



## last time

multi-threaded processes

`pthread_create`: create new thread

`pthread_join`: collect thread return value, wait for thread

passing values to threads

# plan on threading topics

locks: avoiding conflicts between threads (beyond join)

interlude: deadlock

coordinating threads more than locks

# a threading race

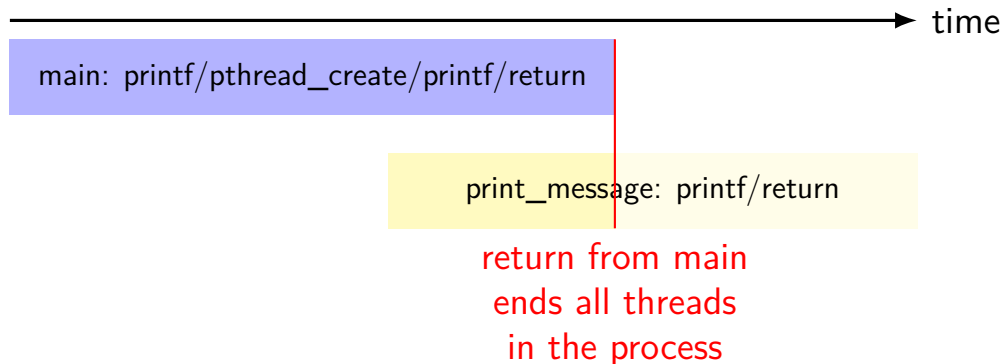
```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In the thread\n"); return NULL;
}
int main() {
    printf("About to start thread\n");
    pthread_t the_thread;
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done starting thread\n");
    return 0;
}
```

My machine: outputs In the thread about 4% of the time.  
What happened?

## 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?



# 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

→ 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

```
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
```

Thread B

# the lost write

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

---

Thread A

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

context switch

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

context switch

lost write to balance

Thread B

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

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

"winner" of the race

# the lost write

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

---

Thread A

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

context switch

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

context switch

lost write to balance

Thread B

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

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

“winner” of the race

lost track of thread A's money

# 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 (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  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$



## thinking about race conditions (3)

what are the possible values of  $x$ ?

(initially  $x = y = 0$ )

<b>Thread A</b>	<b>Thread B</b>
-----------------	-----------------

---

$x \leftarrow 1$
------------------

$x \leftarrow 2$
------------------

## thinking about race conditions (2)

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

Thread A	Thread B
----------	----------

$x \leftarrow y + 1$	$y \leftarrow 2$
	$y \leftarrow y \times 2$

# 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

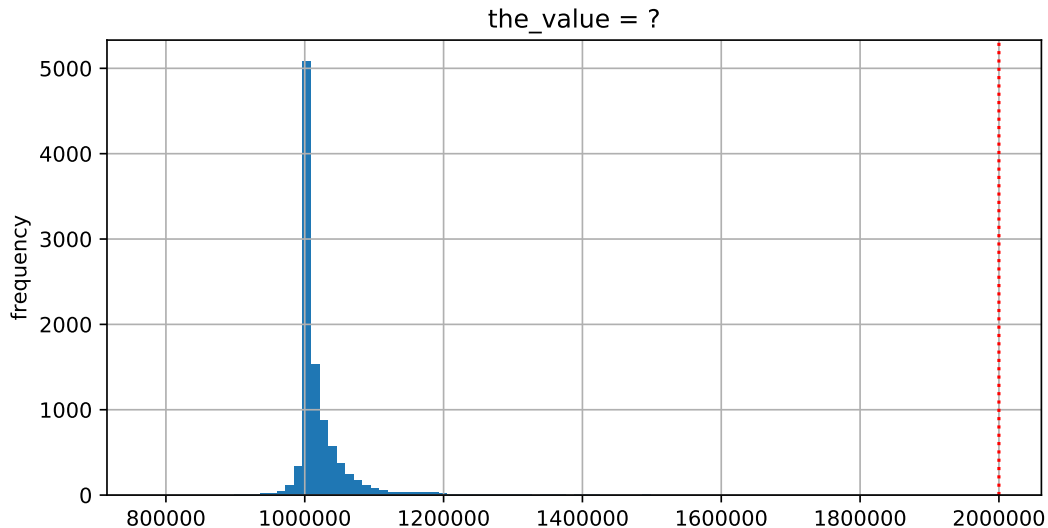
# lost adds (program)

```
.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);
}
```

# lost adds (results)



## 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



# atomic read-modify-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

$\text{temp} \leftarrow M[\text{ECX}]$

$M[\text{ECX}] \leftarrow \text{EAX}$

$\text{EAX} \leftarrow \text{temp}$

...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

## some definitions

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

like checking for and, if needed, buying milk

## some definitions

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

like checking for and, if needed, buying milk

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

result of critical section

# some definitions

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

like checking for and, if needed, buying milk

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

result of critical section

**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);
```

# pthread mutex

```
#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);
}
```

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

## exercise (alternate 1)

```
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;  
string one = "init one", two = "init two";
```

```
void ThreadA() {  
    pthread_mutex_lock(&lock2);  
    two = "two in ThreadA"; // (A2)  
    pthread_mutex_unlock(&lock2);  
    pthread_mutex_lock(&lock1);  
    one = "one in ThreadA"; // (A1)  
    pthread_mutex_unlock(&lock1);  
}
```

```
void ThreadB() {  
    pthread_mutex_lock(&lock1);  
    one = "one in ThreadB"; // (B1)  
    pthread_mutex_lock(&lock2);  
    two = "two in ThreadB"; // (B2)  
    pthread_mutex_unlock(&lock2);  
    pthread_mutex_unlock(&lock1);  
}
```

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

## exercise (alternate 2)

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

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

# POSIX mutex restrictions

pthread\_mutex rule: unlock from same thread you lock in

implementation I gave before — not a problem

...but there other ways to implement mutexes

e.g. might involve comparing with “holding” thread ID

# are locks enough?

do we need more than locks?

## example 1: pipes?

pipes: one thread reads while other writes

want write to complete immediately if buffer space

want read operation to *wait* for write operation

not functionality provided by mutexes/barriers



# barriers

compute minimum of 100M element array with 2 processors

algorithm:

compute minimum of 50M of the elements on each CPU

one thread for each CPU

wait for all computations to finish

take minimum of all the minimums

# barriers

compute minimum of 100M element array with 2 processors

algorithm:

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

wait for all computations to finish

take minimum of all the minimums

# barriers API

`barrier.Initialize(NumberOfThreads)`

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

idea: multiple threads perform computations in parallel

threads wait for **all other threads** to call `Wait()`

# barrier: waiting for finish

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

Thread 0

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

```
barrier.Wait();
```

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

Thread 1

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

## barriers: reuse

### Thread 0

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

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

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

### Thread 1

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

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

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

## barriers: reuse

### Thread 0

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

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

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

### Thread 1

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

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

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

## barriers: reuse

### Thread 0

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

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

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

### Thread 1

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

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

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

# pthread barriers

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



# life homework (pseudocode)

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

# life homework

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

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

parallel version: produce parts of grid in different threads

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

# compilers move loads/stores (1)

```
void WaitForOther() {  
    do {} while (!other_ready);  
}
```

---

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

# compilers move loads/stores (1)

```
void WaitForOther() {  
    do {} while (!other_ready);  
}
```

---

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

## compilers move loads/stores too (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 too (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 too (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
```

## a simple race

thread\_A:

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

thread\_B:

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

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



## a simple race

thread\_A:

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

thread\_B:

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

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

if loads/stores atomic, then possible results:

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

# a simple race: results

thread\_A:

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

thread\_B:

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

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

my desktop, 100M trials:

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

# a simple race: results

thread\_A:

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

thread\_B:

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

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

my desktop, 100M trials:

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

# why reorder here?

thread\_A:

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

thread\_B:

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

thread A: faster to load y right now!

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

# why load/store reordering?

fast processor designs can execute instructions out of order

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

more on this later in the semester

# pthread and reordering

many pthreads functions **prevent reordering**

everything before function call actually happens before

includes **preventing some optimizations**

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

pthread\_mutex\_lock/unlock, pthread\_create, pthread\_join, ...

basically: if pthreads is waiting for/starting something, no weird ordering

implementation part 1: prevent compiler reordering

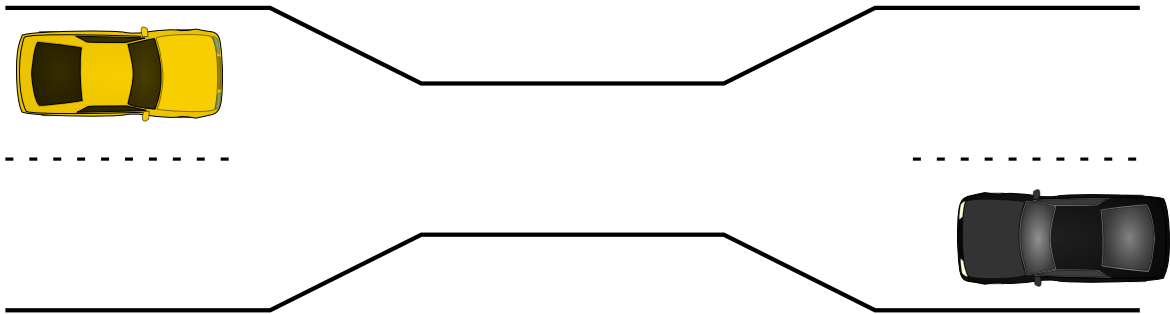
implementation part 2: use special instructions

example: x86 mfence instruction

# interlude: deadlock

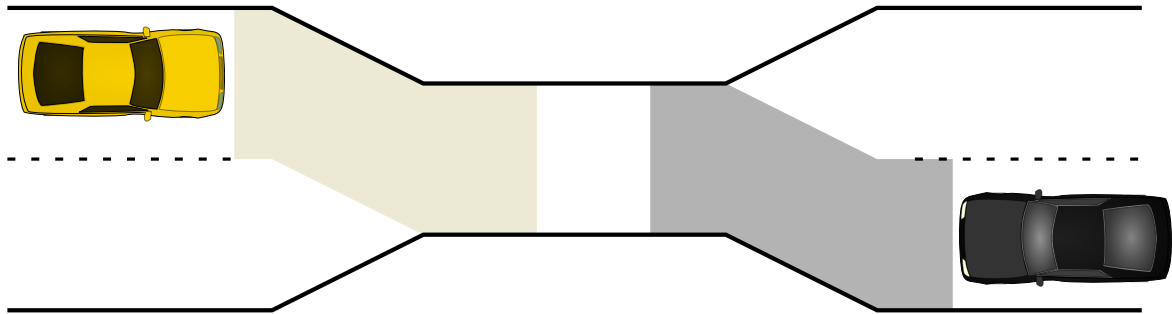
using multiple locks is tricky...

# the one-way bridge

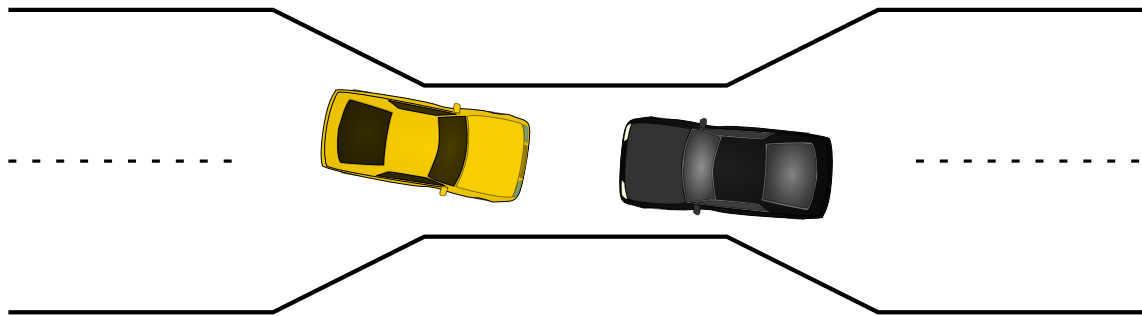




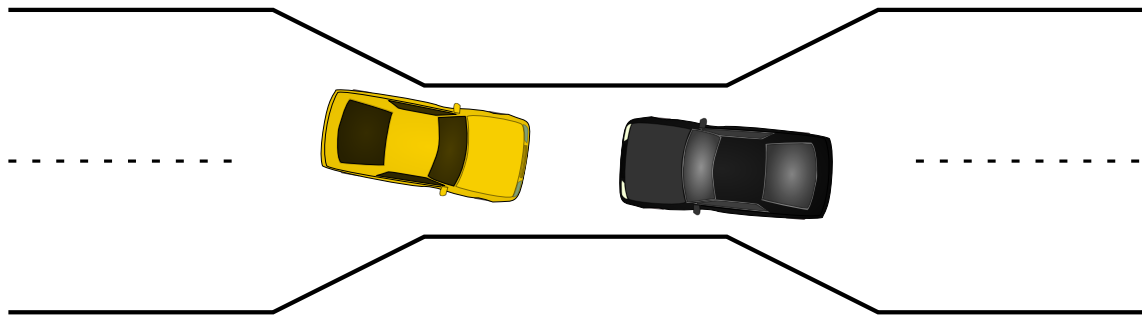
## the one-way bridge



## the one-way bridge



## the one-way bridge



## moving two files

```
struct Dir {  
    mutex_t lock; map<string, DirEntry> entries;  
};  
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {  
    mutex_lock(&from_dir->lock);  
    mutex_lock(&to_dir->lock);  
  
    to_dir->entries[filename] = from_dir->entries[filename];  
    from_dir->entries.erase(filename);  
  
    mutex_unlock(&to_dir->lock);  
    mutex_unlock(&from_dir->lock);  
}
```

Thread 1: MoveFile(A, B, "foo")

Thread 2: MoveFile(B, A, "bar")

## moving two files: lucky timeline (1)

### Thread 1

MoveFile(A, B, "foo")

---

lock(&A->lock);

lock(&B->lock);

(do move)

unlock(&B->lock);

unlock(&A->lock);

### Thread 2

MoveFile(B, A, "bar")

---

lock(&B->lock);

lock(&A->lock);

(do move)

unlock(&B->lock);

unlock(&A->lock);

## moving two files: lucky timeline (2)

### Thread 1

MoveFile(A, B, "foo")

---

lock(&A->lock);

lock(&B->lock);

(do move)

unlock(&B->lock);

unlock(&A->lock);

### Thread 2

MoveFile(B, A, "bar")

---

lock(&B->lock...

(waiting for B lock)

lock(&B->lock);

lock(&A->lock...

lock(&A->lock);

(do move)

unlock(&A->lock);

unlock(&B->lock);

## moving two files: unlucky timeline

### Thread 1

```
MoveFile(A, B, "foo")
```

---

```
lock(&A->lock);
```

### Thread 2

```
MoveFile(B, A, "bar")
```

---

```
lock(&B->lock);
```

# moving two files: unlucky timeline

## Thread 1

MoveFile(A, B, "foo")

lock(&A->lock);

lock(&B->lock... stalled

(waiting for lock on B)

(waiting for lock on B)

## Thread 2

MoveFile(B, A, "bar")

lock(&B->lock);

lock(&A->lock... stalled

(waiting for lock on A)



# moving two files: unlucky timeline

## Thread 1

```
MoveFile(A, B, "foo")
```

```
lock(&A->lock);
```

```
lock(&B->lock... stalled
```

```
(waiting for lock on B)
```

```
(waiting for lock on B)
```

```
(do move) unreachable
```

```
unlock(&B->lock); unreachable
```

```
unlock(&A->lock); unreachable
```

## Thread 2

```
MoveFile(B, A, "bar")
```

```
lock(&B->lock);
```

```
lock(&A->lock... stalled
```

```
(waiting for lock on A)
```

```
(do move) unreachable
```

```
unlock(&A->lock); unreachable
```

```
unlock(&B->lock); unreachable
```

# moving two files: unlucky timeline

## Thread 1

```
MoveFile(A, B, "foo")
```

```
lock(&A->lock);
```

```
lock(&B->lock... stalled
```

```
(waiting for lock on B)
```

```
(waiting for lock on B)
```

```
(do move) unreachable
```

```
unlock(&B->lock); unreachable
```

```
unlock(&A->lock); unreachable
```

## Thread 2

```
MoveFile(B, A, "bar")
```

```
lock(&B->lock);
```

```
lock(&A->lock... stalled
```

```
(waiting for lock on A)
```

```
(do move) unreachable
```

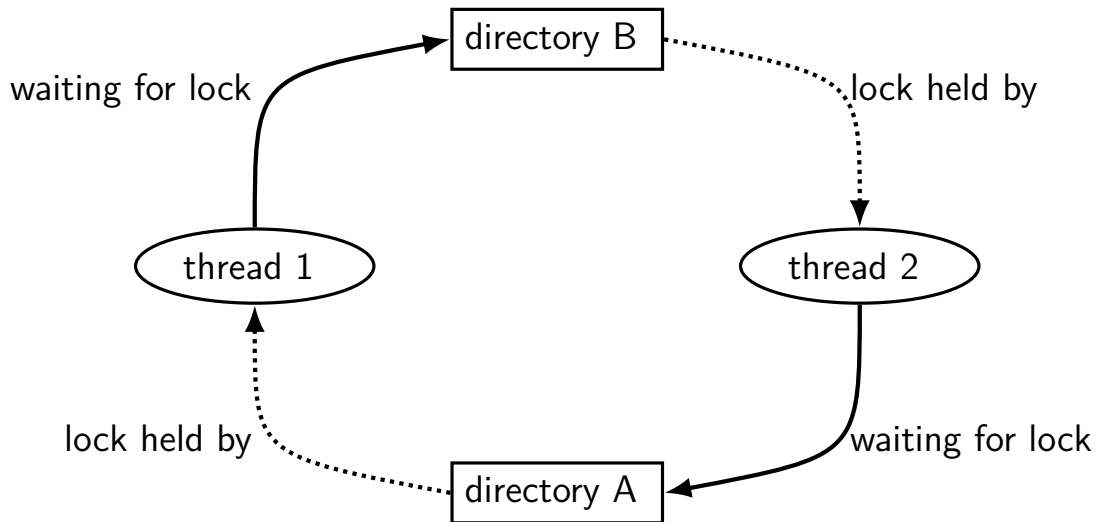
```
unlock(&A->lock); unreachable
```

```
unlock(&B->lock); unreachable
```

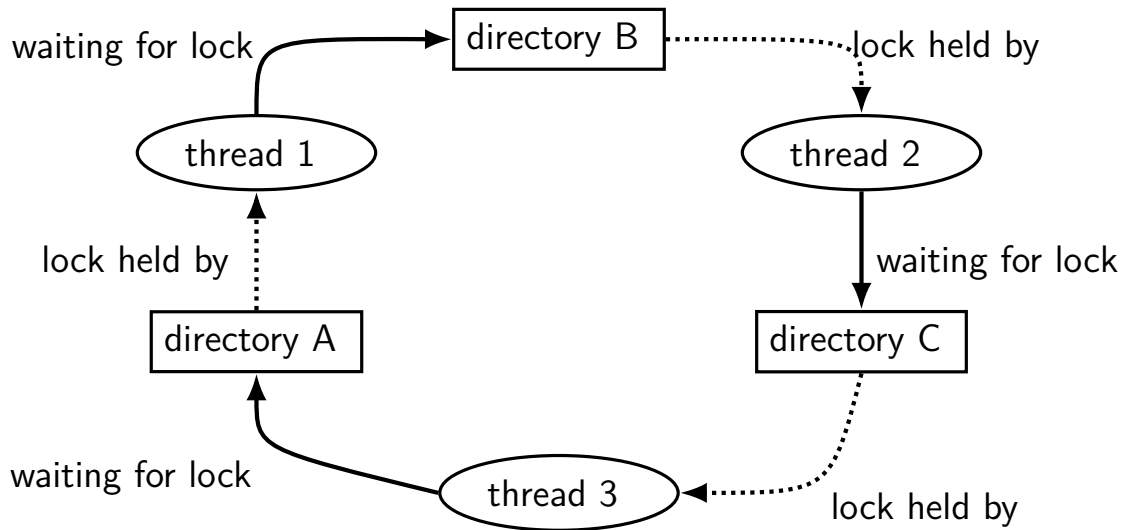
Thread 1 holds A lock, waiting for Thread 2 to release B lock

Thread 2 holds B lock, waiting for Thread 1 to release A lock

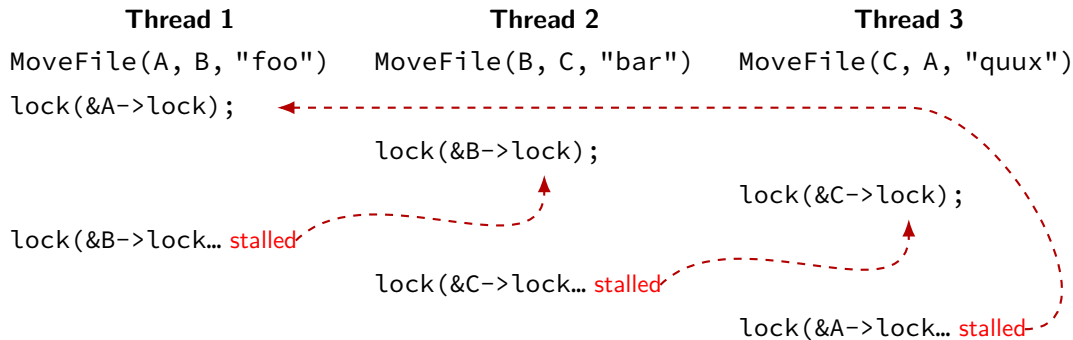
## moving two files: dependencies



## moving three files: dependencies



# moving three files: unlucky timeline



# deadlock with free space

## Thread 1

AllocateOrWaitFor(1 MB)  
AllocateOrWaitFor(1 MB)  
(do calculation)  
Free(1 MB)  
Free(1 MB)

## Thread 2

AllocateOrWaitFor(1 MB)  
AllocateOrWaitFor(1 MB)  
(do calculation)  
Free(1 MB)  
Free(1 MB)

2 MB of space — deadlock possible with unlucky order

# deadlock with free space (unlucky case)

## Thread 1

AllocateOrWaitFor(1 MB)

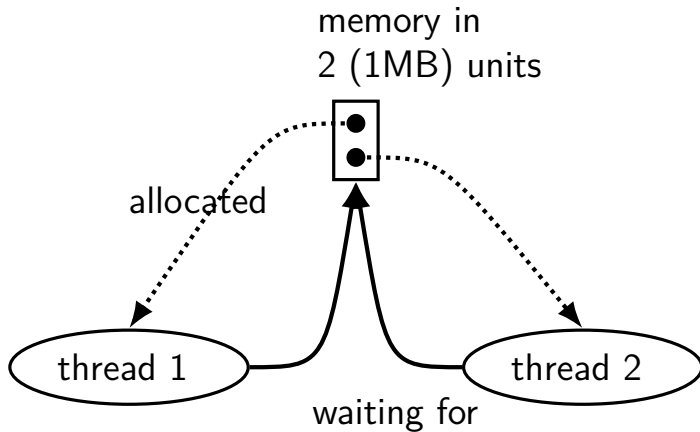
AllocateOrWaitFor(1 MB... stalled

## Thread 2

AllocateOrWaitFor(1 MB)

AllocateOrWaitFor(1 MB... stalled

# free space: dependency graph





# deadlock with free space (lucky case)

## Thread 1

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB)
(do calculation)
Free(1 MB);
Free(1 MB);
```

## Thread 2

```
AllocateOrWaitFor(1 MB)
AllocateOrWaitFor(1 MB)
(do calculation)
Free(1 MB);
Free(1 MB);
```

# deadlock

deadlock — circular waiting for resources

resource = something needed by a thread to do work

- locks

- CPU time

- disk space

- memory

- ...

often non-deterministic in practice

most common example: **when acquiring multiple locks**

# deadlock

deadlock — circular waiting for **resources**

resource = something needed by a thread to do work

- locks

- CPU time

- disk space

- memory

- ...

often non-deterministic in practice

most common example: **when acquiring multiple locks**

# deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

# deadlock versus starvation

starvation: one+ unlucky (no progress), one+ lucky (yes progress)

example: low priority threads versus high-priority threads

deadlock: no one involved in deadlock makes progress

starvation: once starvation happens, taking turns will resolve

low priority thread just needed a chance...

deadlock: once it happens, taking turns won't fix

# deadlock requirements

## mutual exclusion

one thread at a time can use a resource

## hold and wait

thread holding a resources waits to acquire *another* resource

## no preemption of resources

resources are only released voluntarily

thread trying to acquire resources can't 'steal'

## circular wait

there exists a set  $\{T_1, \dots, T_n\}$  of waiting threads such that

$T_1$  is waiting for a resource held by  $T_2$

$T_2$  is waiting for a resource held by  $T_3$

...

$T_n$  is waiting for a resource held by  $T_1$

# how is deadlock possible?

Given list: A, B, C, D, E

```
RemoveNode(LinkedListNode *node) {  
    pthread_mutex_lock(&node->lock);  
    pthread_mutex_lock(&node->prev->lock);  
    pthread_mutex_lock(&node->next->lock);  
    node->next->prev = node->prev;  
    node->prev->next = node->next;  
    pthread_mutex_unlock(&node->next->lock);  
    pthread_mutex_unlock(&node->prev->lock);  
    pthread_mutex_unlock(&node->lock);  
}
```

Which of these (all run in parallel) can deadlock?

- A. RemoveNode(B) and RemoveNode(C)
- B. RemoveNode(B) and RemoveNode(D)
- C. RemoveNode(B) and RemoveNode(C) and RemoveNode(D)
- D. A and C
- E. B and C
- F. all of the above
- G. none of the above

# deadlock prevention techniques

## **infinite resources**

or at least enough that never run out

*no mutual exclusion*

## **no shared resources**

*no mutual exclusion*

## **no waiting**

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*



# deadlock prevention techniques

## infinite resources

or at least enough that never run out

*no mutual exclusion*

## no shared resources

*no mutual exclusion*

## no waiting

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

# deadlock prevention techniques

## infinite resources

or at least enough that never run out

no *mutual exclusion*

## no shared resources

no *mutual exclusion*

## no waiting

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

no *hold and wait*/  
*preemption*

acquire resources in **consistent order**

no *circular wait*

request **all resources at once**

no *hold and wait*

# deadlock prevention techniques

## infinite resources

or at least enough that never run out

*no mutual exclusion*

memory allocation: malloc() fails rather than waiting (no deadlock)

locks: pthread\_mutex\_trylock fails rather than waiting

...

*exclusion*

## no waiting

“busy signal” — abort and (maybe) retry

revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

# deadlock prevention techniques

## infinite resources

or at least enough that never run out

*no mutual exclusion*

## no shared resources

*no mutual exclusion*

requires some way to undo partial changes to avoid errors  
common approach for databases

## no waiting

...

“busy signal” — abort and (maybe) retry

revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

# deadlock prevention techniques

## **infinite resources**

or at least enough that never run out

*no mutual exclusion*

## **no shared resources**

*no mutual exclusion*

## **no waiting**

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

## acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {  
    if (from_dir->path < to_dir->path) {  
        lock(&from_dir->lock);  
        lock(&to_dir->lock);  
    } else {  
        lock(&to_dir->lock);  
        lock(&from_dir->lock);  
    }  
    ...  
}
```

# acquiring locks in consistent order (1)

```
MoveFile(Dir* from_dir, Dir* to_dir, string filename) {  
    if (from_dir->path < to_dir->path) {  
        lock(&from_dir->lock);  
        lock(&to_dir->lock);  
    } else {  
        lock(&to_dir->lock);  
        lock(&from_dir->lock);  
    }  
    ...  
}
```

any ordering will do  
e.g. compare pointers

## acquiring locks in consistent order (2)

often by convention, e.g. Linux kernel comments:

```
/*  
 * ...  
 * Lock order:  
 *     contex.ldt_usr_sem  
 *     mmap_sem  
 *     context.lock  
 */
```

---

```
/*  
 * ...  
 * Lock order:  
 * 1. slab_mutex (Global Mutex)  
 * 2. node->list_lock  
 * 3. slab_lock(page) (Only on some arches and for debugging)  
 * ...  
 */
```



# deadlock prevention techniques

## infinite resources

or at least enough that never run out

*no mutual exclusion*

## no shared resources

*no mutual exclusion*

## no waiting

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

# deadlock detection

why? debugging or fix deadlock by aborting operations

idea: search for cyclic dependencies

# detecting deadlocks on locks

let's say I want to detect deadlocks that only involve mutexes

goal: help programmers debug deadlocks

...by modifying my threading library:

```
struct Thread {  
    ... /* stuff for implementing thread */  
    /* what extra fields go here? */  
};  
  
struct Mutex {  
    ... /* stuff for implementing mutex */  
    /* what extra fields go here? */  
};
```

# deadlock detection

why? debugging or fix deadlock by aborting operations

idea: search for cyclic dependencies

need:

- list of all contended resources

- what thread is waiting for what?

- what thread 'owns' what?

## aside: divisible resources

deadlock is possible with divisible resources like memory,...

example: suppose 6MB of RAM for threads total:

- thread 1 has 2MB allocated, waiting for 2MB

- thread 2 has 2MB allocated, waiting for 2MB

- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

not a deadlock — thread 3 can still finish

- and after it does, thread 1 or 2 can finish

## aside: divisible resources

deadlock is possible with divisible resources like memory,...

example: suppose 6MB of RAM for threads total:

- thread 1 has 2MB allocated, waiting for 2MB

- thread 2 has 2MB allocated, waiting for 2MB

- thread 3 has 1MB allocated, waiting for keypress

cycle: thread 1 waiting on memory owned by thread 2?

not a deadlock — thread 3 can still finish

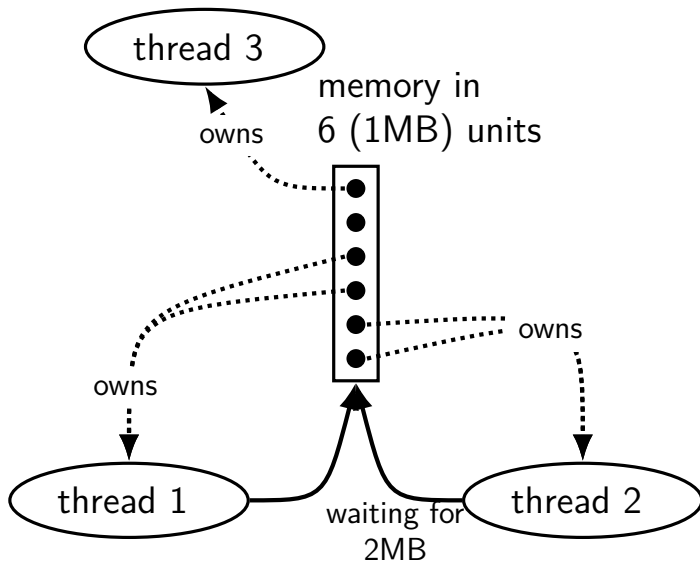
- and after it does, thread 1 or 2 can finish

...but would be deadlock

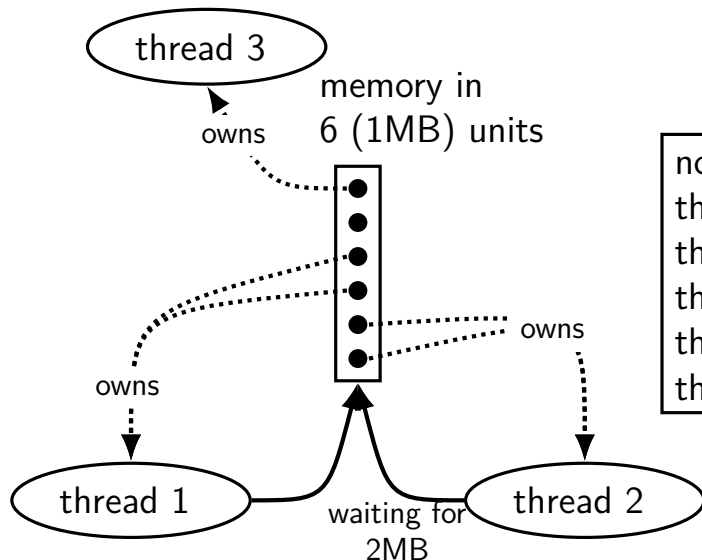
- ...if thread 3 waiting lock held by thread 1

- ...with 5MB of RAM

# divisible resources: not deadlock



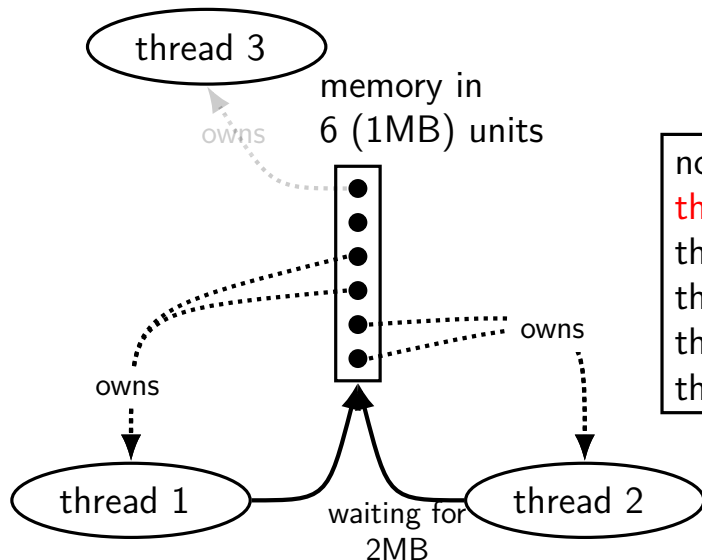
# divisible resources: not deadlock



not deadlock:  
thread 3 finishes  
then thread 1 can get memory  
then thread 1 finishes  
then thread 2 can get resources  
then thread 2 can finish



# divisible resources: not deadlock



not deadlock:

**thread 3 finishes**

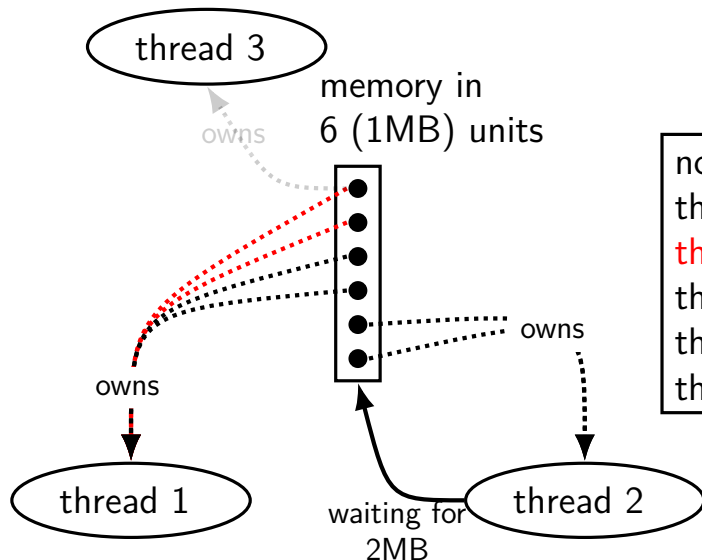
then thread 1 can get memory

then thread 1 finishes

then thread 2 can get resources

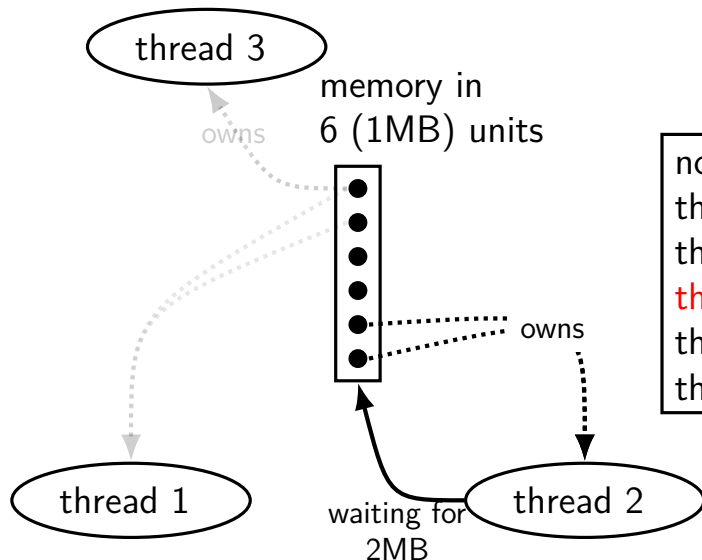
then thread 2 can finish

# divisible resources: not deadlock



not deadlock:  
thread 3 finishes  
then thread 1 can get memory  
then thread 1 finishes  
then thread 2 can get resources  
then thread 2 can finish

# divisible resources: not deadlock



not deadlock:

thread 3 finishes

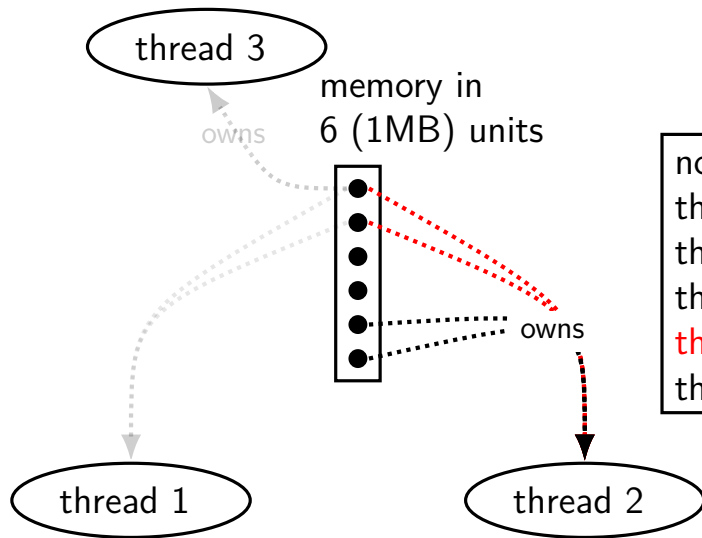
then thread 1 can get memory

**then thread 1 finishes**

then thread 2 can get resources

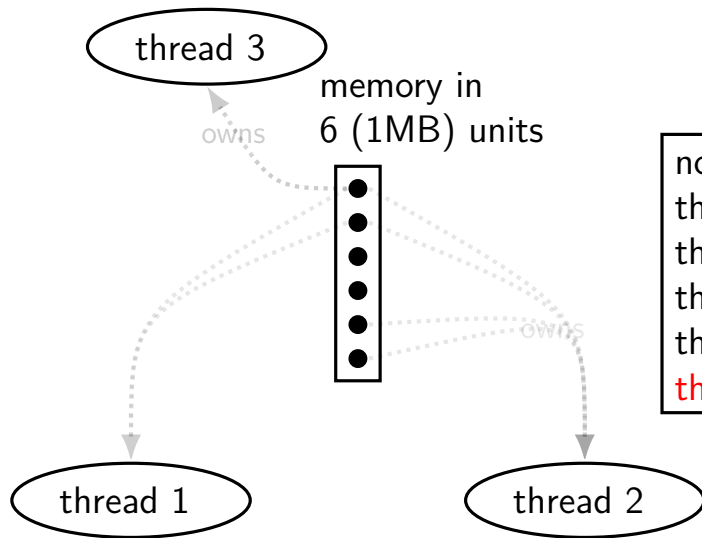
then thread 2 can finish

# divisible resources: not deadlock



not deadlock:  
thread 3 finishes  
then thread 1 can get memory  
then thread 1 finishes  
**then thread 2 can get resources**  
then thread 2 can finish

# divisible resources: not deadlock



not deadlock:

thread 3 finishes

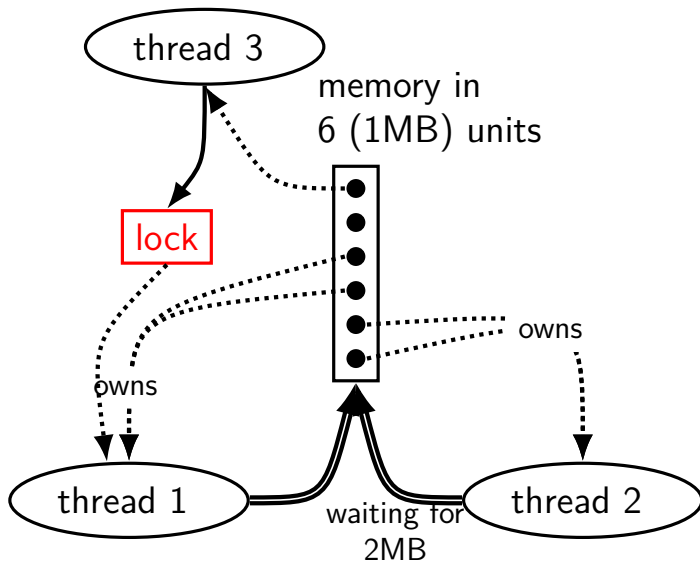
then thread 1 can get memory

then thread 1 finishes

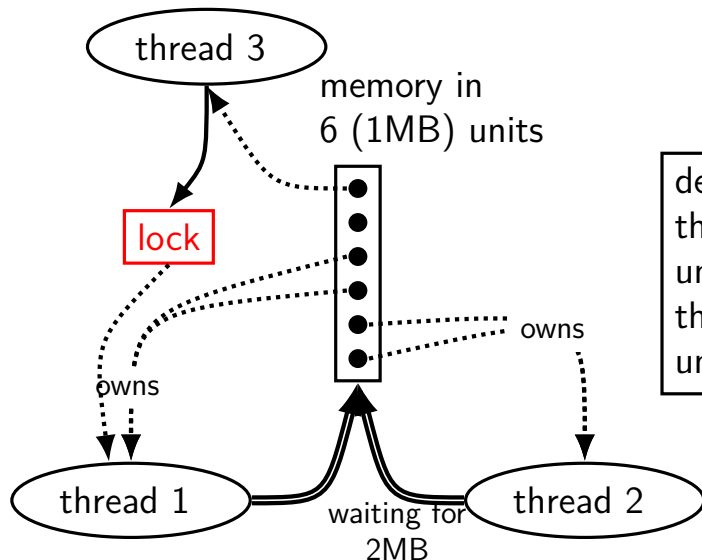
then thread 2 can get resources

**then thread 2 can finish**

# divisible resources: is deadlock



## divisible resources: is deadlock



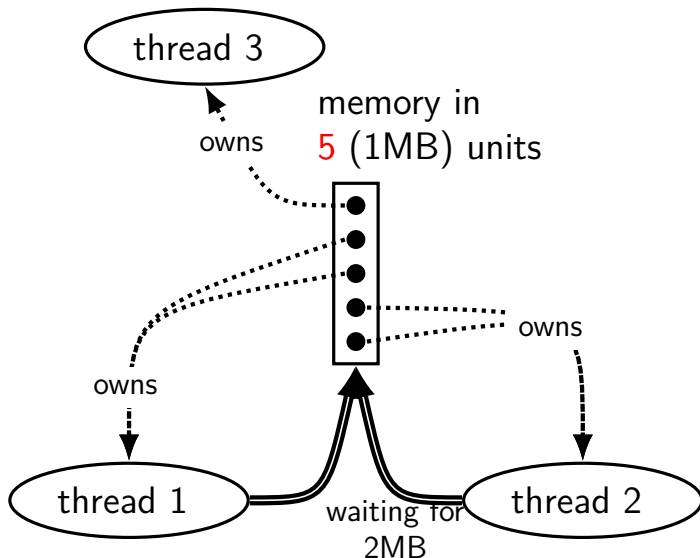
deadlock:

thread 3 can't finish

until thread 1 releases lock, but  
thread 1 can't finish

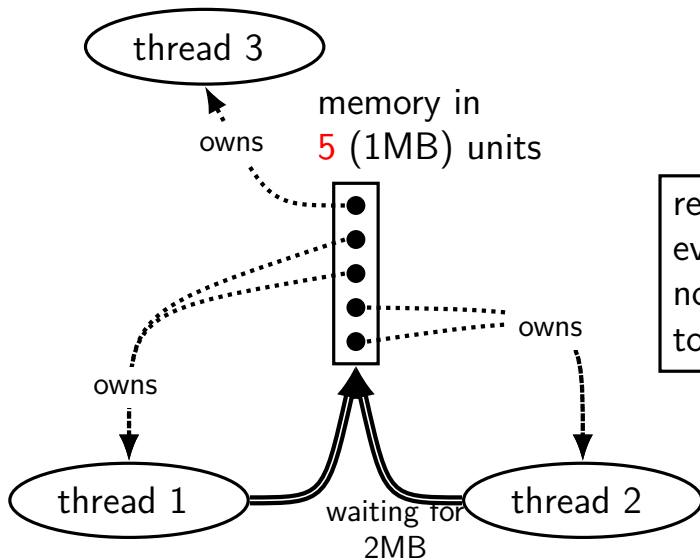
until thread 3 releases memory

# divisible resources: is deadlock



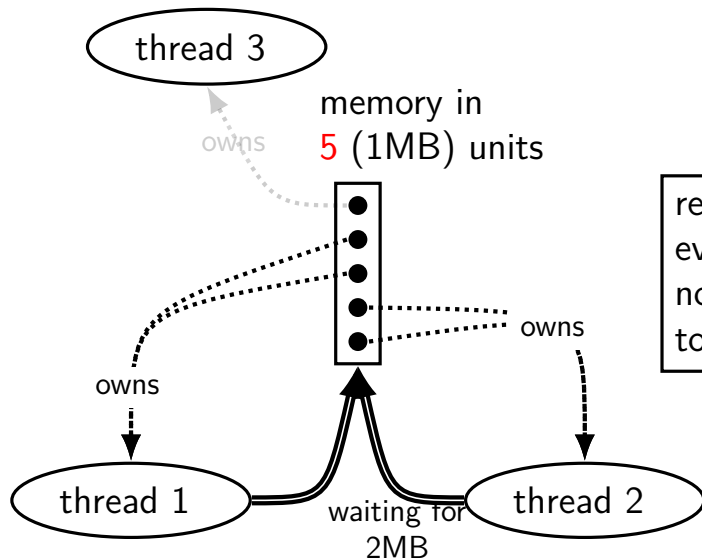


# divisible resources: is deadlock



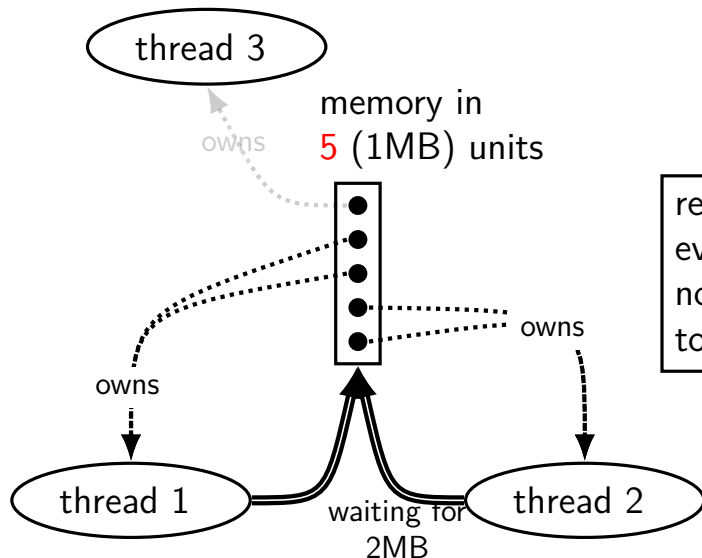
reducing memory: deadlock:  
even after thread 3 finishes  
no way for thread 1+2  
to get what they want

# divisible resources: is deadlock



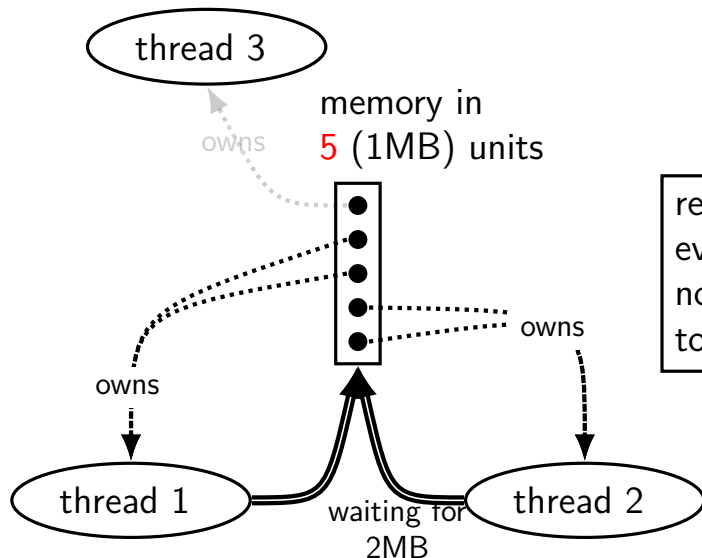
reducing memory: deadlock:  
even after thread 3 finishes  
no way for thread 1+2  
to get what they want

# divisible resources: is deadlock



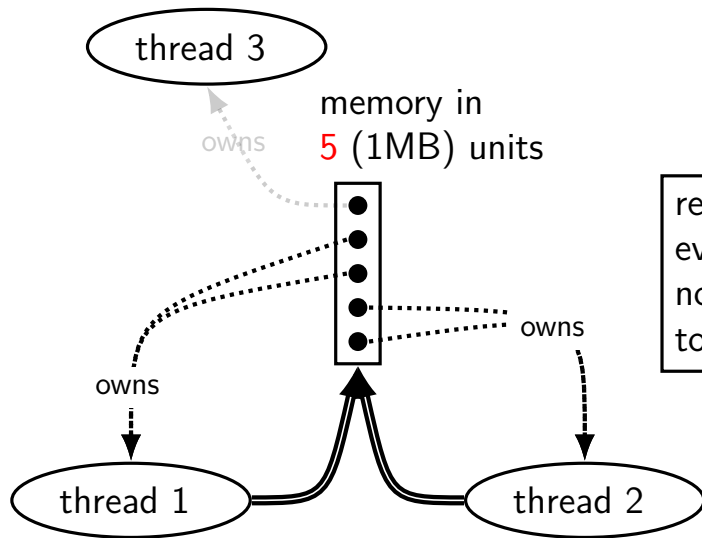
reducing memory: deadlock:  
even after thread 3 finishes  
no way for thread 1+2  
to get what they want

# divisible resources: is deadlock



reducing memory: deadlock:  
even after thread 3 finishes  
no way for thread 1+2  
to get what they want

# divisible resources: is deadlock



reducing memory: deadlock:  
even after thread 3 finishes  
no way for thread 1+2  
to get what they want

# deadlock detection with divisible resources

can't rely on cycles in graphs in this case

alternate algorithm exists

- similar technique to how we showed no deadlock

high-level intuition: simulate what could happen

- find threads that could finish based on resources available now

full details: look up Baker's algorithm

# stealing locks???

how do we make stealing locks possible

unclean: just kill the thread

problem: inconsistent state?

clean: have code to undo partial operation

some databases do this

won't go into detail in this class

# revokable locks?

```
try {  
    AcquireLock();  
    use shared data  
} catch (LockRevokedException le) {  
    undo operation hopefully?  
} finally {  
    ReleaseLock();  
}
```



# deadlock prevention techniques

## infinite resources

or at least enough that never run out

*no mutual exclusion*

## no shared resources

*no mutual exclusion*

## no waiting

“busy signal” — abort and (maybe) retry  
revoke/preempt resources

*no hold and wait/  
preemption*

acquire resources in **consistent order**

*no circular wait*

request **all resources at once**

*no hold and wait*

# abort and retry limits?

abort-and-retry

how many times will you retry?

## moving two files: abort-and-retry

```
struct Dir {  
    mutex_t lock; map<string, DirEntry> entries;  
};  
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {  
    while (true) {  
        mutex_lock(&from_dir->lock);  
        if (mutex_trylock(&to_dir->lock) == LOCKED) break;  
        mutex_unlock(&from_dir->lock);  
    }  
  
    to_dir->entries[filename] = from_dir->entries[filename];  
    from_dir->entries.erase(filename);  
  
    mutex_unlock(&to_dir->lock);  
    mutex_unlock(&from_dir->lock);  
}
```

Thread 1: MoveFile(A, B, "foo")

Thread 2: MoveFile(B, A, "bar")

## moving two files: lots of bad luck?

### Thread 1

MoveFile(A, B, "foo")

lock(&A->lock) → LOCKED

trylock(&B->lock) → FAILED

unlock(&A->lock)

lock(&A->lock) → LOCKED

trylock(&B->lock) → FAILED

unlock(&A->lock)

### Thread 2

MoveFile(B, A, "bar")

lock(&B->lock) → LOCKED

trylock(&A->lock) → FAILED

unlock(&B->lock)

lock(&B->lock) → LOCKED

trylock(&A->lock) → FAILED

unlock(&B->lock)

# livelock

livelock: keep aborting and retrying without end

like deadlock — no one's making progress  
potentially forever

unlike deadlock — threads are not waiting

# preventing livelock

make schedule random — e.g. random waiting after abort

make threads run one-at-a-time if lots of aborting

other ideas?

**backup slides**