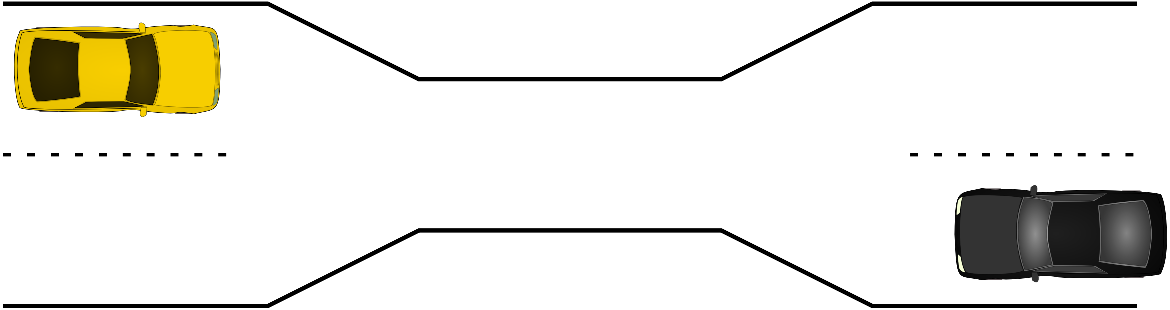
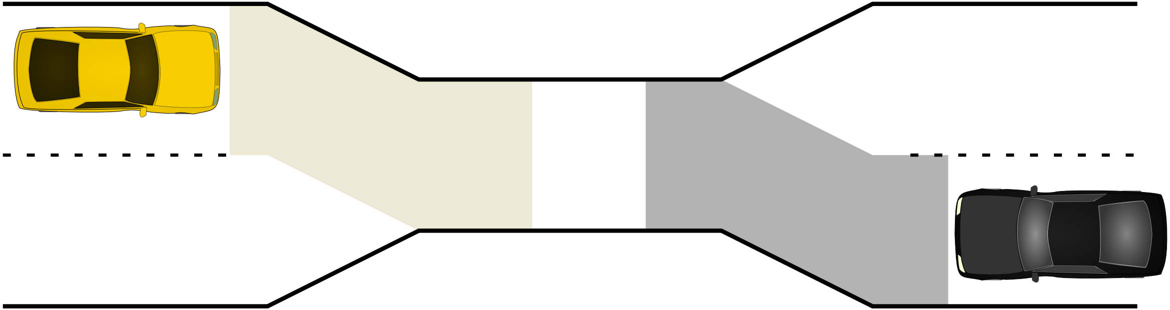
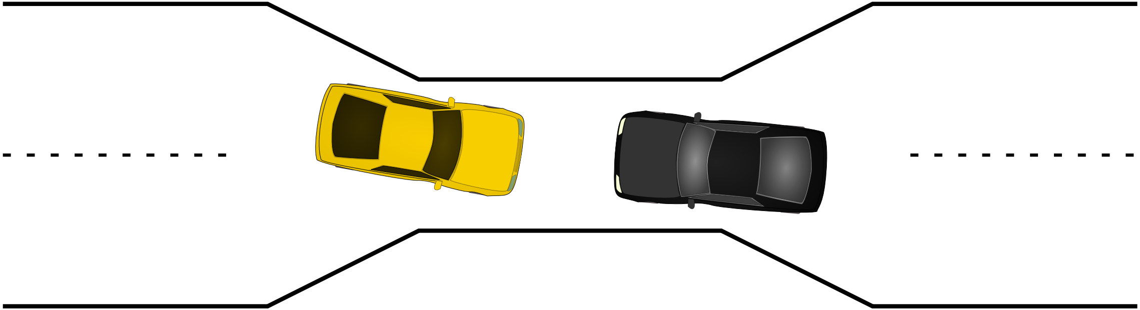
the one-way bridge



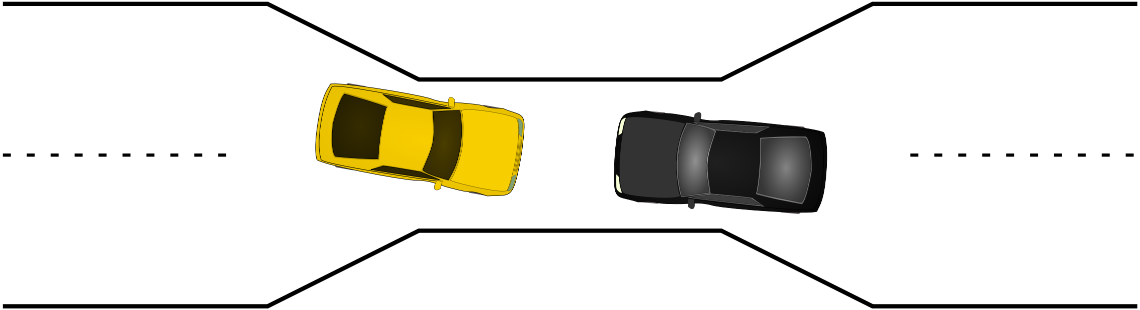
the one-way bridge



the one-way bridge



the one-way bridge



moving two files

```
struct Dir {
  mutex_t lock; HashMap entries;
};
void MoveFile(Dir *from_dir, Dir *to_dir, string filename) {
  mutex_lock(&from_dir->lock);
  mutex_lock(&to_dir->lock);

  HashMap_put(to_dir->entries, filename, HashMap_get(from_dir->entr
  HashMap_erase(from_dir->entries, 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	Thread 2
`MoveFile(A, B, "foo")`	`MoveFile(B, A, "bar")`
`lock(&A->lock);`	
`lock(&B->lock);`	
(do move)	
`unlock(&B->lock);`	
`unlock(&A->lock);`	
	`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")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
`lock(&B->lock);`	
	`lock(&B->lock…`
(do move)	(waiting for B lock)
`unlock(&B->lock);`	
	`lock(&B->lock);`
	`lock(&A->lock…`
`unlock(&A->lock);`	
	`lock(&A->lock);`
	(do move)
	`unlock(&A->lock);`
	`unlock(&B->lock);`

moving two files: unlucky timeline

Thread 1	Thread 2
`MoveFile(A, B, "foo")`	`MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`

moving two files: unlucky timeline

Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`… stalled	
(waiting for lock on B)	`lock(&A->lock`… stalled
(waiting for lock on B)	(waiting for lock on A)

moving two files: unlucky timeline

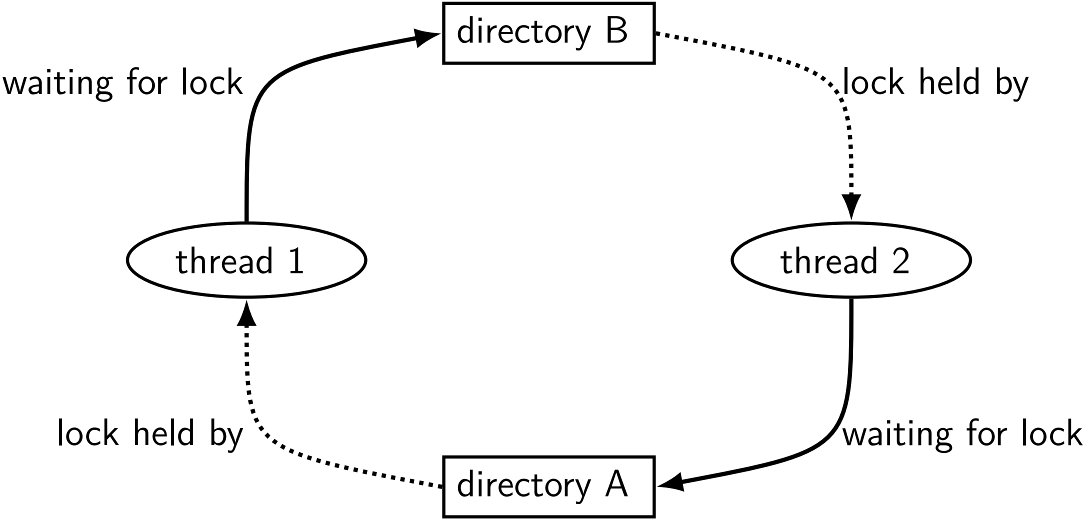
Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock…` stalled	
(waiting for lock on B)	`lock(&A->lock…` stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~ unreachable	~~`unlock(&A->lock);`~~ unreachable
~~`unlock(&A->lock);`~~ unreachable	~~`unlock(&B->lock);`~~ unreachable

moving two files: unlucky timeline

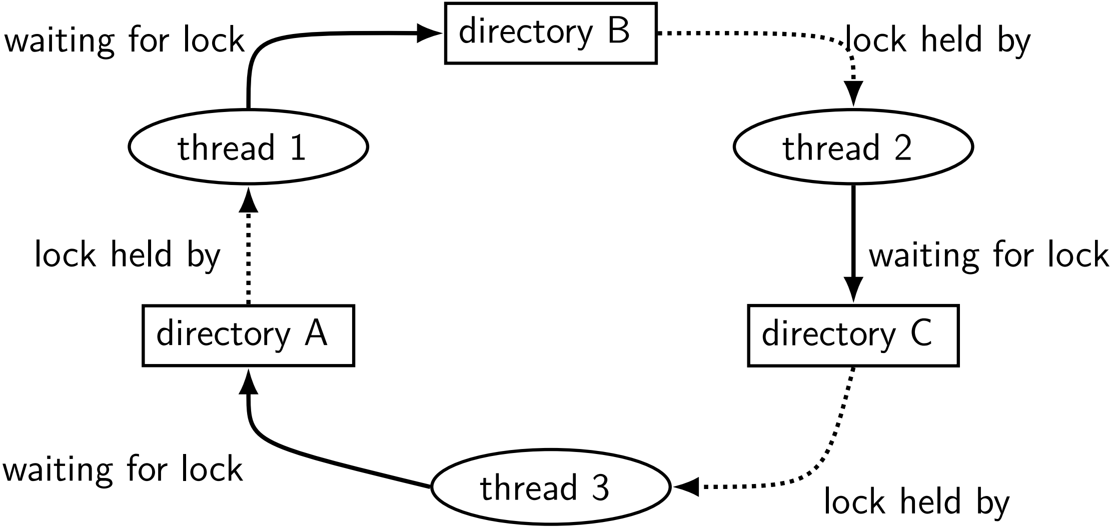
Thread 1	Thread 2
`MoveFile(A, B, "foo")`	`MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`... stalled	
(waiting for lock on B)	`lock(&A->lock`... stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~ unreachable	~~`unlock(&A->lock);`~~ 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

Thread 1	Thread 2	Thread 3
`MoveFile(A, B, "foo")`	`MoveFile(B, C, "bar")`	`MoveFile(C, A, "quux")`
`lock(&A->lock);`		
	`lock(&B->lock);`	
		`lock(&C->lock);`
`lock(&B->lock`… stalled		
	`lock(&C->lock`… stalled	
		`lock(&A->lock`… stalled

deadlock with free space

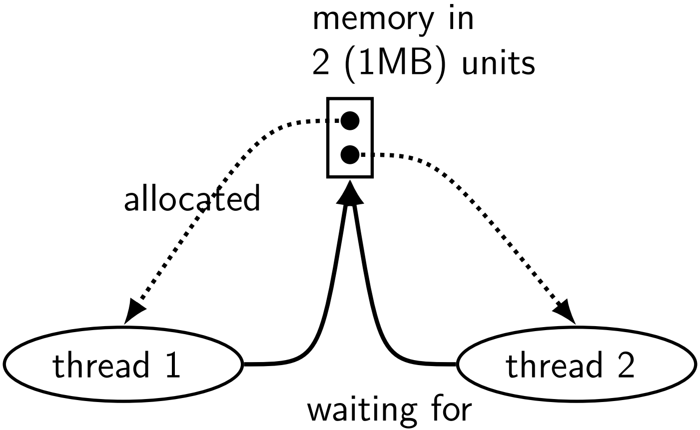
Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

2 MB of space — deadlock possible with unlucky order

deadlock with free space (unlucky case)

Thread 1	**Thread 2**
`AllocateOrWaitFor(1 MB)`	
	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB...` stalled	
	`AllocateOrWaitFor(1 MB...` stalled

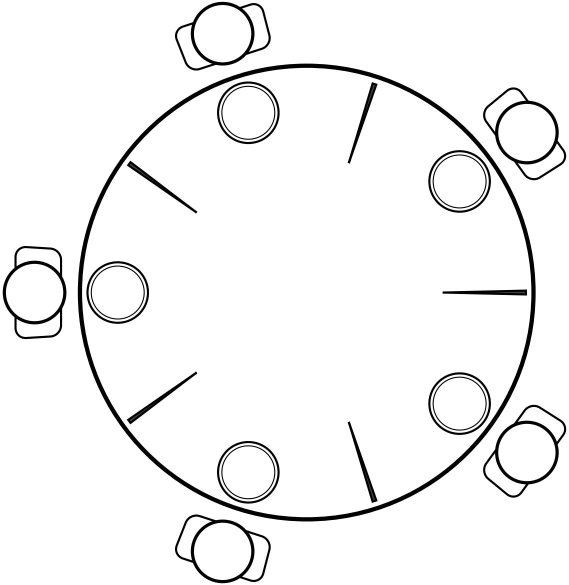
free space: dependency graph



deadlock with free space (lucky case)

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	
`AllocateOrWaitFor(1 MB)`	
`(do calculation)`	
`Free(1 MB);`	
`Free(1 MB);`	
	`AllocateOrWaitFor(1 MB)`
	`AllocateOrWaitFor(1 MB)`
	`(do calculation)`
	`Free(1 MB);`
	`Free(1 MB);`

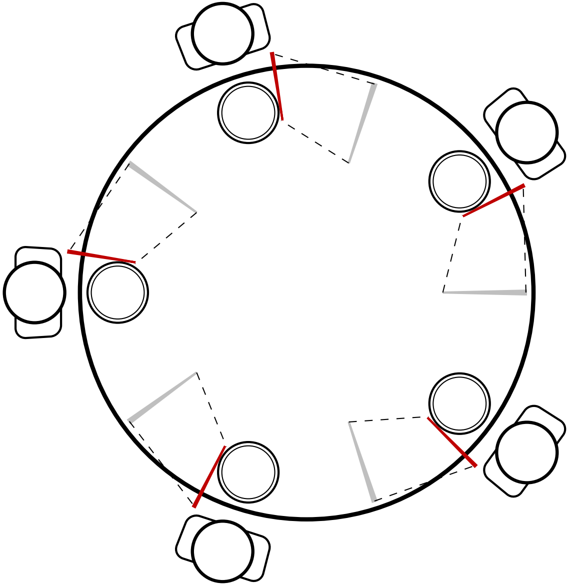
dining philosophers



five philosophers either think or eat

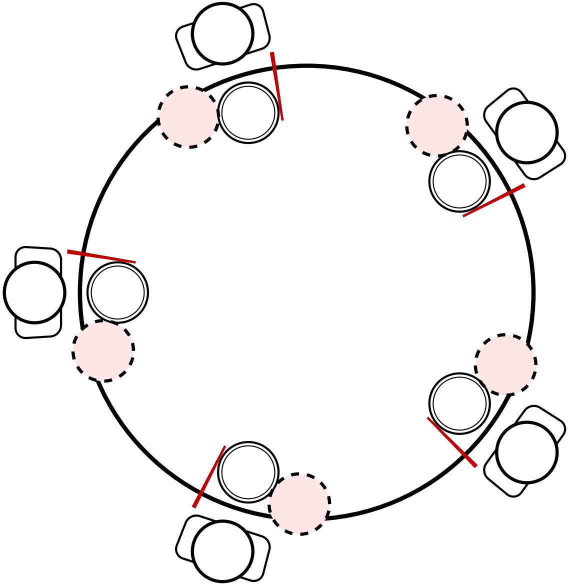
to eat, grab chopsticks on either side

dining philosophers



everyone eats at the same time?
grab left chopstick, then…

dining philosophers



everyone eats at the same time?
grab left chopstick, then
try to grab right chopstick, …
we're at an impasse

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 $\{T_1, \ldots, T_n\}$ 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->p
    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 — no *mutual exclusion*

or at least enough that never run out

no shared resources — no *mutual exclusion*

no waiting — no *hold and wait*/ *preemption*

“busy signal” — abort and (maybe) retry

revoke/preempt resources

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 — no *mutual exclusion*

or at least enough that never run out

memory allocation: malloc() fails rather than waiting (no deadlock)
locks: `pthread_mutex_trylock` fails rather than waiting
...

r ... *exclusion*

no waiting — no *hold and wait*/*preemption*

"busy signal" — abort and (maybe) retry

revoke/preempt resources

acquire resources in **consistent order** — no *circular wait*

request **all resources at once** — no *hold and wait*

deadlock with free space

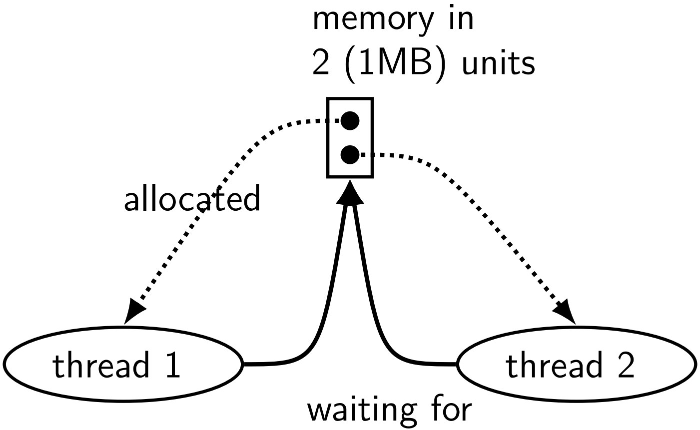
Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

2 MB of space — deadlock possible with unlucky order

deadlock with free space (unlucky case)

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	
	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB`… stalled	
	`AllocateOrWaitFor(1 MB`… stalled

free space: dependency graph



deadlock with free space (lucky case)

Thread 1	Thread 2
`AllocateOrWaitFor(1 MB)`	
`AllocateOrWaitFor(1 MB)`	
`(do calculation)`	
`Free(1 MB);`	
`Free(1 MB);`	
	`AllocateOrWaitFor(1 MB)`
	`AllocateOrWaitFor(1 MB)`
	`(do calculation)`
	`Free(1 MB);`
	`Free(1 MB);`

AllocateOrFail

Thread 1	Thread 2
`AllocateOrFail(1 MB)`	
	`AllocateOrFail(1 MB)`
`AllocateOrFail(1 MB)` fails!	
	`AllocateOrFail(1 MB)` fails!
`Free(1 MB)` (cleanup after failure)	
	`Free(1 MB)` (cleanup after failure)

okay, now what?

give up?

both try again? — maybe this will keep happening? (called livelock)

try one-at-a-time? — gaurenteed to work, but tricky to implement

AllocateOrSteal

Thread 1	Thread 2
`AllocateOrSteal(1 MB)`	
	`AllocateOrSteal(1 MB)`
`AllocateOrSteal(1 MB)`	`Thread killed to free 1MB`
(do work)	

problem: can one actually implement this?

problem: can one kill thread and keep system in consistent state?

fail/steal with locks

pthreads provides pthread_mutex_trylock — "lock or fail"

some databases implement *revocable locks*

- do equivalent of throwing exception in thread to 'steal' lock
- need to carefully arrange for operation to be cleaned up

stealing locks???

how do we make stealing locks possible

unclean: just kill the thread

- problem: inconsistent state?

clean: have code to undo partial oepration

- 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")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	
	`unlock(&B->lock)`
`lock(&A->lock)` → `LOCKED`	
	`lock(&B->lock)` → `LOCKED`
`trylock(&B->lock)` → `FAILED`	
	`trylock(&A->lock)` → `FAILED`
`unlock(&A->lock)`	
	`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?

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 waiti

"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 divislbe 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 divislbe 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?

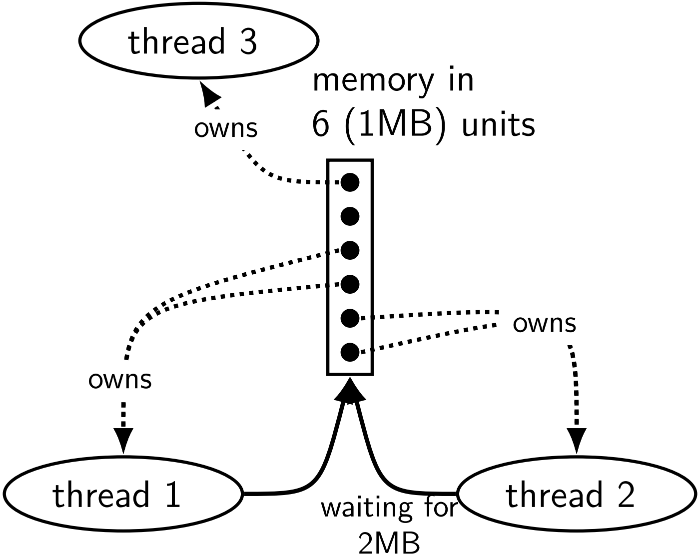
not a deadlock — thread 3 can still finish

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

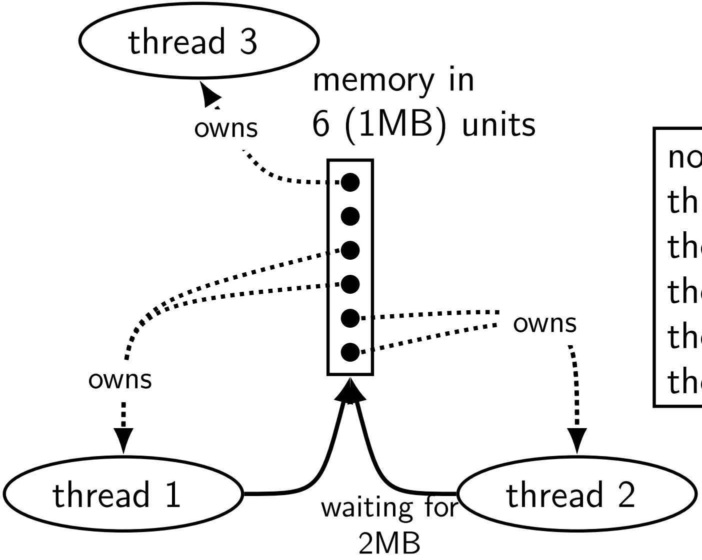
...but would be deadlock

- ...if thread 3 waiting lock held by thread 1
- ...with 5MB of RAM

divisible resources: not deadlock

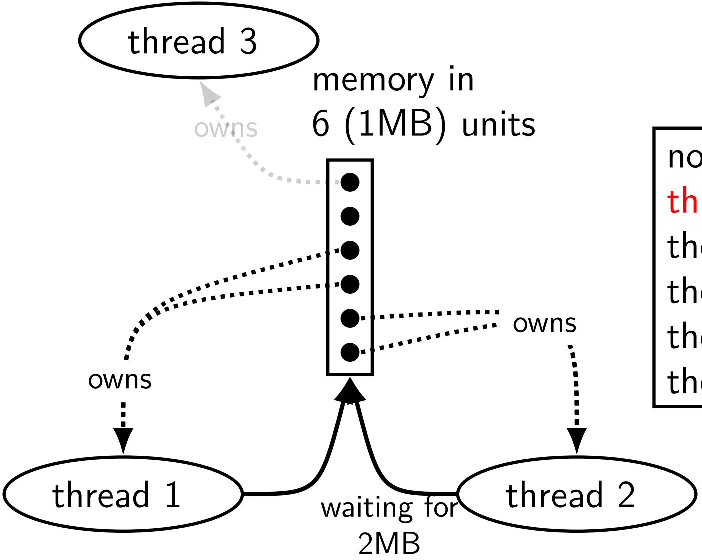


divisible resources: not deadlock



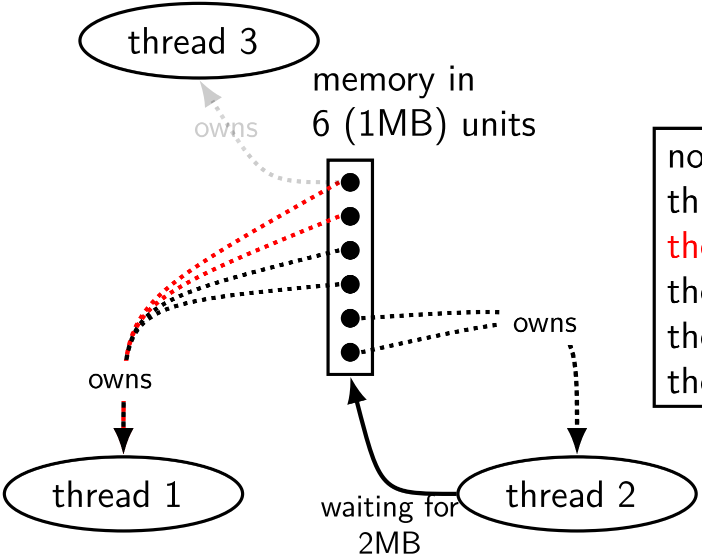
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



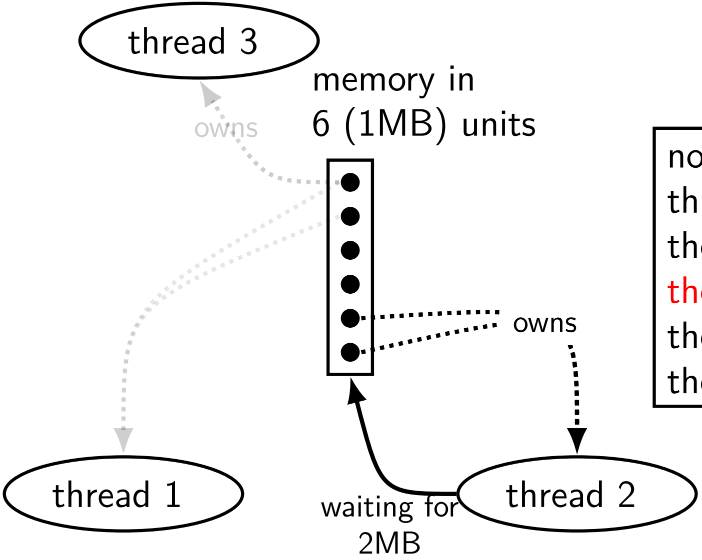
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



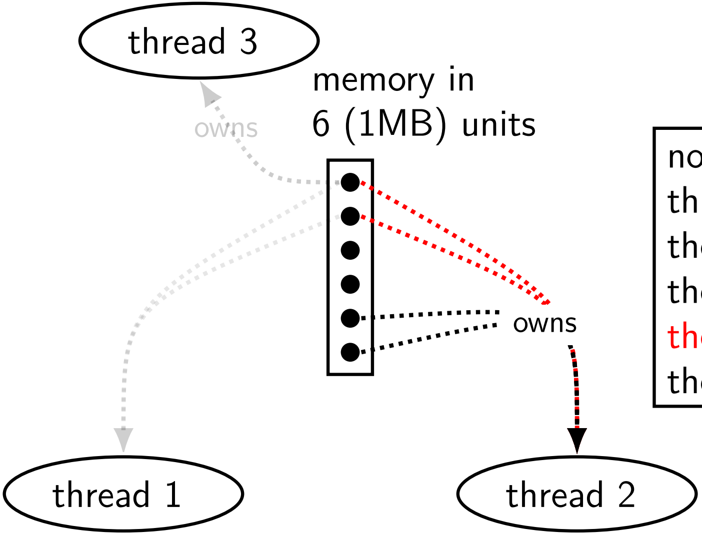
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



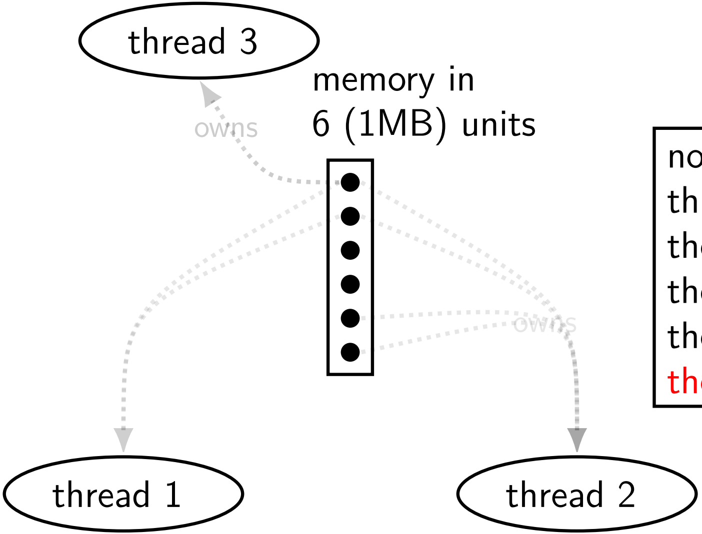
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



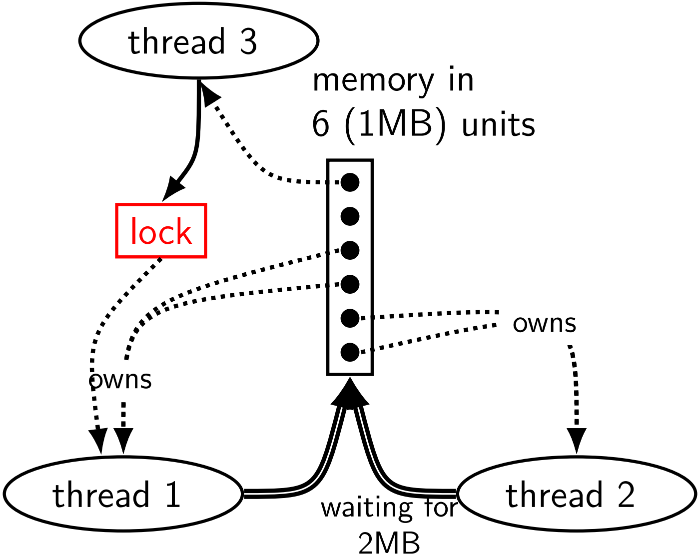
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock

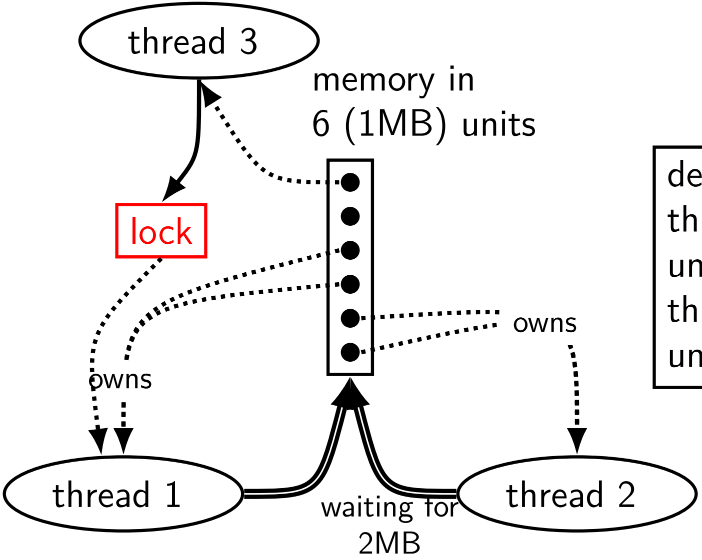


not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: is deadlock

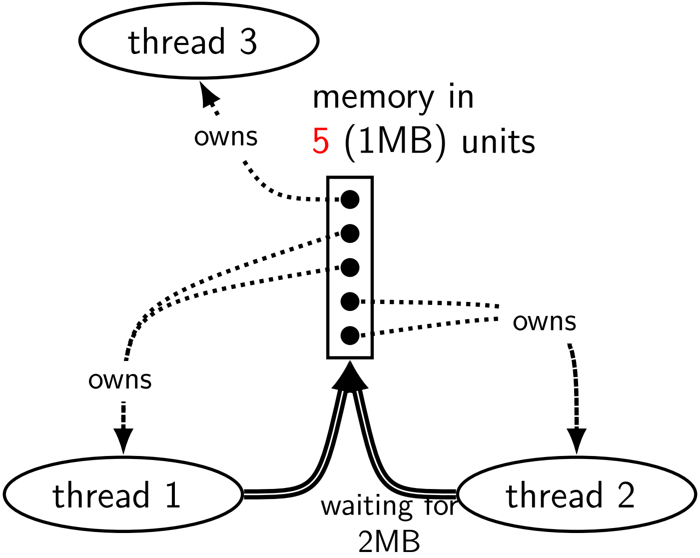


divisible resources: is deadlock

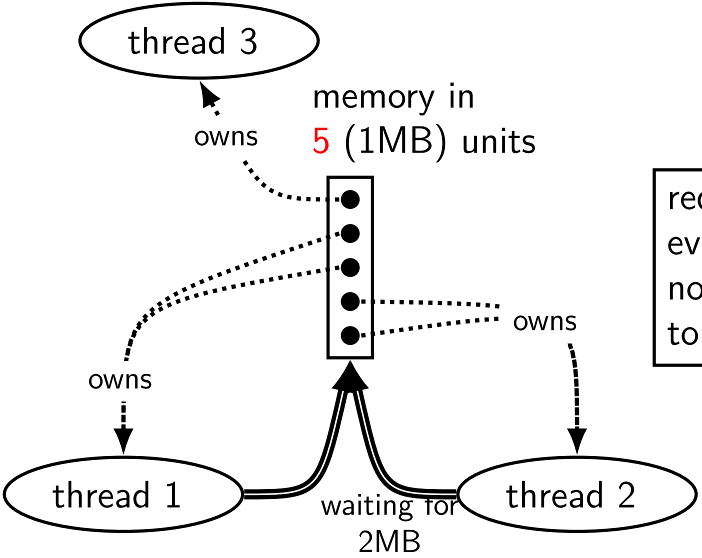


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

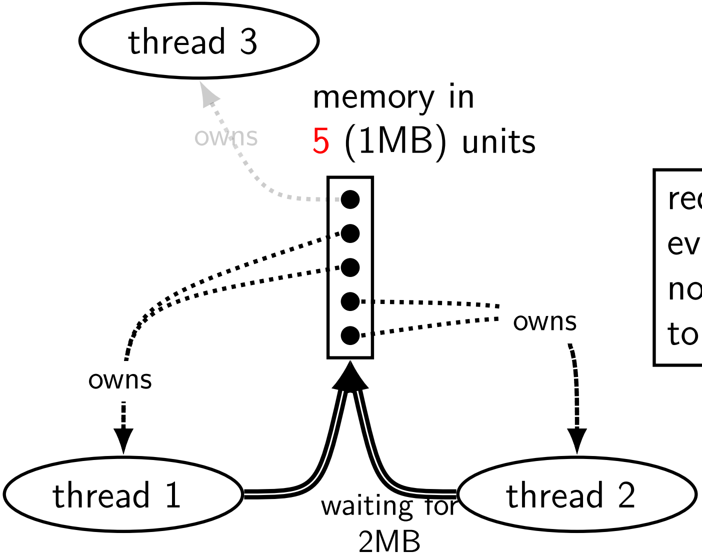


divisible resources: is deadlock



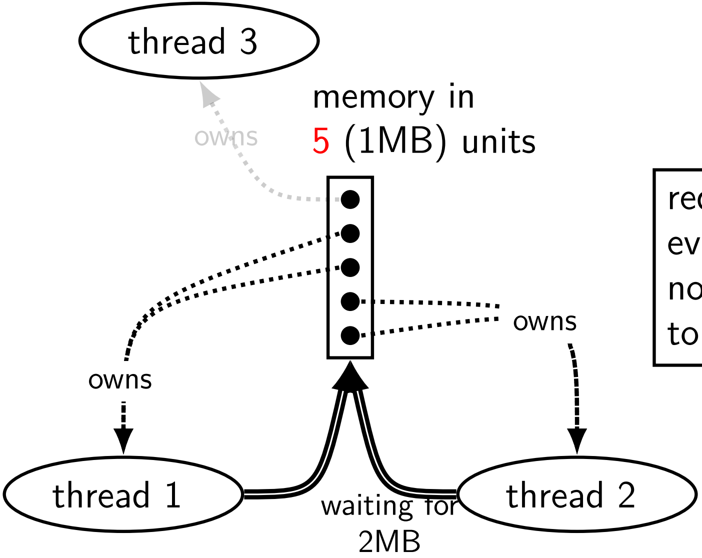
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



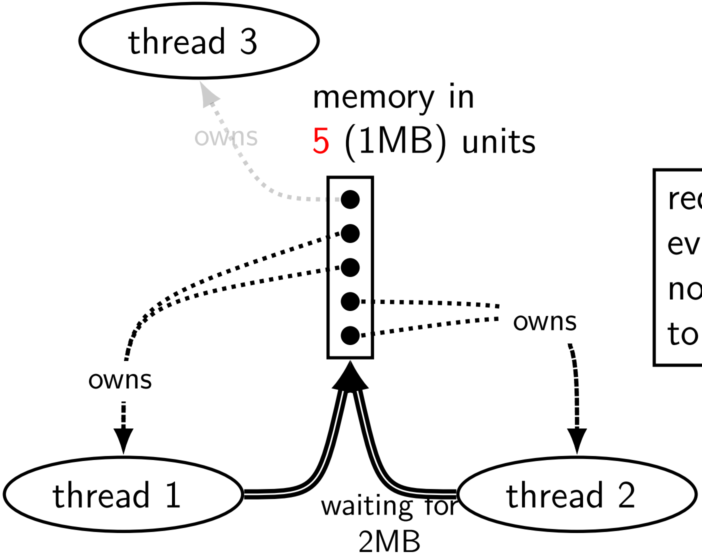
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



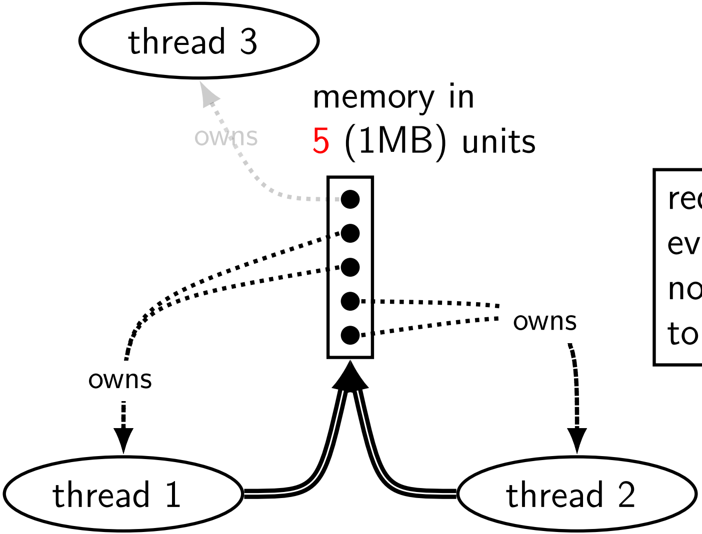
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

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

for each resource: track which threads have those resources

for each thread: resources they are waiting for

repeatedly:

find a thread where all the resources it needs are available
remove that thread and mark the resources it has as free — it can complete now!

either: all threads eliminated *or* found deadlock

aside: deadlock detection in reality

instrument all contended resources?

- add tracking of who locked what
- modify every lock implementation — no simple spinlocks?
- some tricky cases: e.g. what about counting semaphores?

doing something useful on deadlock?

- want way to "undo" partially done operations

...but done for some applications

common example: for locks in a database

- database typically has customized locking code
- "undo" exists as side-effect of code for handling power/disk failures

using deadlock detection for prevention

suppose you know the *maximum resources* a process could request

make decision when starting process ("*admission control*")

using deadlock detection for prevention

suppose you know the *maximum resources* a process could request

make decision when starting process ("*admission control*")

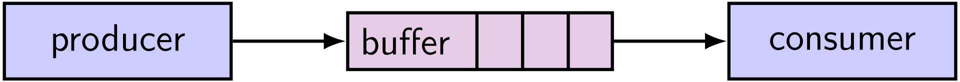
ask "what if every process was waiting for maximum resources"

including the one we're starting

would it cause deadlock? then don't let it start

called Banker's algorithm

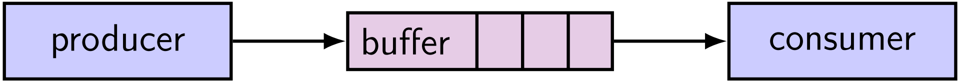
example: producer/consumer



shared buffer (queue) of fixed size

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

example: producer/consumer



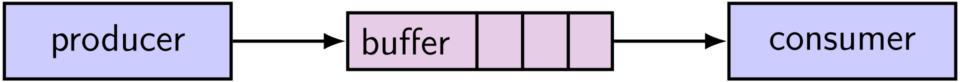
shared buffer (queue) of fixed size

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

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

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

example: producer/consumer



shared buffer (queue) of fixed size

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

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

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

example: C compiler

- preprocessor → compiler → assembler → linker

monitors/condition variables

locks for mutual exclusion

condition variables for waiting for event

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

related data structures

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

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

monitor idea

a monitor

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

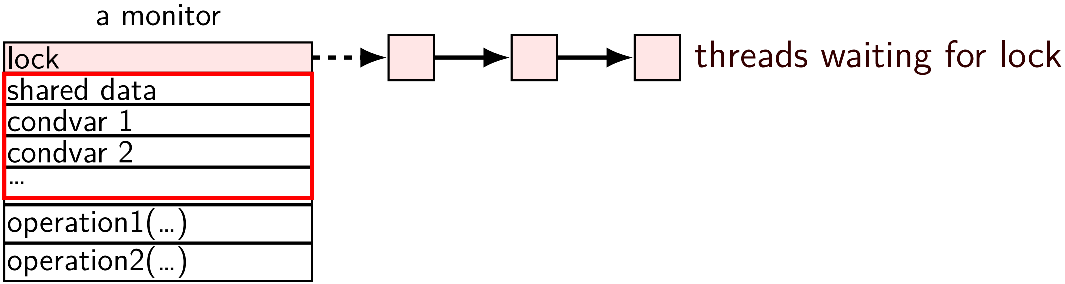
monitor idea

a monitor

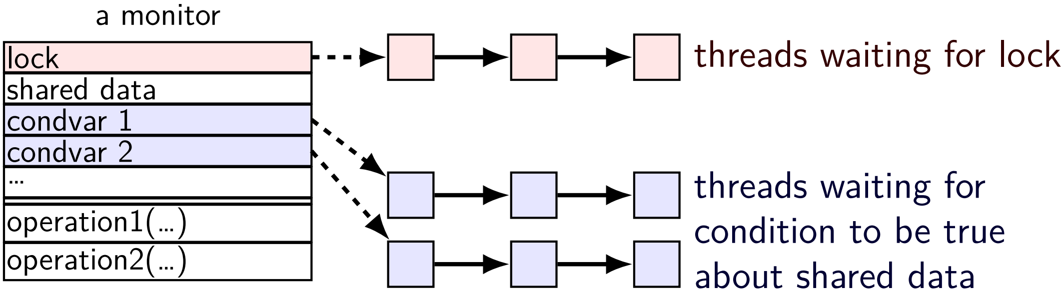
lock
shared data
condvar 1
condvar 2
...
operation1(...)
operation2(...)

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

monitor idea



monitor idea



condvar operations

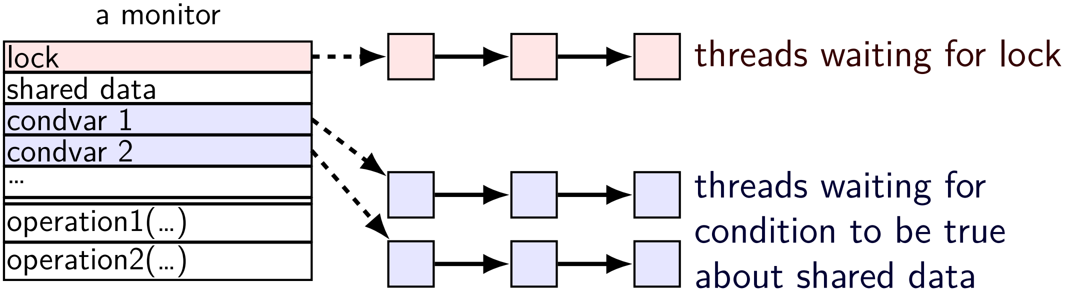
condvar operations:

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

…and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

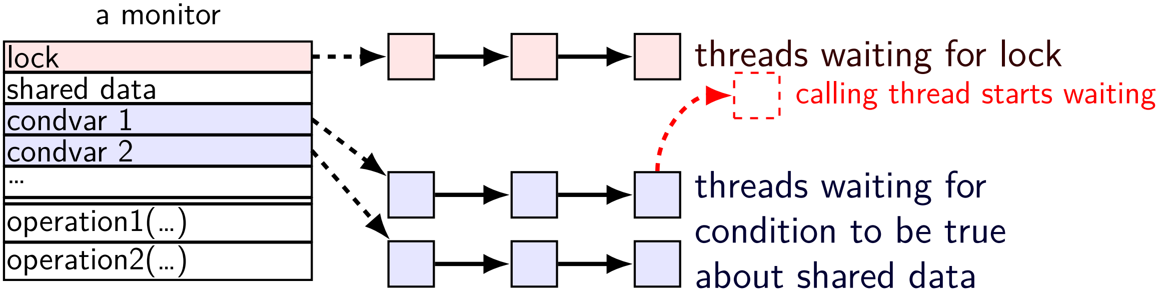
condvar operations:

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

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

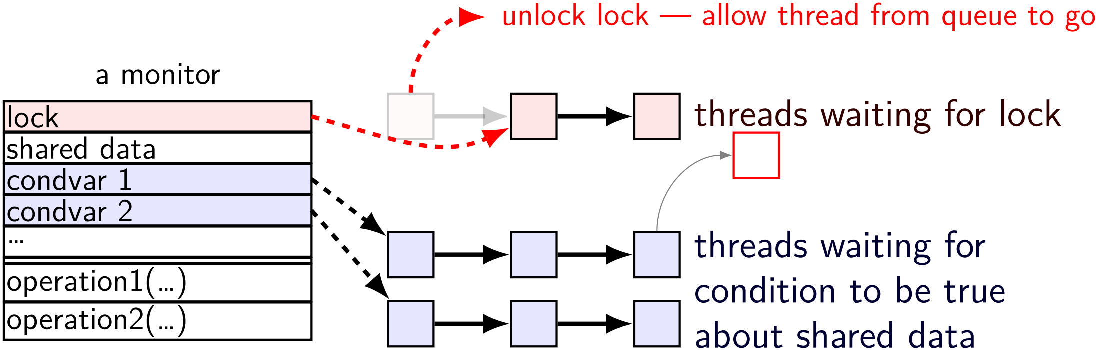
condvar operations:

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

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

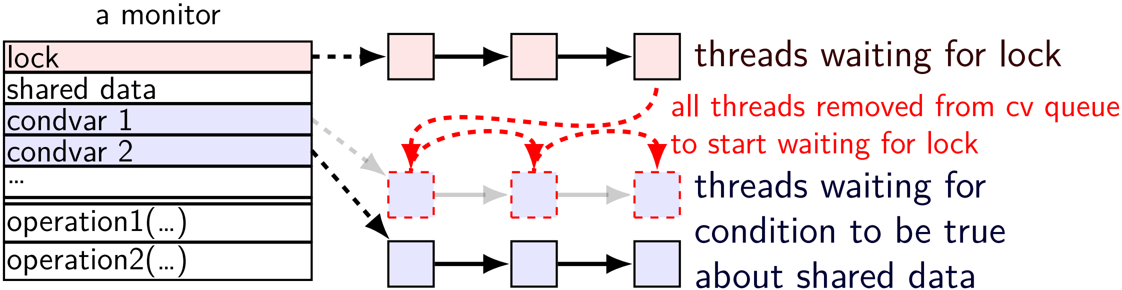
condvar operations:

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

…and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



condvar operations

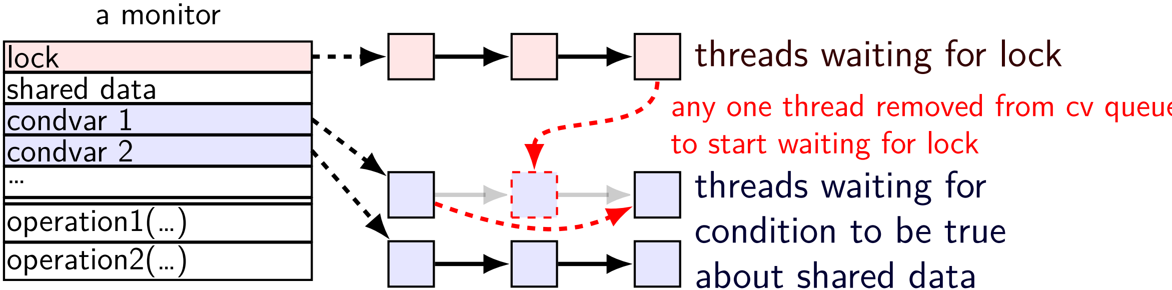
condvar operations:

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

...and reacquire lock before returning

Broadcast(cv) — remove all from condvar queue

Signal(cv) — remove one from condvar queue



pthread cv usage

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

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

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

pthread cv usage

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

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

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

acquire lock before reading or writing `finished`

pthread cv usage

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

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

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

check whether we need to wait at all

(why a loop? we'll explain later)

pthread cv usage

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

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

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

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

pthread cv usage

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

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

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

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

WaitForFinish timeline 1

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

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

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

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

WaitForFinish timeline 2

WaitForFinish thread	Finish thread

	`mutex_lock(&lock)`
	`finished = true`
	`cond_broadcast(&finished_cv)`
	`mutex_unlock(&lock)`

`mutex_lock(&lock)`
`while (!finished) ...`
(finished now true, so return)
`mutex_unlock(&lock)`	

why the loop

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

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

why the loop

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

we only `broadcast` if `finished` is true

so why check `finished` afterwards?

pthread_cond_wait manual page:

> "Spurious wakeups ... may occur."

spurious wakeup = `wait` returns even though nothing happened

unbounded buffer producer/consumer

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

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

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

unbounded buffer producer/consumer

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

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

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

rule: never touch `buffer` without acquiring lock

otherwise: what if two threads simulatenously en/dequeue?

(both use same array/linked list entry?)

(both reallocate array?)

unbounded buffer producer/consumer

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

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

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

check if empty
if so, dequeue

okay because have lock

other threads can**not** dequeue here

unbounded buffer producer/consumer

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

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

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

wake one Consume thread
if any are waiting

unbounded buffer producer/consumer

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

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

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

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

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

unbounded buffer producer/consumer

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

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

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

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

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

unbounded buffer producer/consumer

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

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

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

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

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

unbounded buffer producer/consumer

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

in pthreads: signalled thread not gaurenteed to hold lock next

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

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

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

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

Hoare versus Mesa monitors

Hoare-style monitors

signal 'hands off' lock to awoken thread

Mesa-style monitors

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

every current threading library I know of does Mesa-style

bounded buffer producer/consumer

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

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

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

bounded buffer producer/consumer

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

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

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

bounded buffer producer/consumer

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

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

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

correct (but slow?) to replace with:

`pthread_cond_broadcast(&space_ready);`

(just more "spurious wakeups")

bounded buffer producer/consumer

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

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

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

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

monitor pattern

```
pthread_mutex_lock(&lock);
while (!condition A) {
    pthread_cond_wait(&condvar_for_A, &lock);
}
... /* manipulate shared data, changing other conditions */
if (set condition 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_fin

void WaitForBothFinishe
  pthread_mutex_lock(&l
  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_singal(&both_finished_cv);
```

D.
```
if (finished[1-index])
      pthread_cond_broadcast(&both_finished_cv);
```

E. something else

monitor exercise: barrier

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

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

void BarrierWait(BarrierInfo *b) {
    pthread_mutex_lock(&b->lock);
    ++b->number_reached;
    if (b->number_reached == b->total_threads) {
        _____________________
    } else {
        _____________________
    }
    pthread_mutex_unlock(&b->lock);
}
```

monitor exercise: ConsumeTwo

suppose we want producer/consumer, but...

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

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

what should we change below?

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

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

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

monitor exercise: ordering

suppose we want producer/consumer, but...

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

(no matter what ordering cond_signal/cond_broadcast use)

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

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

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

producer/consumer signal?

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

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    /* GOOD CODE: pthread_cond_signal(&data_ready); */
    /* BAD CODE: */
    if (buffer.size() == 1)
        pthread_cond_signal(&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
		enqueue	
wait for lock		size $=$ 1? signal	
		unlock	gets lock
			enqueue
			size $\neq$ 1: don't signal
			unlock
gets lock			
dequeue			
	still waiting		

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 $\boxed{3}$

counting resources: reserving books

suppose tracking copies of same library book

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

up = give back book; down = take book

taken out ~~Copy 1~~

Copy 2

Copy 3

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

after calling down to reserve

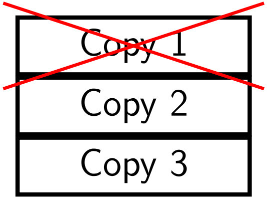
counting resources: reserving books

suppose tracking copies of same library book

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

up = give back book; down = take book

taken out

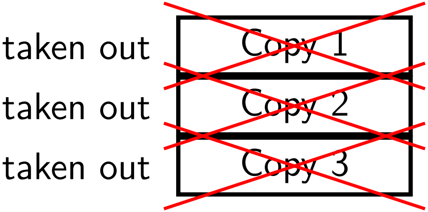


free copies $\boxed{2}$

after calling down to reserve

counting resources: reserving books

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

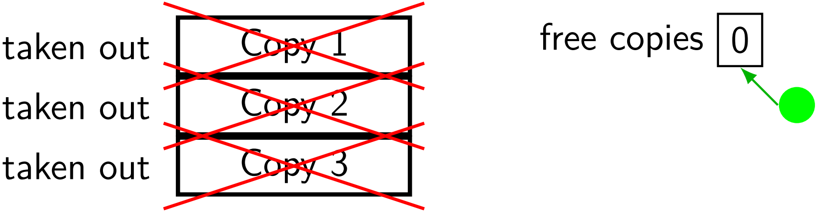


free copies $\boxed{0}$

after calling down three times
to reserve all copies

counting resources: reserving books

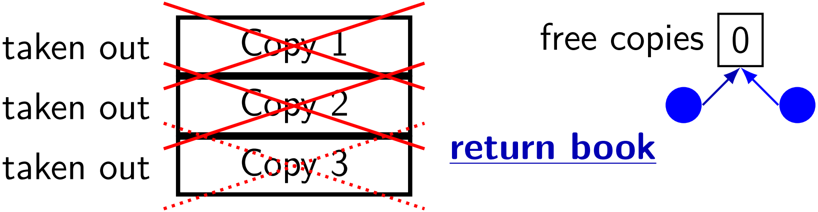
suppose tracking copies of same library book
non-negative integer count = # how many books used?
up = give back book; down = take book



reserve book
call *down* again
start waiting...

counting resources: reserving books

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



return book

call *up*
release waiter

reserve book

call *down*
~~waiting~~
done waiting

implementing mutexes with semaphores

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

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

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

implementing join with semaphores

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

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

POSIX semaphores

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

semaphore 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().

PutValue() waits for prior GetValue(), places value, then allows next GetValue().

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);                                 re 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);                               re 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*)

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                          s 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!

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

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
}
```

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” =

locking to make sure no other operations interfere (consistency)

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

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 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

implementing durability

what if there's a crash

we might have written some things but not others

most common approach: write-ahead logging

write list of intended operations + marker that list is complete

then do operations

after crash: check for intended operations

redo them *only if list is complete*

backup slides

backup slides

recall: pthread mutex

```
#include <pthread.h>

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

life homework even/odd

naive way has an operation that needs locking:

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

but this alternative needs less locking:

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

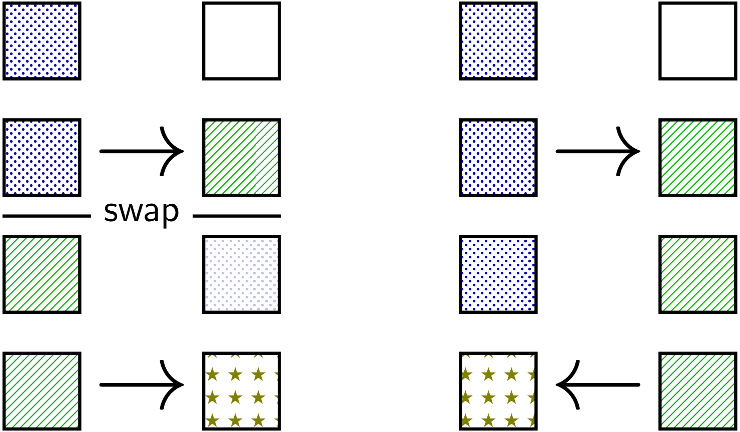
life homework even/odd

naive way has an operation that needs locking:

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

but this alternative needs less locking:

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


swap

x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

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

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

set lock variable to 1 (taken)

read old value

x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

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

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

x86-64 spinlock with xchg

lock variable in shared memory: `the_lock`

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

```
acquire:
    movl $1, %eax               // %eax <- 1
    lock xchg %eax, the_lock    // swap %eax and the_lock
                                   // sets the_lock to 1 (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`
B. `mov $1, finished; mfence`
C. `mov $0, %eax`
D. `mov $1, %eax`
E. je
F. jne

spinlock problems

lock abstraction is not powerful enough

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

spinlocks waste CPU time more than needed

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

spinlocks can send a lot of messages on the shared bus

- more efficient atomic operations to implement locks

spinlock problems

lock abstraction is not powerful enough

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

spinlocks waste CPU time more than needed

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

spinlocks can send a lot of messages on the shared bus

- more efficient atomic operations to implement locks

mutexes: intelligent waiting

want: locks that wait better

- example: POSIX mutexes

instead of running infinite loop, give away CPU

lock = go to sleep, add self to list

- sleep = scheduler runs something else

unlock = wake up sleeping thread

mutexes: intelligent waiting

want: locks that wait better

- example: POSIX mutexes

instead of running infinite loop, give away CPU

lock = go to sleep, add self to list

- sleep = scheduler runs something else

unlock = wake up sleeping thread

better lock implementation idea

shared list of waiters

spinlock protects list of waiters from concurrent modification

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

unlock = use spinlock to remove item from list

better lock implementation idea

shared list of waiters

spinlock protects list of waiters from concurrent modification

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

unlock = use spinlock to remove item from list

one possible implementation

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

one possible implementation

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

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

one possible implementation

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

tracks whether any thread has locked and not unlocked

one possible implementation

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

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

one possible implementation

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

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    mark current thread as waiting
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler (context switch)
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    mark thread as no longer waiting
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
    m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

one possible implementation

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

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

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    mark current thread as waiting
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler (context switch)
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    mark thread as no longer waiting
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
     m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

one possible implementation

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

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

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    mark current thread as waiting
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler (context switch)
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    mark thread as no longer waiting
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
    m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

one possible implementation

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

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    mark current thread as waiting
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler (context switch)
  } else {
    m->lock_taken = true;
    UnlockSpinlock(&m->guard_spinlock);
  }
}
```

```
UnlockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->wait_queue not empty) {
    remove a thread from m->wait_queue
    mark thread as no longer waiting
    /* xv6: myproc()->state = RUNNABLE; */
  } else {
    m->lock_taken = false;
  }
  UnlockSpinlock(&m->guard_spinlock);
}
```

mutex and scheduler subtly

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

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

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

mutex and scheduler subtly

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

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

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

mutex efficiency

'normal' mutex **uncontended** case:

- lock: acquire + release spinlock, see lock is free
- unlock: acquire + release spinlock, see queue is empty

not much slower than spinlock

implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

implementing locks: single core

intuition: context switch only happens on interrupt

- timer expiration, I/O, etc. causes OS to run

solution: disable them

- reenable on unlock

x86 instructions:

- `cli` — disable interrupts
- `sti` — enable interrupts

naive interrupt enable/disable (1)

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

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

naive interrupt enable/disable (1)

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

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

problem: user can hang the system:

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

naive interrupt enable/disable (1)

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

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

problem: user can hang the system:

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

problem: can't do I/O within lock

```
Lock(some_lock);
read from disk
    /* waits forever for (disabled) interrupt
       from disk IO finishing */
```

naive interrupt enable/disable (2)

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

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

naive interrupt enable/disable (2)

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

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

naive interrupt enable/disable (2)

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

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

naive interrupt enable/disable (2)

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

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

problem: nested locks

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

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

GCC: preventing reordering example (1)

```
void Alice() {
    int one = 1;
    __atomic_store(&note_from_alice, &one, __ATOMIC_SEQ_CST);
    do {
    } while (__atomic_load_n(&note_from_bob, __ATOMIC_SEQ_CST));
    if (no_milk) {++milk;}
}
```

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

GCC: preventing reordering example (2)

```
void Alice() {
    note_from_alice = 1;
    do {
        __atomic_thread_fence(__ATOMIC_SEQ_CST);
    } while (note_from_bob);
    if (no_milk) {++milk;}
}
```

```
Alice:
  movl $1, note_from_alice  // note_from_alice <- 1
.L3:
  mfence  // make sure store is visible to other cores before
          // on x86: not needed on second+ iteration of loop
  cmpl $0, note_from_bob  // if (note_from_bob == 0) repeat fe
  jne .L3
  cmpl $0, no_milk
  ...
```

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

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

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

solution

xv6 spinlock: acquire

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

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

  // 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.
  __
  ..
}
```

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_synchronize();
  ...
}
```

xchg wraps the lock xchg instruction
same loop as before

xv6 spinlock: acquire

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

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

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

xv6 spinlock: release

```
void
release(struct spinlock *lk)
  ...
  // Tell the C compiler and the processor to not move loads or 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 be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli
}
```

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

xv6 spinlock: release

```
void
release(struct spinlock *lk)
  ...
  // Tell the C compiler and the processor to not move loads or 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 releasea
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

reenable interrupts (taking nested locks into account)

fetch-and-add with CAS (1)

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

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

implementation sketch:

- fetch value from pointer `old`
- compute in temporary value result of addition `new`
- try to change value at pointer from `old` to `new` [`compare-and-swap`]
- if not successful, repeat

fetch-and-add with CAS (2)

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

exercise: append to singly-linked list

ListNode is a singly-linked list

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

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

- atomically change `*pointer` from `old` to `new`
- return true if successful
- return false (and change nothing) if `*pointer` is not `old`

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

some common atomic operations (1)

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

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

some common atomic operations (2)

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

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

common atomic operation pattern

try to do operation, ...

detect if it failed

if so, repeat

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

cache coherency states

extra information for each cache block

- overlaps with/replaces valid, dirty bits

stored in each cache

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

different caches may have different states for same block

MSI state summary

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

Shared value is the same as memory

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

MSI scheme

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

blue: transition requires sending message on bus

MSI scheme

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

blue: transition requires sending message on bus

example: write while Shared

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

MSI scheme

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

blue: transition requires sending message on bus

example: write while Shared

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

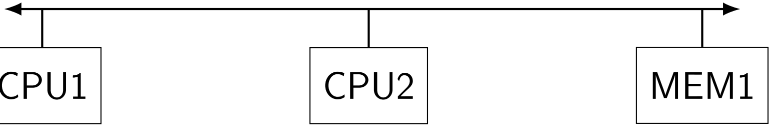
example: hear write while Shared

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

example: write while Modified

- nothing to do — no other CPU can have a copy

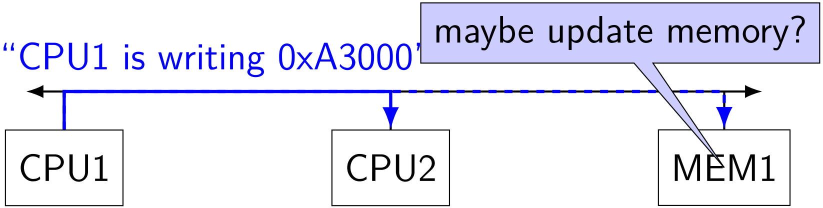
MSI example



address	value	state
`0xA300`	`100`	Shared
`0xC400`	`200`	Shared
`0xE500`	`300`	Shared

address	value	state
`0x9300`	`172`	Shared
`0xA300`	`100`	Shared
`0xC500`	`200`	Shared

MSI example



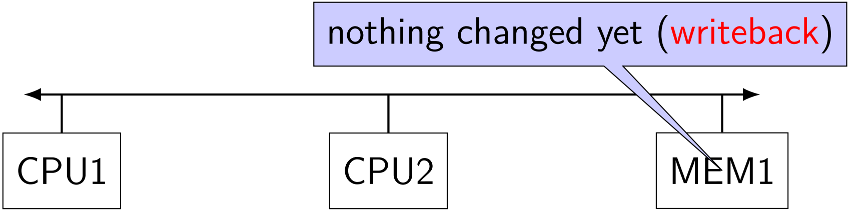
address	value	state
0xA300	~~100~~101	Modified
0xC400	200	Shared
0xE500	300	Shared

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

cache sees write:
invalidate 0xA300

CPU1 writes 101 to 0xA300

MSI example



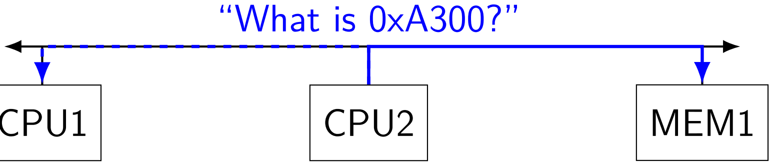
address	value	state
0xA300	~~101~~102	Modified
0xC400	200	Shared
0x		

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

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

CPU1 writes 102 to 0xA300

MSI example



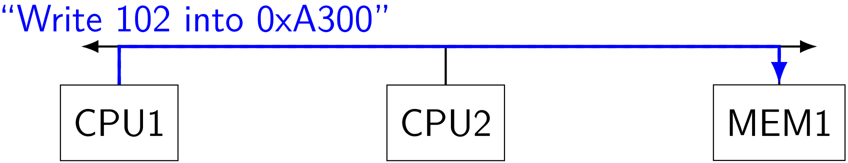
address	value	state
`0xA300`	`102`	Modified
`0xC400`	`200`	Shared
`0`		

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

modified state — must update for CPU2!

CPU2 reads 0xA300

MSI example



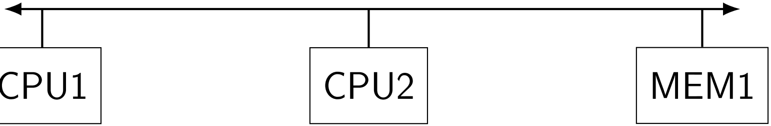
address	value	state
0xA300	102	Shared
0xC400	200	Shared
0xE		

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

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

CPU2 reads 0xA300

MSI example



address	value	state
`0xA300`	`102`	Shared
`0xC400`	`200`	Shared
`0xE500`	`300`	Shared

address	value	state
`0x9300`	`172`	Shared
~~`0xA300`~~	~~`100`~~`102`	Shared
`0xC500`	`200`	Shared

MSI: update memory

to write value (enter modified state), need to invalidate others

can avoid sending actual value (shorter message/faster)

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

MSI: on cache replacement/writeback

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

changes state to invalid

requires writeback if modified (= dirty bit)

cache coherency exercise

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

- CPU 1: read 0x1000
- CPU 2: read 0x1000
- CPU 1: write 0x1000
- CPU 1: read 0x2000
- CPU 2: read 0x1000
- CPU 2: write 0x2008
- CPU 3: read 0x1008

Q1: final state of 0x1000 in caches?

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

CPU 1: CPU 2: CPU 3:

Q2: final state of 0x2000 in caches?

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

CPU 1: CPU 2: CPU 3:

why load/store reordering?

fast processor designs can execute instructions out of order

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

more on this later in the semester

C++: preventing reordering

to help implementing things like pthread_mutex_lock

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

prevent CPU reordering *and* prevent compiler reordering

also provide other tools for implementing locks (more later)

could also hand-write assembly code

- compiler can't know what assembly code is doing

C++: preventing reordering example

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

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

C++ atomics: no reordering

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

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

GCC: built-in atomic functions

used to implement std::atomic, etc.

predate std::atomic

builtin functions starting with `__sync` and `__atomic`

these are what xv6 uses

aside: some x86 reordering rules

each core sees its own loads/stores in order

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

stores *from other cores* appear in a consistent order

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

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

Source: Intel 64 and IA-32 Software Developer's Manual, Volume 3A, Chapter 8

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

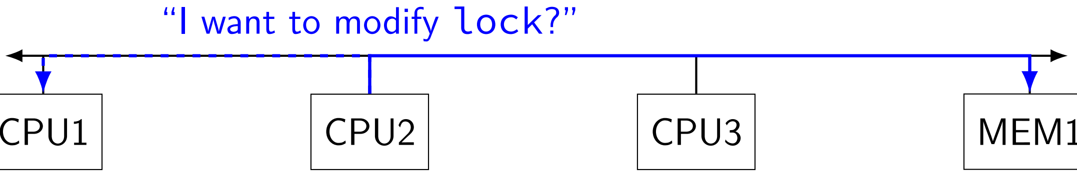
ping-ponging

address	value	state
`lock`	`locked`	Modified

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

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

ping-ponging



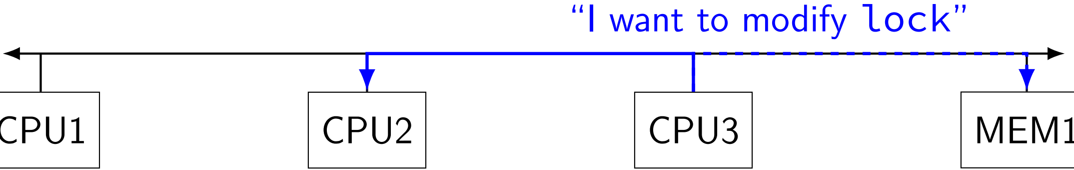
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

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

ping-ponging



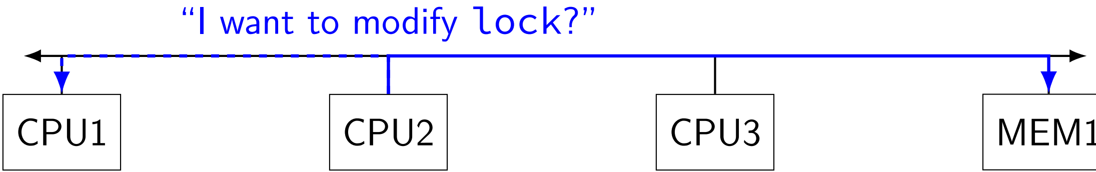
address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

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

ping-ponging



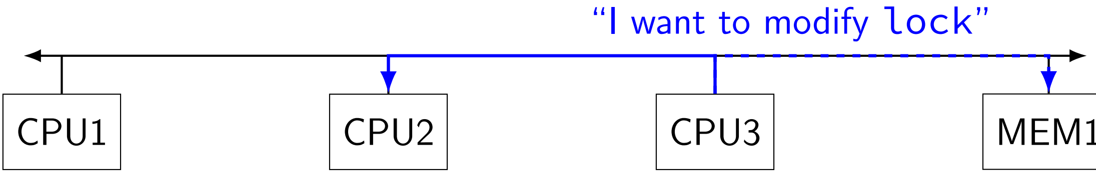
address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

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

ping-ponging



address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

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

ping-ponging

“I want to modify `lock`”

address	**value**	**state**
`lock`	`unlocked`	Modified

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

address	**value**	**state**
`lock`		Invalid

CPU1 sets lock to unlocked

ping-ponging

"I want to modify `lock`"

address	value	state
lock	---	Invalid

address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

some CPU (this example: CPU2) acquires lock

ping-ponging

test-and-set problem: cache block "ping-pongs" between caches

- each waiting processor reserves block to modify
- could maybe wait until it determines modification needed — but not typical implementation

each transfer of block sends messages on bus

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

- like what the processor with the lock is doing

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

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

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

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

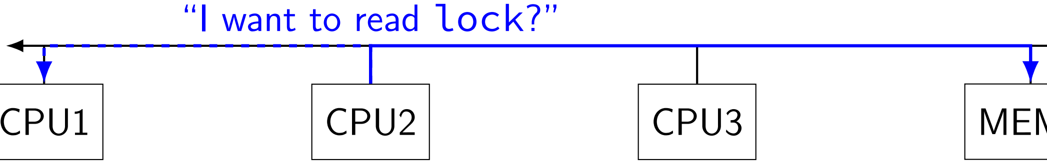
less ping-ponging

address	value	state
lock	locked	Modified

address	value	state
lock	---	Invalid

address	value	state
lock	---	Invalid

less ping-ponging



address	value	state
lock	locked	Modified

address	value	state
lock		Invalid

address	value	state
lock		Invalid

CPU2 reads lock
(to see it is still locked)

less ping-ponging

“set lock to locked”

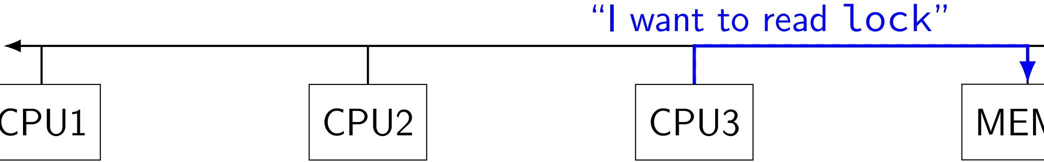
address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock		Invalid

CPU1 writes back lock value, then CPU2 reads it

less ping-ponging



address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

CPU3 reads lock
(to see it is still locked)

less ping-ponging

address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

address	value	state
lock	locked	Shared

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

less ping-ponging

“I want to modify `lock`”

address	value	state
`lock`	`unlocked`	Modified

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

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

CPU1 sets lock to unlocked

less ping-ponging

"I want to modify `lock`"

address	value	state
`lock`		Modified

address	value	state
`lock`		Invalid

address	value	state
`lock`		Invalid

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

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

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

and keep it in the shared state

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

more room for improvement?

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

there other spinlock designs that avoid this

- ticket locks
- MCS locks
- ...

MSI extensions

real cache coherency protocols sometimes more complex:

separate tracking modifications from whether other caches have copy

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

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

too much milk

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

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

how can Alice and Bob coordinate better?

too much milk “solution” 1 (algorithm)

leave a note: “I am buying milk”

place before buying, remove after buying

don’t try buying if there’s a note

≈ setting/checking a variable (e.g. “`note = 1`”)

with atomic load/store of variable

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

too much milk "solution" 1 (algorithm)

leave a note: "I am buying milk"

place before buying, remove after buying

don't try buying if there's a note

≈ setting/checking a variable (e.g. "`note = 1`")

with atomic load/store of variable

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

exercise: why doesn't this work?

too much milk “solution” 1 (timeline)

Alice

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

Bob

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

too much milk “solution” 2 (algorithm)

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

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

too much milk: "solution" 2 (timeline)

Alice

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

too much milk: “solution” 2 (timeline)

Alice

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

but there’s always a note

too much milk: “solution” 2 (timeline)

Alice

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

but there’s always a note

…will never buy milk (twice *or* once)

“solution” 3: algorithm

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

computer equivalent: separate noteFromAlice and noteFromBob variables

Alice

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

Bob

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

too much milk: “solution” 3 (timeline)

Alice	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`
`remove note from Alice`	

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

```
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

```
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

```
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

cache coherency works on cache blocks

but typical memory access — less than cache block

- e.g. one 4-byte array element in 64-byte cache block

what if two processors modify different parts same cache block?

- 4-byte writes to 64-byte cache block

cache coherency — write instructions happen one at a time:

- processor ‘locks’ 64-byte cache block, fetching latest version
- processor updates 4 bytes of 64-byte cache block
- later, processor might give up cache block

modifying things in parallel (code)

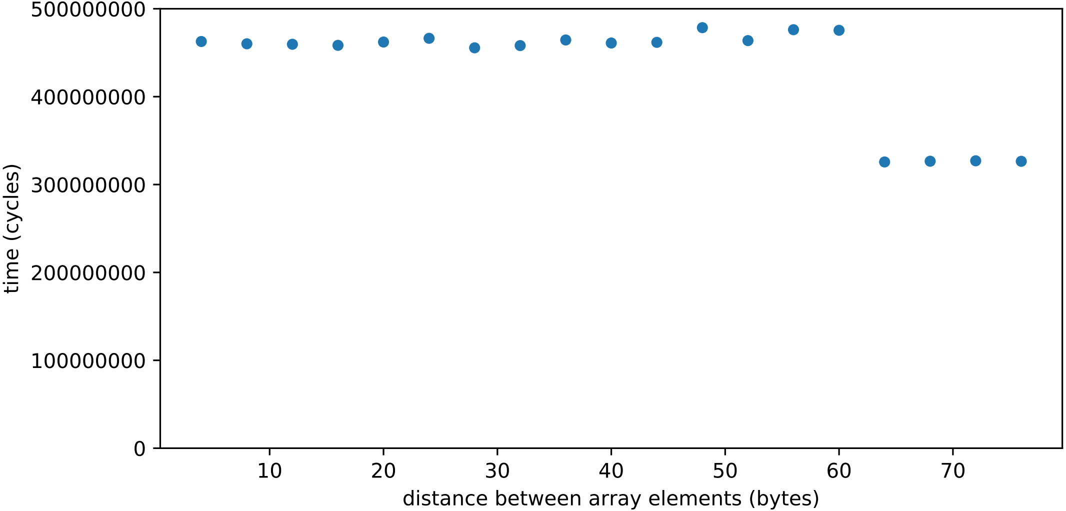
```
void *sum_up(void *raw_dest) {
    int *dest = (int *) raw_dest;
    for (int i = 0; i < 64 * 1024 * 1024; ++i) {
        *dest += data[i];
    }
}

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

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

performance v. array element gap

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



false sharing

synchronizing to access two independent things

two parts of same cache block

solution: separate them

exercise (1)

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

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

exercise (2)

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

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

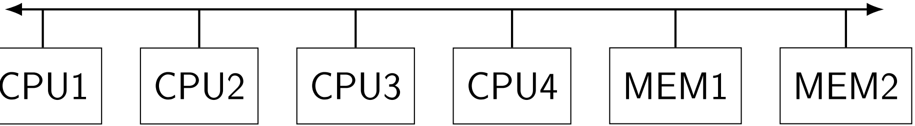
Where is false sharing likely to occur?

connecting CPUs and memory

multiple processors, common memory

how do processors communicate with memory?

shared bus



one possible design

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

tagged messages — everyone gets everything, filters

contention if multiple communicators

- some hardware enforces only one at a time

shared buses and scaling

shared buses perform poorly with “too many” CPUs

so, there are other designs

we’ll gloss over these for now

shared buses and caches

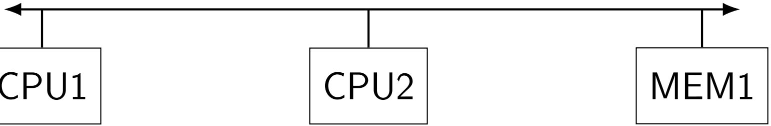
remember caches?

memory is pretty slow

each CPU wants to keep local copies of memory

what happens when multiple CPUs cache same memory?

the cache coherency problem



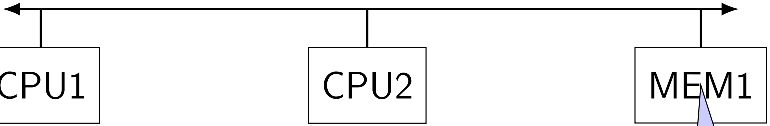
address	value
`0xA300`	`100`
`0xC400`	`200`
`0xE500`	`300`

CPU1's cache

address	value
`0x9300`	`172`
`0xA300`	`100`
`0xC500`	`200`

CPU2's cache

the cache coherency problem



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

CPU1's cache

address	value
0x9300	172
0xA300	100
0xC500	200

When does this change?

When does this change?

CPU1 writes 101 to 0xA300?

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) {
  sem_wait(&private_lock);
  ++num_waiters;
  sem_post(&private_lock);
  lock.Unlock();
  sem_wait(&threads_to_wakeup);
  lock.Lock();
}
```

```
Signal() {
  sem_wait(&private_lock);
  if (num_waiters > 0) {
    sem_post(&threads_to_wakeup);
    --num_waiters;
  }
  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

- semaphore: wait for count to become positive (for down)

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

- semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

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

- semaphore: wait for count to become positive (for down)

wait using condvar; broadcast/signal when condition changes

counting semaphores with binary semaphores

via Hemmendinger, "Comments on 'A correect and unrestrictive implementation of general semaphores' " (1989); Barz, "Implementing semaphores by binary semaphores" (1983)

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

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

```
void Up() {
  mutex.Down();
  value += 1;
  if (value == 1) {
    gate.Up();
    // because down should finish now
    // but could not before
  }
  mutex.Up();
}
```

backup slides