```

Can we do
    sem_post(&full_slots);
    sem_post(&mutex);
instead?
Yes — post never waits

producer/consumer summary

producer: wait (down) empty_slots, post (up) full_slots

consumer: wait (down) full_slots, post (up) empty_slots

two producers or consumers?
    still works!

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

backup slides

atomic read-modfiy-write

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

…so processors provide read/modify/write operations

one instruction that
atomically
reads *and* modifies *and* writes back a value

used by OS to implement higher-level synchronization tools

x86 atomic exchange

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

atomic exchange

temp ← M[ECX]

M[ECX] ← EAX

EAX ← temp

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

implementing atomic exchange

make sure other processors don't have cache block

    probably need to be able to do this to keep caches in sync

do read+modify+write operation

higher level tools

usually we won't use atomic operations directly

instead rely on OS/standard libraries using them

(along with context switching, disabling interrupts, ...)

OS/standard libraries will provide higher-level tools like...

pthread_join

locks (pthread_mutex)

...and more

backup slides

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 ...
}
```
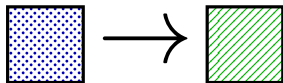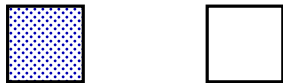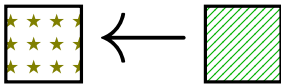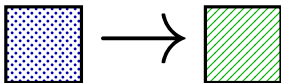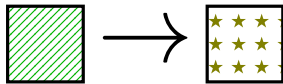
x86-64 spinlock with xchg

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 t
    test %eax, %eax          // if the_lock wasn't 0 before:
    jne acquire              //    try again
    ret

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

    test %eax, %eax           // if set lock variable to 1 (taken)
    jne acquire               //     read old value
    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 original value of t

    test %eax, %eax
    jne acquire
    ret

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

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

x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

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

    test %eax, %eax      release lock by setting it to 0 (not taken)  of t
    jne acquire          allows looping acquire to finish
    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)
                                                        of t

    test %eax, %eax
    jne acquire
    ret

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

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

exercise: spin wait

consider implementing 'waiting' functionality of pthread_join

thread calls ThreadFinish() when done

complete code below:

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

A mfence; mov $1, finished C mov $0, %eax E je

exercise: spin wait

```
finished: .quad 0
ThreadFinish:
    _____A_____
    ret
ThreadWaitForFinish:                /* or without using a writing instr
    _____B_____        mov %eax, finished
    lock xchg %eax, finished        mfence
    cmp $0, %eax                     cmp $0, %eax
    __C_ ThreadWaitForFinish        je ThreadWaitForFinish
    ret                             ret
```

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

```
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;
```

```
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;
```

```
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;
```

```
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 verison of registers	
thread A runs scheduler		…
…finally saving registers		…

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

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 verison of registers	
thread A runs scheduler		...
...finally saving registers		...

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

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

naive interrupt enable/disable (2)

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

naive interrupt enable/disable (2)

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

naive interrupt enable/disable (2)

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

problem: nested locks

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

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


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

a simple race

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


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

if loads/stores atomic, then possible results:

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

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

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

a simple race: results

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

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

my desktop, 100M trials:

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

a simple race: results

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

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

my desktop, 100M trials:

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

why reorder here?

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

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)

```c
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 f
  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

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

xv6 spinlock: acquire

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

  // Tell the C compiler and the processor to not move loads or sto
  // 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)
    ;
```

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 sto
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchr| 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 sto
  // past this point, to ensure that the critical section's memory
  //  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 sto
  // 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 sto
  // 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
  asm vo    turns into instruction to tell processor not to reorder
            plus tells compiler not to reorder

  popcli();
}
```

xv6 spinlock: release

```
void
release(struct spinlock *lk)
  ...
  // Tell the C compiler and the processor to not move loads or sto
  // 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 sto
  // 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) {
    ...
}
```

append to singly-linked list

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

some common atomic operations (1)

```
// x86: emulate with exchange
test_and_set(address) {
    old_value = memory[address];
    memory[address] = 1;
    return old_value != 0;  // e.g. set ZF flag
}

// x86: xchg REGISTER, (ADDRESS)
exchange(register, address) {
    temp = memory[address];
    memory[address] = register;
    register = temp;
}
```

some common atomic operations (2)

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

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

common atomic operation pattern

try to do operation, …

detect if it failed

if so, repeat

atomic operation does "try and see if it failed" part

cache coherency states

extra information for each cache block
    overlaps with/replaces valid, dirty bits

stored in each cache

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

different caches may have different states for same block

MSI state summary

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

Shared   value is the same as memory

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

MSI scheme

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

blue: transition requires sending message on bus

MSI scheme

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

blue: transition requires sending message on bus

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

MSI scheme

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

blue: transition requires sending message on bus

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

example: hear write while Shared
    change to Invalid
    can send read later to get value from writer

example: write while Modified
    nothing to do — no other CPU can have a copy

MSI example

address	value	state	address	value	state
0xA300	100	Shared	0x9300	172	Shared
0xC400	200	Shared	0xA300	100	Shared
0xE500	300	Shared	0xC500	200	Shared

MSI example

"CPU1 is writing 0xA3000"

maybe update memory?

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

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

cache sees write:
invalidate 0xA300

CPU1 writes 101 to 0xA300

129

MSI example

nothing changed yet (writeback)

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

CPU1   CPU2   MEM1

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

modified state — nothing communicated!
will "fix" later if there's a read

CPU1 writes 102 to 0xA300

129

MSI example

"What is 0xA300?"

CPU1			CPU2		
address	**value**	**state**	**address**	**value**	**state**
0xA300	102	Modified	0x9300	172	Shared
0xC400	200	Shared	~~0xA300~~	~~100~~	Invalid
0					Shared

MEM1

modified state — must update for CPU2!
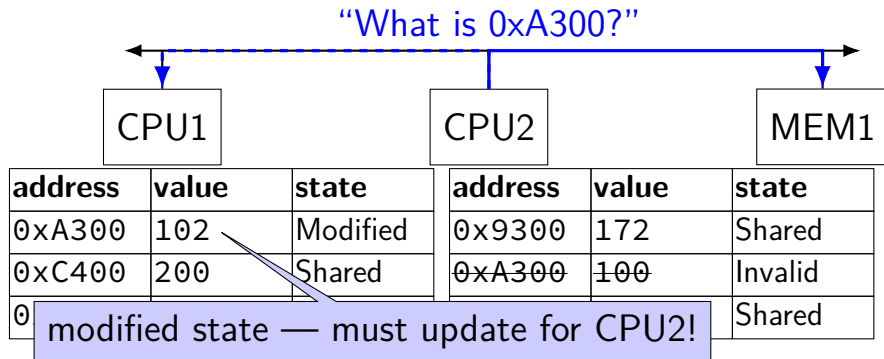
CPU2 reads 0xA300

MSI example

"Write 102 into 0xA300"

CPU1			CPU2			MEM1

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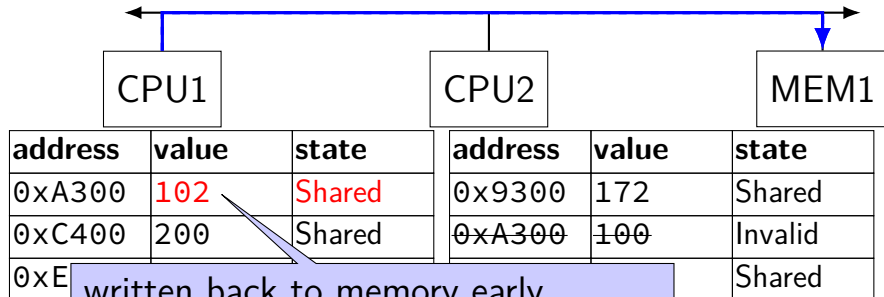
written back to memory early
(could also become Invalid at CPU1)

CPU2 reads 0xA300

MSI example

address	value	state		address	value	state
0xA300	102	Shared		0x9300	172	Shared
0xC400	200	Shared		~~0xA300~~	~~100~~102	Shared
0xE500	300	Shared		0xC500	200	Shared

CPU1   CPU2   MEM1
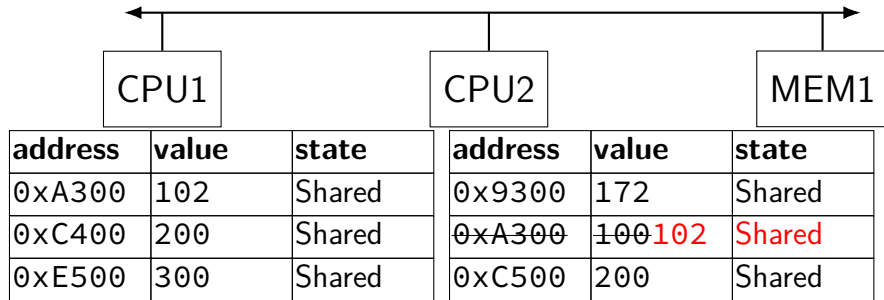
MSI: update memory

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

cache coherency exercise solution

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

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

```cpp
#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

```cpp
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

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

CPU1 CPU2 CPU3 MEM1

ping-ponging

"I want to modify `lock`?"

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

CPU1

address	value	state
`lock`	`locked`	Modified

CPU2

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

CPU3

MEM1

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

ping-ponging

"I want to modify `lock`"

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

CPU1   CPU2   CPU3   MEM1

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

ping-ponging

"I want to modify `lock`?"

address	value	state
CPU1		
lock	---	Invalid

address	value	state
CPU2		
lock	locked	Modified

address	value	state
CPU3		
lock	---	Invalid

MEM1

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

ping-ponging

"I want to modify `lock`"

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

CPU1   CPU2   CPU3   MEM1

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
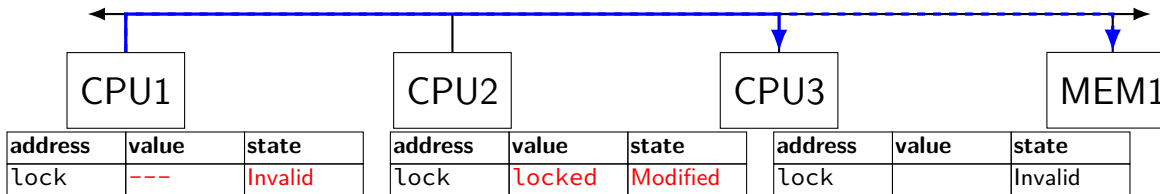
some CPU (this example: CPU2) acquires lock

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

"I want to read lock?"

CPU1		
address	**value**	**state**
lock	locked	Modified

CPU2		
address	**value**	**state**
lock		Invalid

CPU3		
address	**value**	**state**
lock		Invalid

MEM
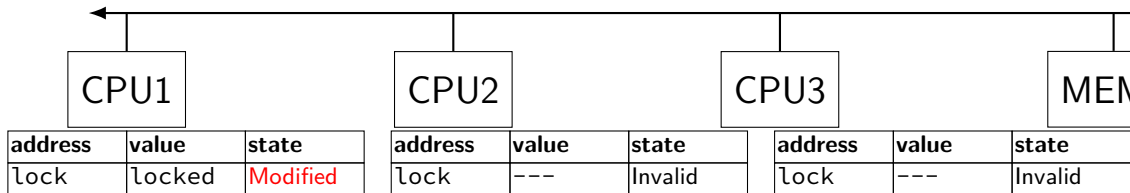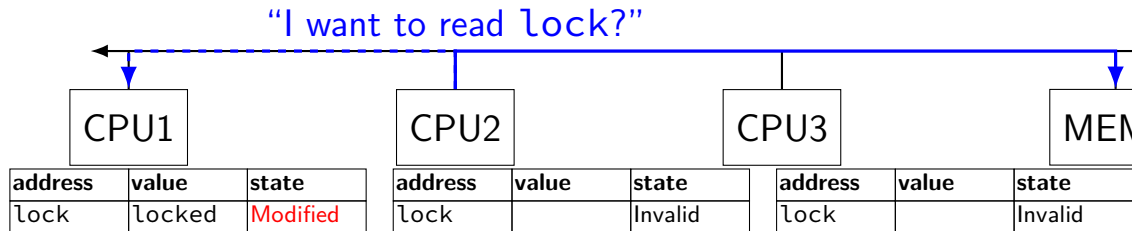
CPU2 reads lock
(to see it is still locked)

less ping-ponging

"set lock to locked"

address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock		Invalid

CPU1 CPU2 CPU3 MEM
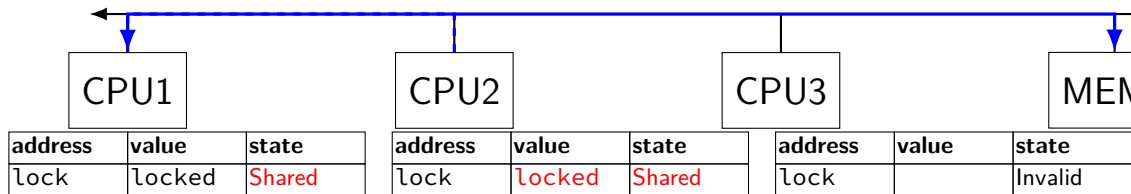
CPU1 writes back lock value,
then CPU2 reads it

less ping-ponging

"I want to read `lock`"

CPU1		
address	**value**	**state**
lock	locked	Shared

CPU2		
address	**value**	**state**
lock	locked	Shared

CPU3		
address	**value**	**state**
lock	locked	Shared

MEM

CPU3 reads lock
(to see it is still locked)

less ping-ponging

address	value	state	address	value	state	address	value	state
lock	locked	Shared	lock	locked	Shared	lock	locked	Shared
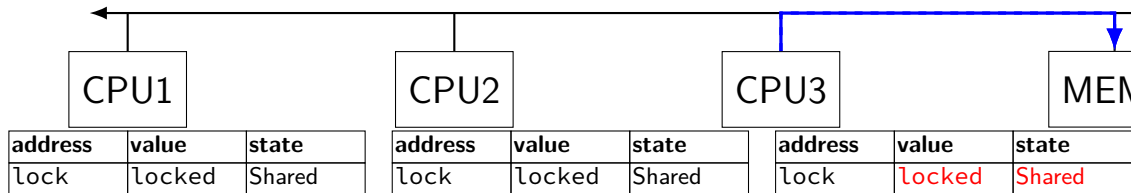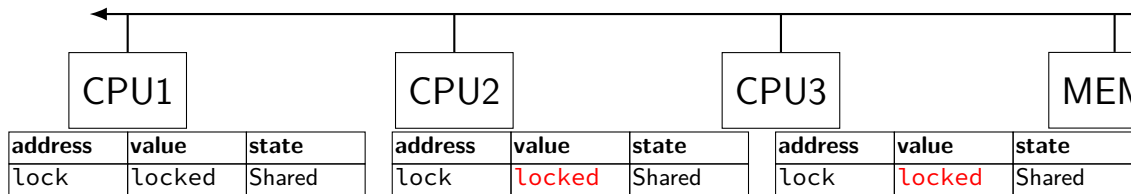
CPU2, CPU3 continue to read lock from cache
no messages on the bus

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 — CPU2 — CPU3 — MEM
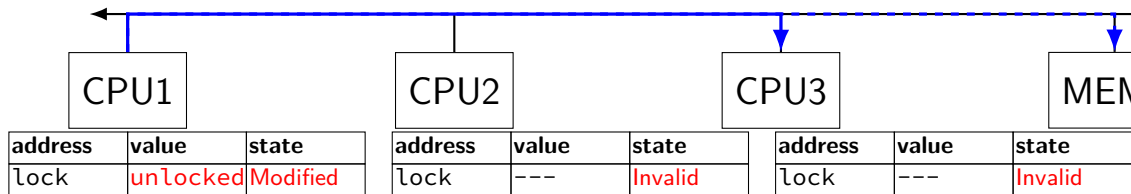
CPU1 sets lock to unlocked

less ping-ponging

"I want to modify `lock`"

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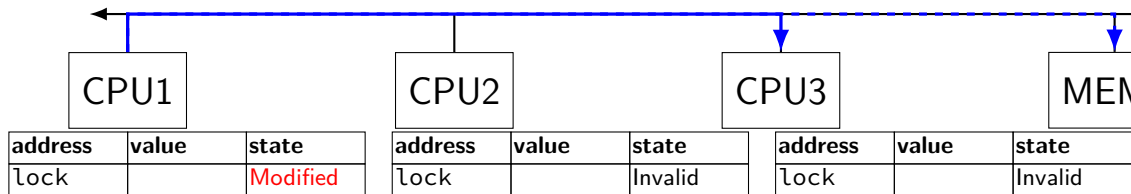
some CPU (this example: CPU2) acquires lock
(CPU1 writes back value, then CPU2 reads + modifies it)

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

    …

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

$\approx$ 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

$\approx$ 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	Bob

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

                                if (no milk) {
                                    if (no note) {

        leave note;
        buy milk;
        remove 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) { ←——— but there's always a note
        buy milk;           …will never buy milk (twice or once)
    }
}
remove note;
```

"solution" 3: algorithm

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

    computer equivalent: separate noteFromAlice and noteFromBob
    variables

Alice	**Bob**

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

too much milk: "solution" 3 (timeline)

Alice	Bob

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

                                    leave note from Bob

    if (no note from Bob) {
        buy milk
    }
}


                                    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?
    $\approx$ 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	**Bob**

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

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

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice	Bob

Alice

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

Bob

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

exercise (hard): prove (in)correctness

too much milk: solution 4 (algorithm)

Alice	Bob

```
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 is 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)
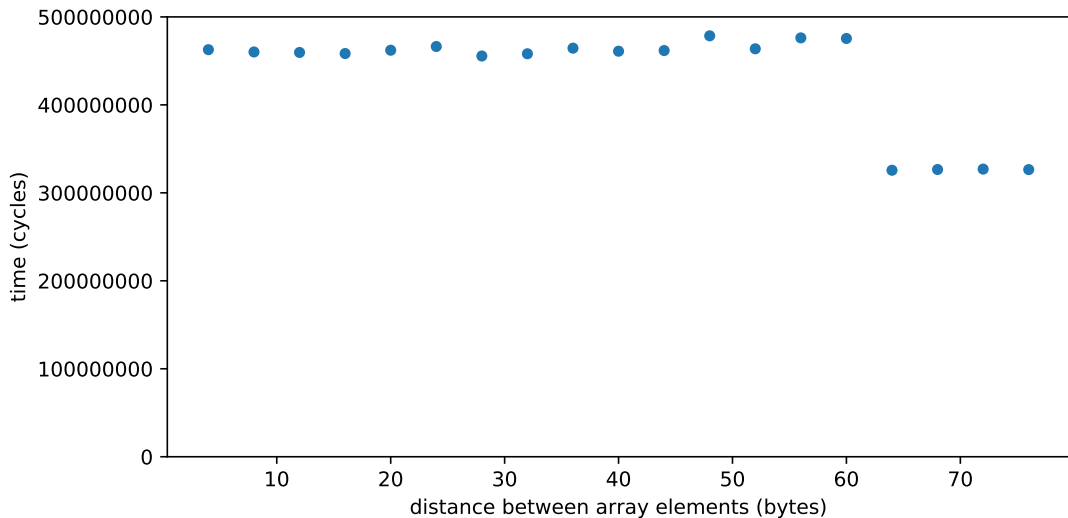
false sharing

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];
}
```

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;
}
```
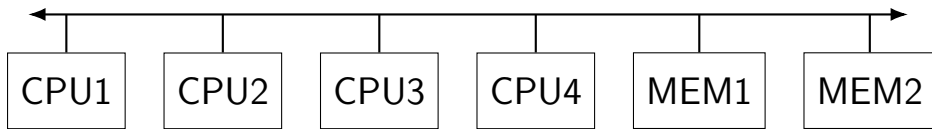
Where data-layout-like happens?

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
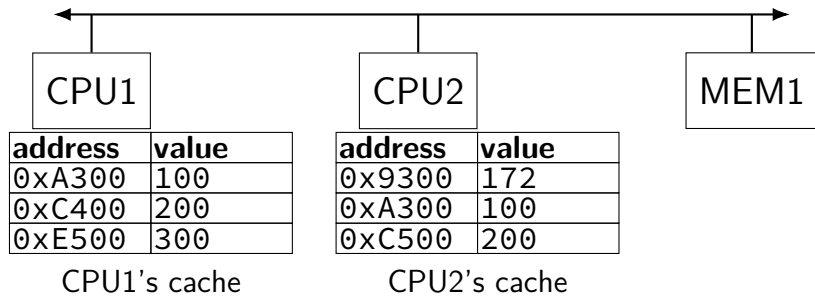
remember caches?

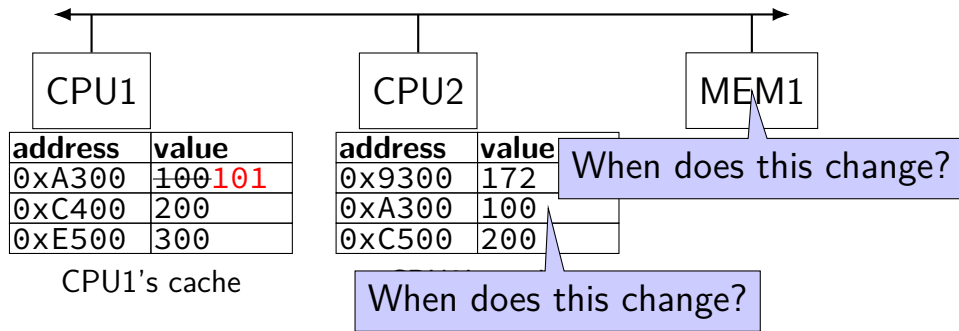memory is pretty slow

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

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

CPU1's cache

address	value
0x9300	172
0xA300	100
0xC500	200

When does this change?

When does this change?

CPU1 writes 101 to 0xA300?

BROKEN: producer/consumer signal

exercise: example why signal here is BROKEN? hint: two consume()+two produce()

```
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(&item);
    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	
			Produce(): wait for lock
wait for lock		enqueue size = 1? signal unlock	
			gets lock enqueue size ≠ 1: don't signal unlock
gets lock dequeue			

174

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

monitor exercise: solution (1)

(one of many possible solutions)

Assuming ConsumeTwo **replaces** Consume:

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  if (buffer.size() > 1) { pthread_cond_signal(&data_ready); }
  pthread_mutex_unlock(&lock);
}
ConsumeTwo() {
    pthread_mutex_lock(&lock);
    while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }
    item1 = buffer.dequeue(); item2 = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return Combine(item1, item2);
}
```

monitor exercise: solution (2)

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using two CVs):

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  pthread_cond_signal(&one_ready);
  if (buffer.size() > 1) { pthread_cond_signal(&two_ready); }
  pthread_mutex_unlock(&lock);
}
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 1) { pthread_cond_wait(&one_ready, &lock); }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
ConsumeTwo() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 2) { pthread_cond_wait(&two_ready, &lock); }
  item1 = buffer.dequeue(); item2 = buffer.dequeue();
  pthread_mutex_unlock(&lock);
```

monitor exercise: slower solution

(one of many possible solutions)

Assuming ConsumeTwo is **in addition to** Consume (using one CV):

```
Produce() {
  pthread_mutex_lock(&lock);
  buffer.enqueue(item);
  // broadcast and not signal, b/c we might wakeup only ConsumeTwo() otherwise
  pthread_cond_broadcast(&data_ready);
  pthread_mutex_unlock(&lock);
}
Consume() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 1) { pthread_cond_wait(&data_ready, &lock); }
  item = buffer.dequeue();
  pthread_mutex_unlock(&lock);
  return item;
}
ConsumeTwo() {
  pthread_mutex_lock(&lock);
  while (buffer.size() < 2) { pthread_cond_wait(&data_ready, &lock); }
  item1 = buffer.dequeue(); item2 = buffer.dequeue();
  pthread_mutex_unlock(&lock);
```

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

monitor ordering exercise: solution

(one of many possible solutions)

```
struct Waiter {
    pthread_cond_t cv;
    bool done;
    T item;
}
Queue<Waiter*> waiters;

Produce(item) {
 pthread_mutex_lock(&lock);
 if (!waiters.empty()) {
   Waiter *waiter = waiters.dequeue();
   waiter->done = true;
   waiter->item = item;
   cond_signal(&waiter->cv);
   ++num_pending;
 } else {
   buffer.enqueue(item);
 }
 pthread_mutex_unlock(&lock);
```

```
Consume() {
  pthread_mutex_lock(&lock);
  if (buffer.empty()) {
    Waiter waiter;
    cond_init(&waiter.cv);
    waiter.done = false;
    waiters.enqueue(&waiter);
    while (!waiter.done)
      cond_wait(&waiter.cv, &lock);
    item = waiter.item;
  } else {
    item = buffer.dequeue();
  }
  pthread_mutex_unlock(&lock):
  return item;
}
```

Anderson-Dahlin and semaphores

Anderson/Dahlin complains about semaphores
"Our view is that programming with locks and condition variables is superior to programming with semaphores."

argument 1: clearer to have separate constructs for
waiting for condition to be come true, and
allowing only one thread to manipulate a thing at a time

arugment 2: tricky to verify thread calls up exactly once for every down
alternatives allow one to be sloppier (in a sense)

monitors with semaphores: locks

```
sem_t semaphore;  // initial value 1

Lock() {
    sem_wait(&semaphore);
}

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

monitors with semaphores: [broken] cvs

start with only wait/signal:

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

monitors with semaphores: [broken] cvs

start with only wait/signal:

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

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

monitors with semaphores: cvs (better)

start with only wait/signal:

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

monitors with semaphores: broadcast

now allows broadcast:

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

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

building semaphore with monitors

```
pthread_mutex_t lock;
```

lock to protect shared state

building semaphore with monitors

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

lock to protect shared state
    shared state: semaphore tracks a count

building semaphore with monitors

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

lock to protect shared state
     shared state: semaphore tracks a count

add cond var for each reason we wait

building semaphore with monitors

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

lock to protect shared state
    shared state: semaphore tracks a count

add cond var for each reason we wait

building semaphore with monitors

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

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

lock to protect shared state
    shared state: semaphore tracks a count

add cond var for each reason we wait

binary semaphores

binary semaphores — semaphores that are only zero or one

as powerful as normal semaphores
> exercise: simulate counting semaphores with binary semaphores (more than one) and an integer

counting semaphores with binary semaphores

```
// assuming initialValue > 0
BinarySemaphore mutex(1);
int value = initialValue ;
BinarySemaphore gate(1 /* if initialValue >= 1 */);
     /* gate = # threads that can Down() now */
```

```
void Down() {                          void Up() {
  gate.Down();                           mutex.Down();
  // wait, if needed                     value += 1;
  mutex.Down();                          if (value == 1) {
  value -= 1;                              gate.Up();
  if (value > 0) {                         // because down should finish now
    gate.Up();                             // but could not before
    // because next down should finish   }
    // now (but not marked to before)     mutex.Up();
  }                                    }
  mutex.Up();
}
```

gate intuition/pattern

pattern to allow one thread at a time:

```
sem_t gate; // 0 = closed; 1 = open
ReleasingThread() {
    ... // finish what the other thread is waiting for
    while (another thread is waiting and can go) {
        sem_post(&gate)  // allow EXACTLY ONE thread
        ... // other bookkeeping
    }
    ...
}
WaitingThread() {
    ... // indicate that we're waiting
    sem_wait(&gate) // wait for gate to be open
    ... // indicate that we're not waiting
}
```