semaphores / rwlocks / deadlock

last time (1)

monitor = mutex + condition variable pattern
- lock for all accesses to shared data
- while (condition based on shared data) wait
- after changing condition broadcast (or signal)
- picky about who goes first: more specific conditions

Mesa-style monitors (vs Hoare-style)
- signal/broadcasted threads might not get lock next
- easier to implement (and most common)
- usually need while loop even w/o spurious wakeups

producer/consumer with monitors
- condition = buffer empty/full
- could signal instead of broadcast b/c only one thread

last time (2)

- counting semaphores
  - hold one non-negative integer
  - down/wait: decrement that integer (waiting for it to be positive first)
  - up/post: increment that integer (possibly wake up thread)
- semaphore intuition
  - library book example: semaphore tracks amount of resource that's free
  - down/wait: reserve one of them, waiting if needed
  - up/post: put back one
- implementing locks with semaphores

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 exercise [solution]

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

semaphore intuition

What do you need to wait for?

- critical section to be finished
- queue to be non-empty
- array to have space for new items

what can you count that will be 0 when you need to wait?

- # of threads that can start critical section now
- # of threads that can join another thread without waiting
- # of items in queue
- # of empty spaces in array

use up/down operations to maintain count

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

producer/consumer constraints

consumer waits for producer(s) if buffer is empty

producer waits for consumer(s) if buffer is full

any thread waits while a thread is manipulating the buffer

one semaphore per constraint:

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

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

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

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

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

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);  // tell consumers there is more data
}

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

producer/consumer pseudocode

```
sem_init(&full_slots, ..., 0 /* # buffer slots initially used */);
sem_init(&empty_slots, ..., BUFFER_CAPACITY);
sem_init(&mutex, ..., 1 /* # thread that can use buffer at once */);
buffer.set_size(BUFFER_CAPACITY);
...

Produce(item) {
    sem_wait(&empty_slots);  // wait until free slot, reserve it
    sem_wait(&mutex);
    buffer.enqueue(item);
    sem_post(&mutex);
    sem_post(&full_slots);                                 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
}
```

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)

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access from multiple threads is safe

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access from multiple threads is safe

could use lock — but doesn't allow multiple readers

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until no readers and no writers
- write unlock: stop being registered as writer

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until no readers and no writers
- write unlock: stop being registered as writer

pthread rwlocks

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL /* attributes */);
...
    pthread_rwlock_rdlock(&rwlock);
    ...  /* read shared data */
    pthread_rwlock_unlock(&rwlock);

    pthread_rwlock_wrlock(&rwlock);
    ...  /* read+write shared data */
    pthread_rwlock_unlock(&rwlock);

...
pthread_rwlock_destroy(&rwlock);
```

rwlock effects exercise

```
pthread_rwlock_t lock;
void ThreadA() {
  pthread_rwlock_rdlock(&lock);
  puts("a");
  ...
  puts("A");
  pthread_rwlock_unlock(&lock);
}
void ThreadB() {
  pthread_rwlock_rdlock(&lock);
  puts("b");
  ...
  puts("B");
  pthread_rwlock_unlock(&lock);
}
```

```
void ThreadC() {
  pthread_rwlock_wrlock(&lock);
  puts("c");
  ...
  puts("C");
  pthread_rwlock_unlock(&lock);
}
void ThreadD() {
  pthread_rwlock_wrlock(&lock);
  puts("d");
  ...
  puts("D");
  pthread_rwlock_unlock(&lock);
}
```

exercise: which of these outputs are possible?

1. aAbBcCdD 2. abABcdDC 3. cCabBAdD
4. cdCDaAbB 5. caACdDbB

rwlocks with monitors (attempt 1)

```
mutex_t lock;
```

lock to protect shared state

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
```

state: number of active readers, writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
```

conditions to wait for (no readers or writers, no writers)

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
```

```
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

broadcast — wakeup all readers when no writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
```

```
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

wakeup a single writer when no readers or writers

rwlocks with monitors (attempt 1)

```
mutex_t lock;
unsigned int readers, writers;
/* condition, signal when writers becomes 0 */
cond_t ok_to_read_cv;
/* condition, signal when readers + writers becomes 0 */
cond_t ok_to_write_cv;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}
ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  cond_signal(&ok_to_write_cv);
  cond_broadcast(&ok_to_read_cv);
  mutex_unlock(&lock);
}
```

problem: wakeup readers first or writer first?

this solution: wake them all up and they fight! inefficient!

reader/writer-priority

policy question: writers first or readers first?

- writers-first: no readers go when writer waiting
- readers-first: no writers go when reader waiting

previous implementation: whatever randomly happens

- writers signalled first, maybe gets lock first?
- ...but non-determinstic in pthreads

can make explicit decision

reader/writer-priority

policy question: writers first or readers first?

- writers-first: no readers go when writer waiting
- readers-first: no writers go when reader waiting

previous implementation: whatever randomly happens

- writers signalled first, maybe gets lock first?
- …but non-determinstic in pthreads

can make explicit decision

key method: track number of waiting readers/writers

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
        || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
         || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

writer-priority (1)

```
mutex_t lock; cond_t ok_to_read_cv; cond_t ok_to_write_cv;
int readers = 0, writers = 0;
int waiting_writers = 0;
```

```
ReadLock() {
  mutex_lock(&lock);
  while (writers != 0
         || waiting_writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
  mutex_unlock(&lock);
}
```

```
WriteLock() {
  mutex_lock(&lock);
  ++waiting_writers;
  while (readers + writers != 0) {
    cond_wait(&ok_to_write_cv, &lock);
  }
  --waiting_writers;
  ++writers;
  mutex_unlock(&lock);
}

WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_writers != 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				`0`	`0`	`0`

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				`0`	`0`	`0`

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	reader 2	writer 1	reader 3	W	R	WW
				0	0	0
ReadLock				0	1	0

```
mutex_lock(&lock);
while (writers != 0 || waiting_writers != 0) {
  cond_wait(&ok_to_read_cv, &lock);
}
++readers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				`0`	`0`	`0`
`ReadLock`				`0`	`1`	`0`
`(reading)`	`ReadLock`			`0`	`2`	`0`

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
`ReadLock`				0	1	0
`(reading)`	`ReadLock`			0	2	0
`(reading)`	`(reading)`	`WriteLock wait`		0	2	1

```
mutex_lock(&lock);
++waiting_writers;
while (readers + writers != 0) {
  cond_wait(&ok_to_write_cv, &lock);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				`0`	`0`	`0`
`ReadLock`				`0`	`1`	`0`
`(reading)`	`ReadLock`			`0`	`2`	`0`
`(reading)`	`(reading)`	`WriteLock` wait		`0`	`2`	`1`
`(reading)`	`(reading)`	`WriteLock` wait	`ReadLock` wait	`0`	`2`	`1`

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(read	wait	ReadLock wait	0	2	1
ReadUnlock	(r	wait	ReadLock wait	0	1	1

```
mutex_lock(&lock);
--readers;
if (readers == 0)
  ...
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	Write			1	1
	ReadUnlock	Wri			0	1

```
mutex_lock(&lock);
--readers;
if (readers == 0)
  cond_signal(&ok_to_write_cv)
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	Read			0	2	0
(reading)	(rea			0	2	1
(reading)	(rea		it	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0

```
while (readers + writers != 0) {
  cond_wait(&ok_to_write_cv, &lock);
}
--waiting_writers; ++writers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLo			0	2	0
(reading)	(readi			0	2	1
(reading)	(readi		wait	0	2	1
ReadUnlock	(readi		wait	0	1	1
	ReadUr		wait	0	0	1
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0

```
mutex_lock(&lock);
if (waiting_writers != 0) {
  cond_signal(&ok_to_write_cv);
} else {
  cond_broadcast(&ok_to_read_cv);
}
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)					
(reading)	(reading)					
ReadUnlock	(reading)					
	ReadUnlock					
		WriteLock	ReadLoc wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0
			ReadLock	0	1	0

```
while (writers != 0 && waiting_writers != 0) {
  cond_wait(&ok_to_read_cv, &lock);
}
++readers;
mutex_unlock(&lock);
```

simulation of reader/write lock

writer-priority version

W = writers, R = readers, WW = waiting_writers

reader 1	**reader 2**	**writer 1**	**reader 3**	**W**	**R**	**WW**
				0	0	0
ReadLock				0	1	0
(reading)	ReadLock			0	2	0
(reading)	(reading)	WriteLock wait		0	2	1
(reading)	(reading)	WriteLock wait	ReadLock wait	0	2	1
ReadUnlock	(reading)	WriteLock wait	ReadLock wait	0	1	1
	ReadUnlock	WriteLock wait	ReadLock wait	0	0	1
		WriteLock	ReadLock wait	1	0	0
		(read+writing)	ReadLock wait	1	0	0
		WriteUnlock	ReadLock wait	0	0	0
			ReadLock	0	1	0

reader-priority (1)

```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  ...
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers +
         readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (readers == 0 && waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

reader-priority (1)

```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  ...
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
}
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers +
         readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (readers == 0 && waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

rwlock exercise

suppose we want something in-between reader and writer priority:

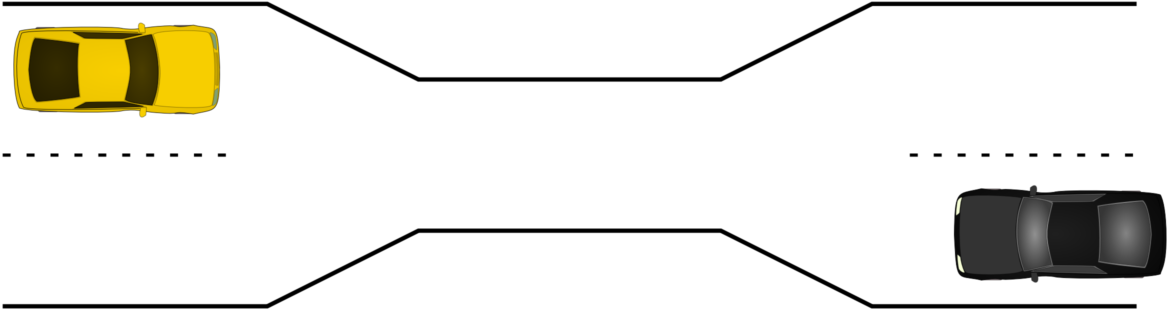
reader-priority except if writers wait more than 1 second

exercise: what do we change?

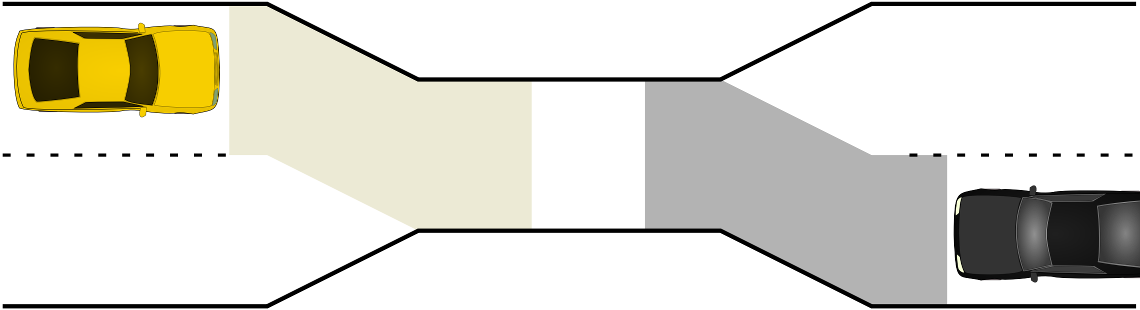
```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

ReadUnlock() {
  mutex_lock(&lock);
  --readers;
  if (waiting_readers == 0 &&
      readers == 0) {
    cond_signal(&ok_to_write_cv);
  }
```

```
WriteLock() {
  mutex_lock(&lock);
  while (waiting_readers + readers + writers != 0) {
    cond_wait(&ok_to_write_cv);
  }
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);
  --writers;
  if (waiting_readers == 0) {
    cond_signal(&ok_to_write_cv);
  } else {
    cond_broadcast(&ok_to_read_cv);
  }
  mutex_unlock(&lock);
}
```

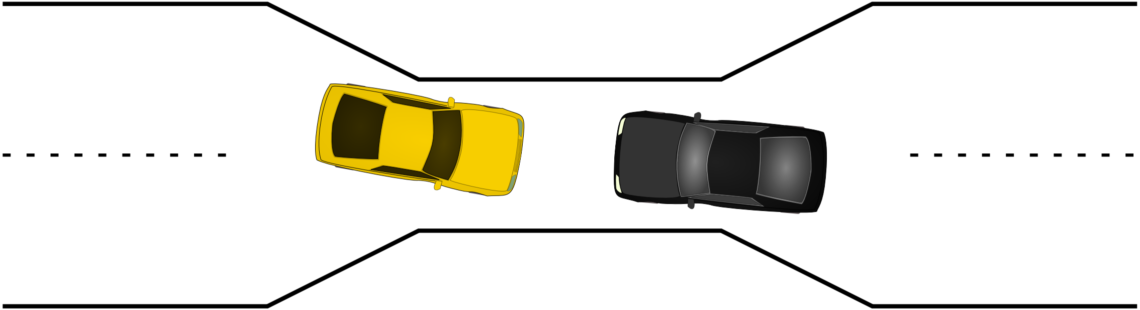
the one-way bridge



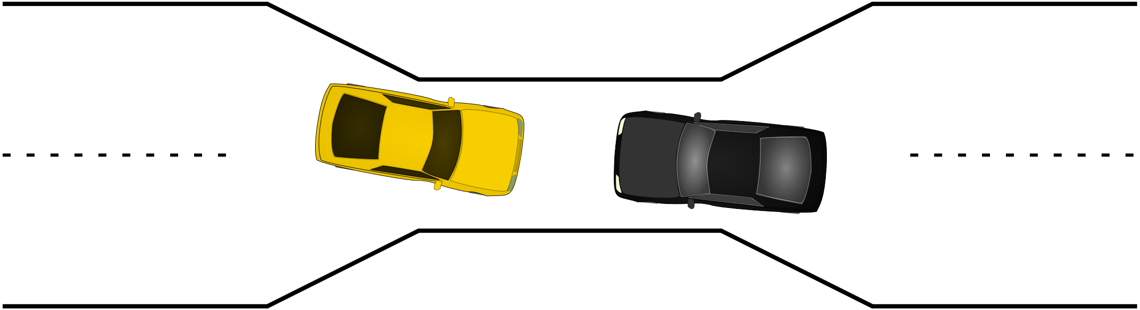
the one-way bridge



the one-way bridge



the one-way bridge



pipe() deadlock

BROKEN example:

```
int child_to_parent_pipe[2], parent_to_child_pipe[2];
pipe(child_to_parent_pipe); pipe(parent_to_child_pipe);
if (fork() == 0) {
    /* child */
    write(child_to_parent_pipe[1], buffer, HUGE_SIZE);
    read(parent_to_child_pipe[0], buffer, HUGE_SIZE);
    exit(0);
} else {
    /* parent */
    write(parent_to_child_pipe[1], buffer, HUGE_SIZE);
    read(child_to_parent[0], buffer, HUGE_SIZE);
}
```

This will hang forever (if `HUGE_SIZE` is big enough).

deadlock waiting

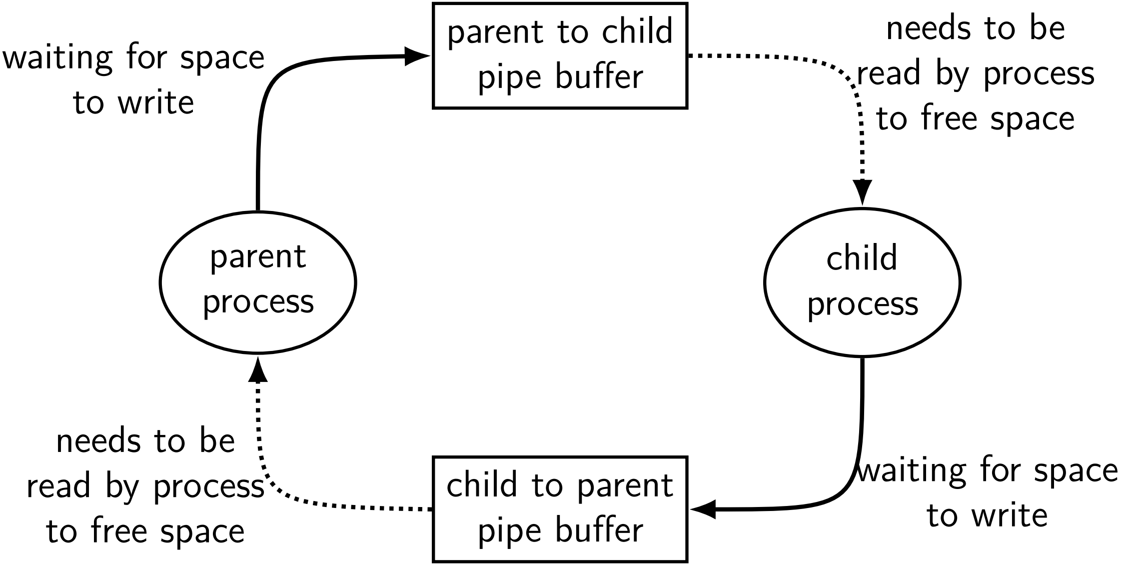
child writing to pipe waiting for free buffer space

…which will not be available until parent reads

parent writing to pipe waiting for free buffer space

…which will not be available until child reads

circular dependency



parent to child
pipe buffer

waiting for space
to write

needs to be
read by process
to free space

parent
process

child
process

child to parent
pipe buffer

needs to be
read by process
to free space

waiting for space
to write

moving two files

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

  to_dir->entries[filename] = from_dir->entries[filename];
  from_dir->entries.erase(filename);

  mutex_unlock(&to_dir->lock);
  mutex_unlock(&from_dir->lock);
}
```

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

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

moving two files: lucky timeline (1)

Thread 1	**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	**Thread 2**
`MoveFile(A, B, "foo")`	`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	**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)

moving two files: unlucky timeline

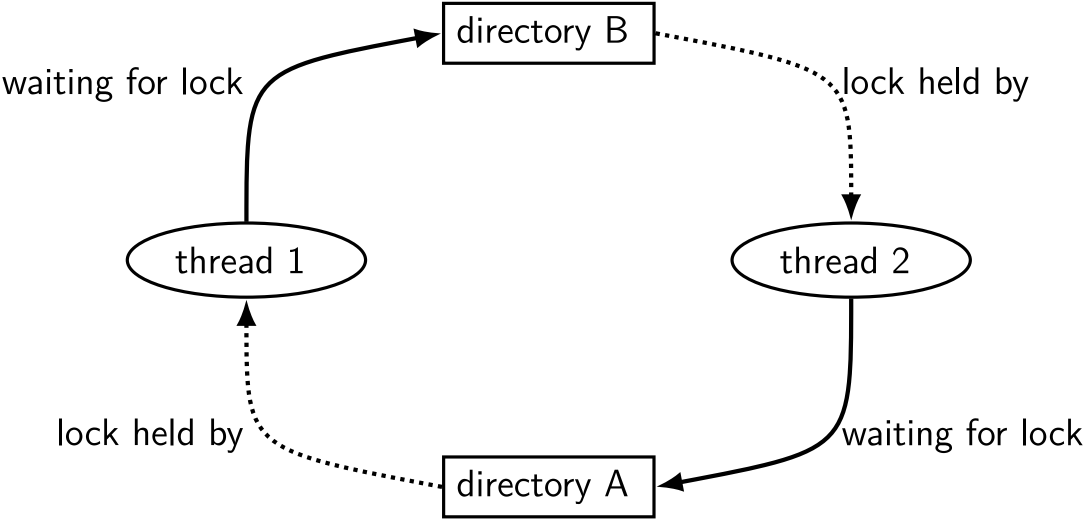
Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`… stalled	
(waiting for lock on B)	`lock(&A->lock`… stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~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

moving two files: unlucky timeline

Thread 1 `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

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



directory B

waiting for lock

lock held by

thread 1

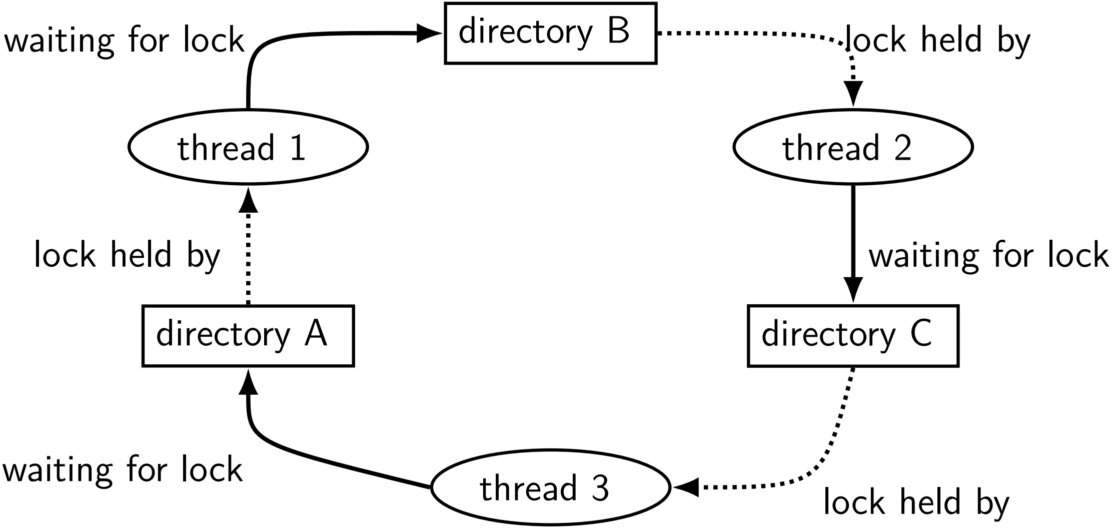
thread 2

lock held by

waiting for lock

directory A

moving three files: dependencies


waiting for lock

directory B

lock held by

thread 1

thread 2

lock held by

waiting for lock

directory A

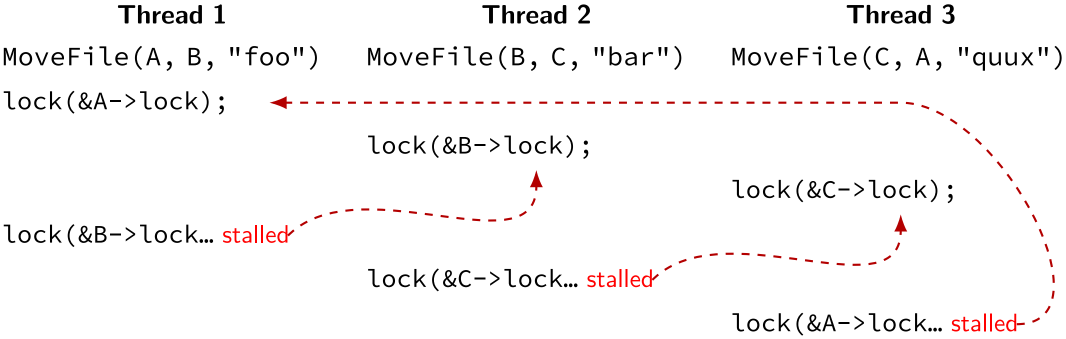
directory C

waiting for lock

thread 3

lock held by

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

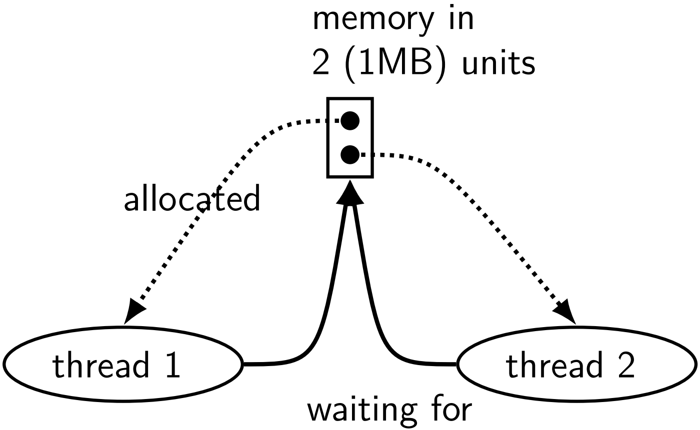
Thread 1	**Thread 2**
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

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);`

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

backup slides

deadlock detection

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

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?

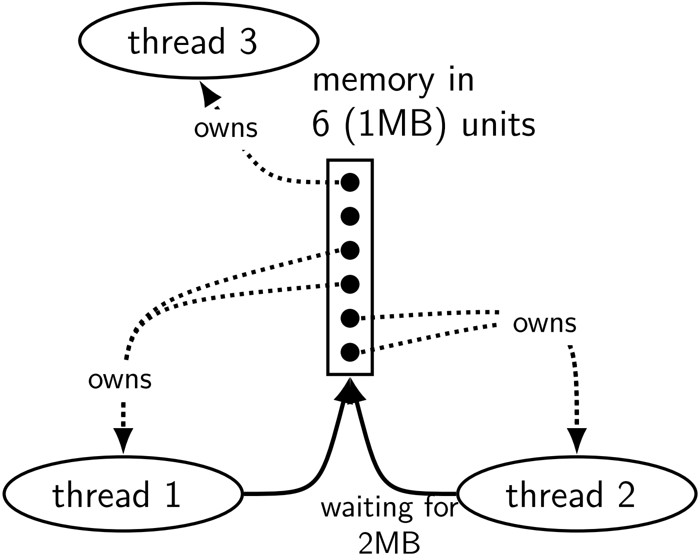
not a deadlock — thread 3 can still finish

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

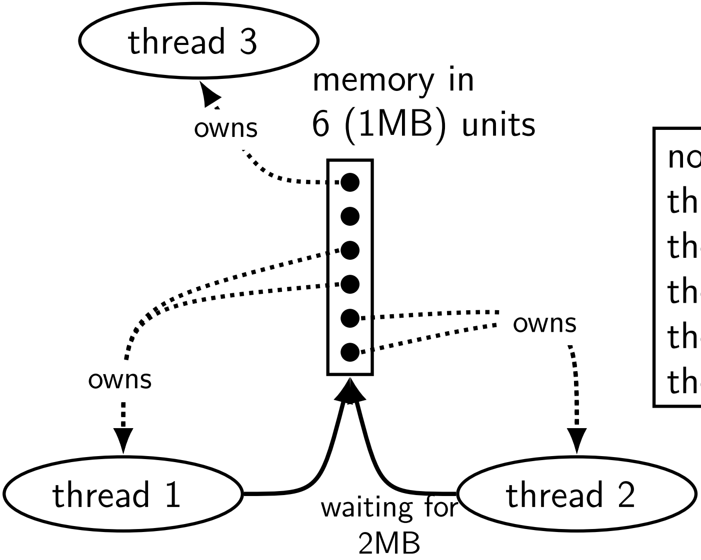
…but would be deadlock

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

divisible resources: not deadlock

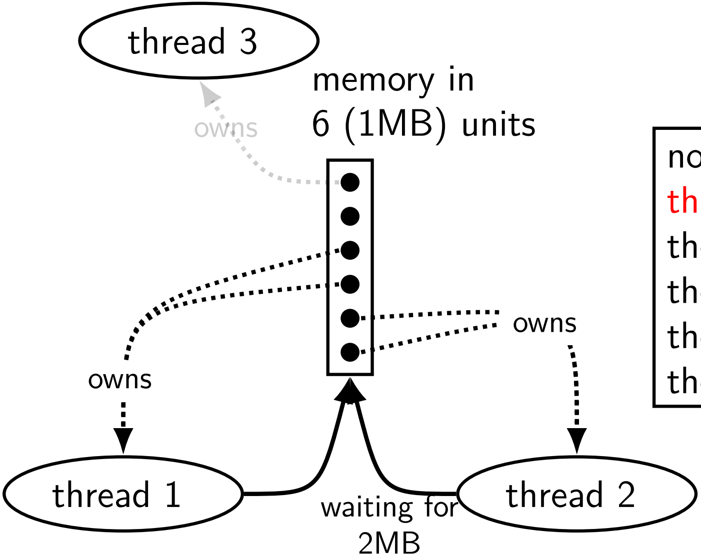


divisible resources: not deadlock



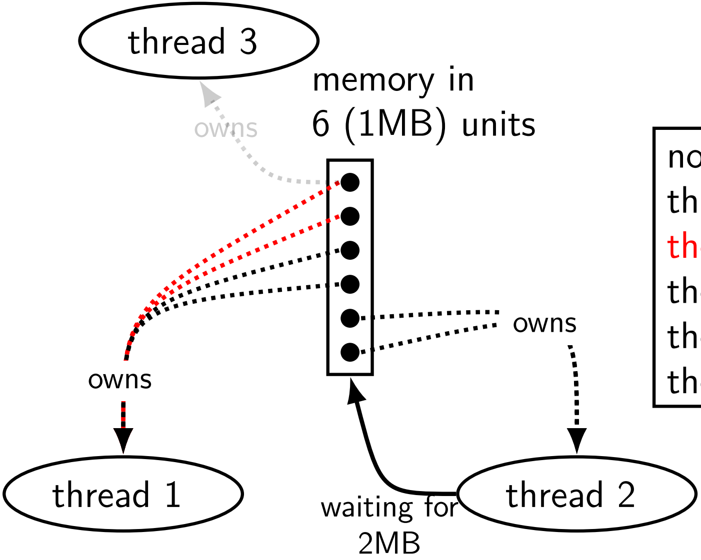
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



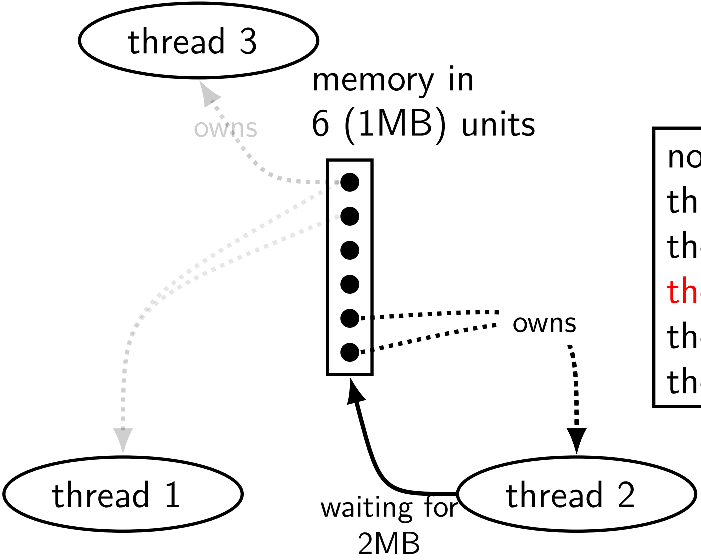
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



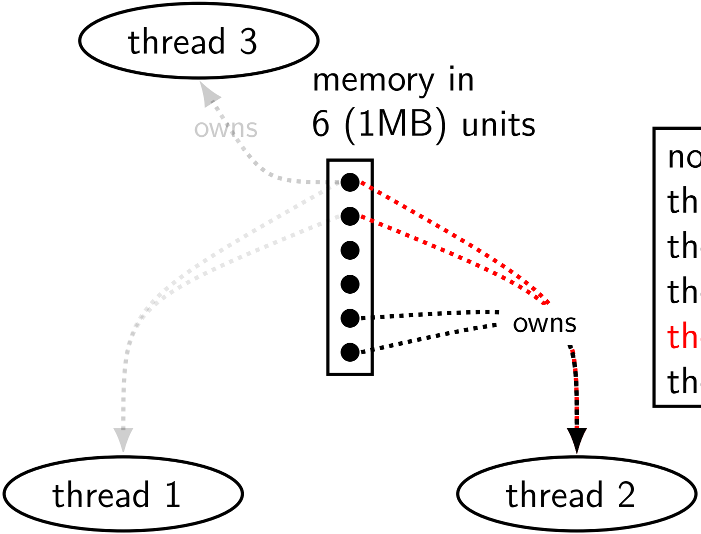
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



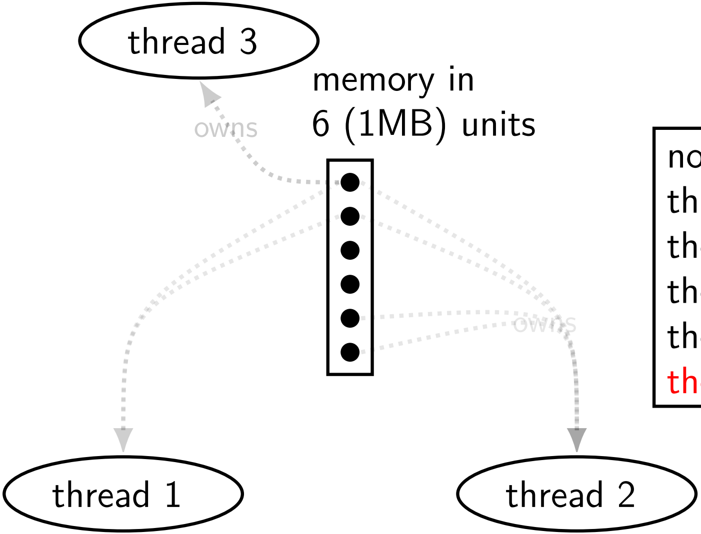
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



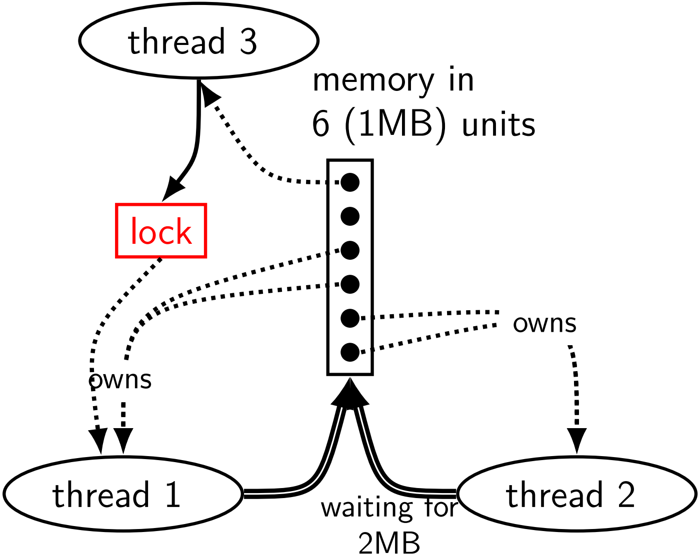
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

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



thread 3

memory in
6 (1MB) units

lock

owns

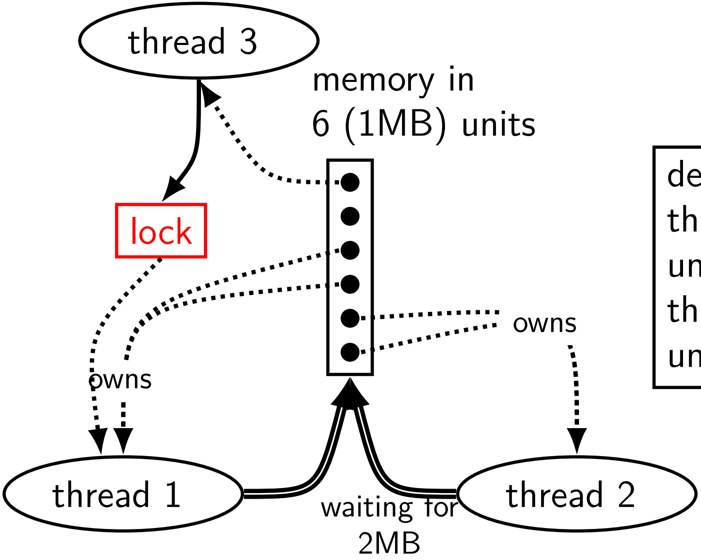
owns

thread 1

waiting for
2MB

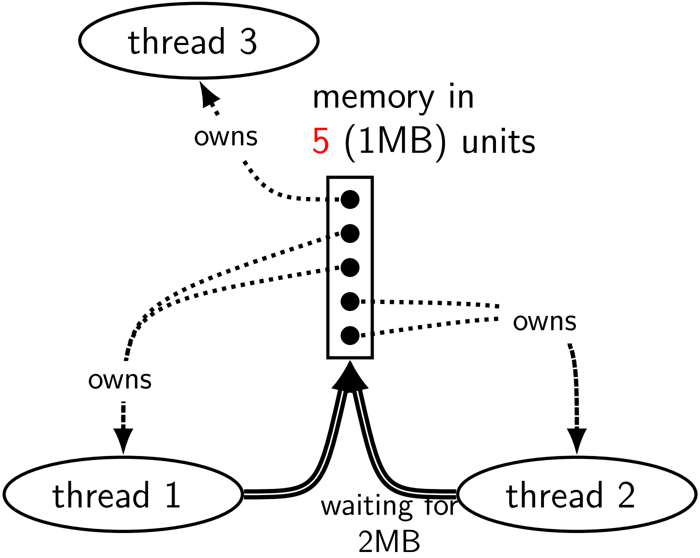
thread 2

divisible resources: is deadlock

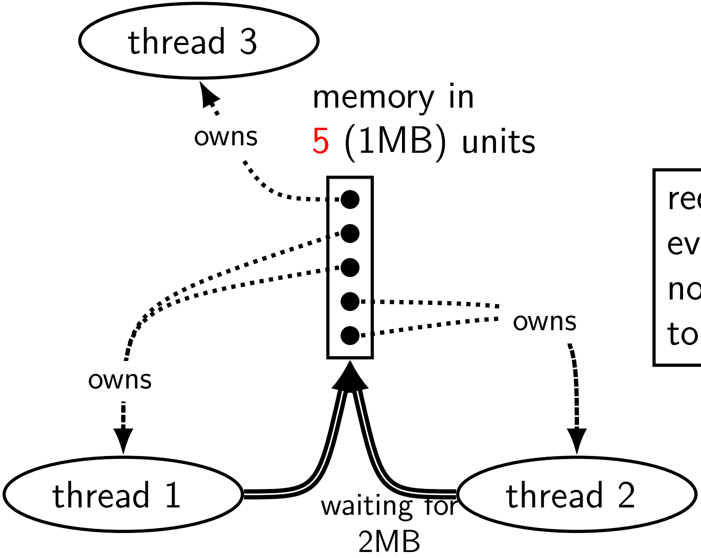


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

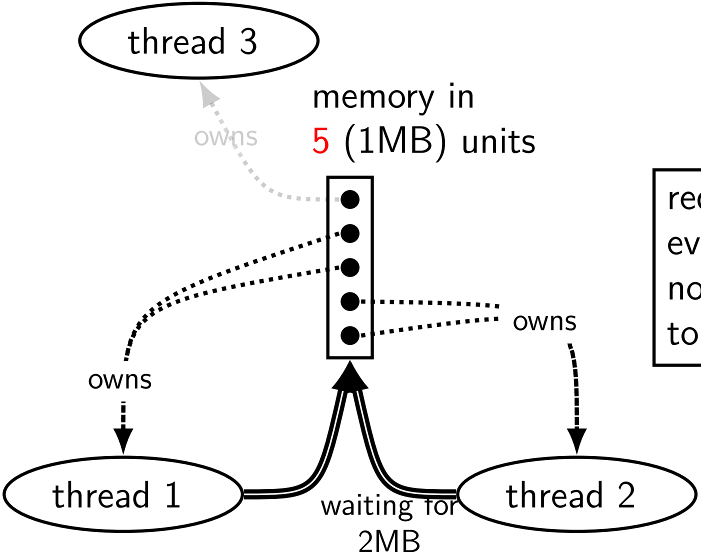


divisible resources: is deadlock



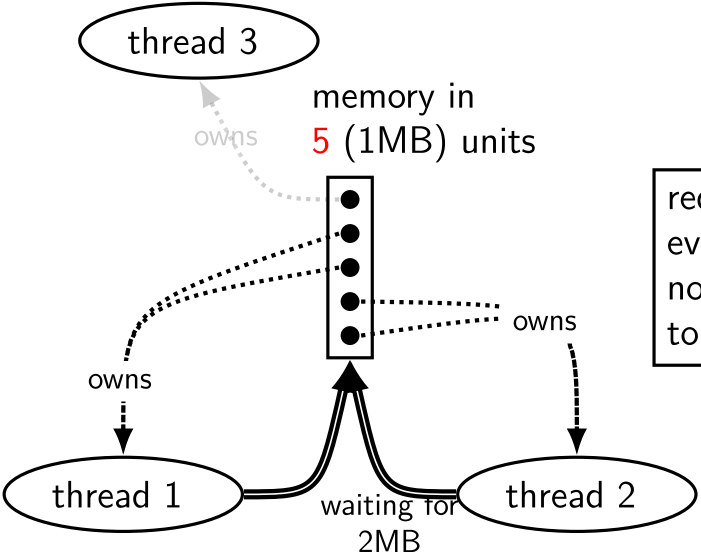
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



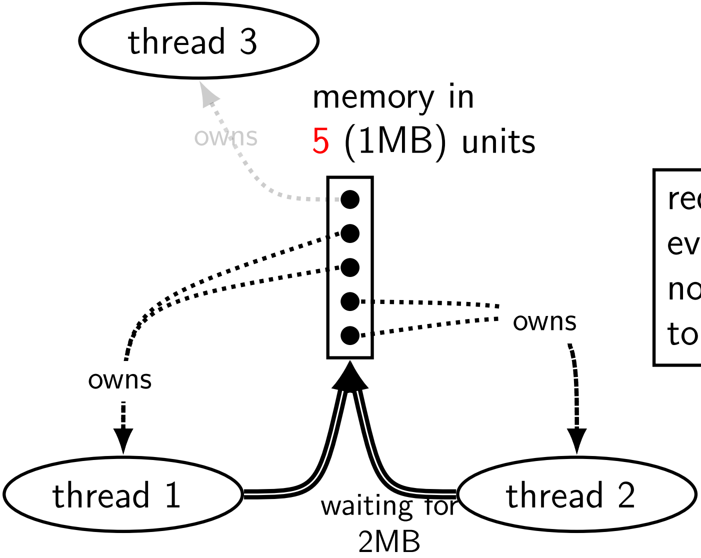
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



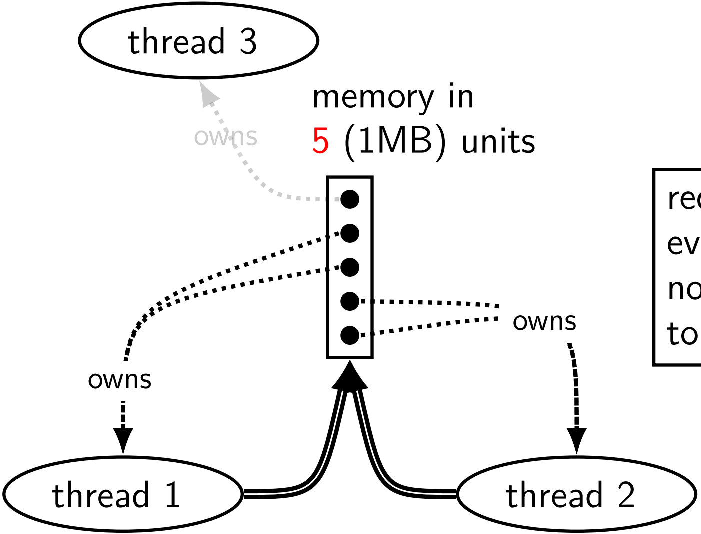
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

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 divisibe resources

can't rely on cycles in graphs in this case

alternate algorithm exists

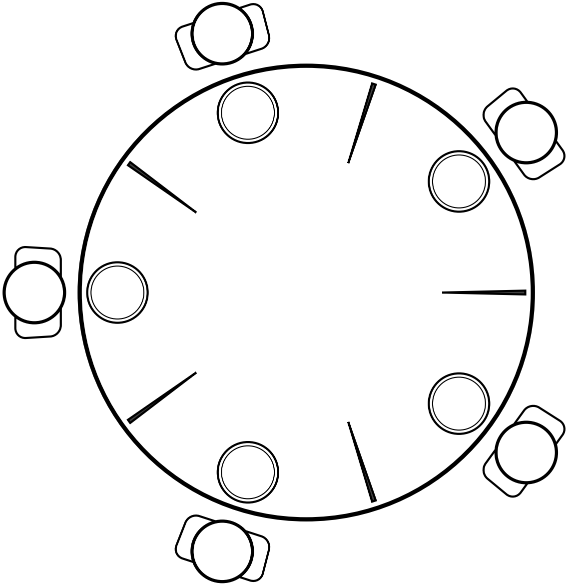
- similar technique to how we showed no deadlock

high-level intuition: simulate what could happen

- find threads that could finish based on resources available now

full details: look up Baker's algorithm

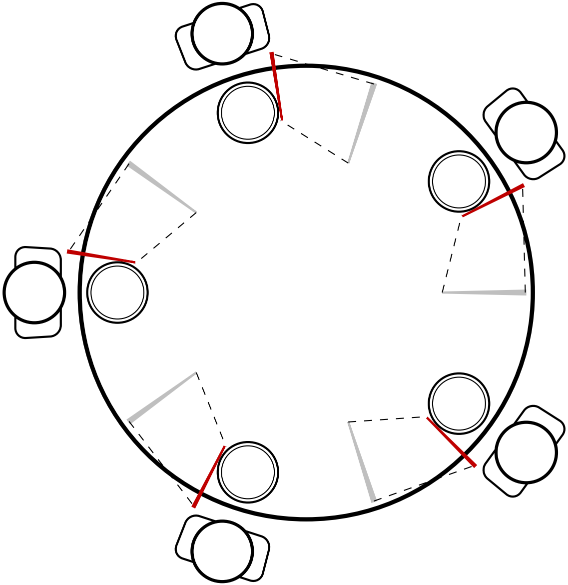
dining philosophers



five philosophers either think or eat

to eat, grab chopsticks on either side

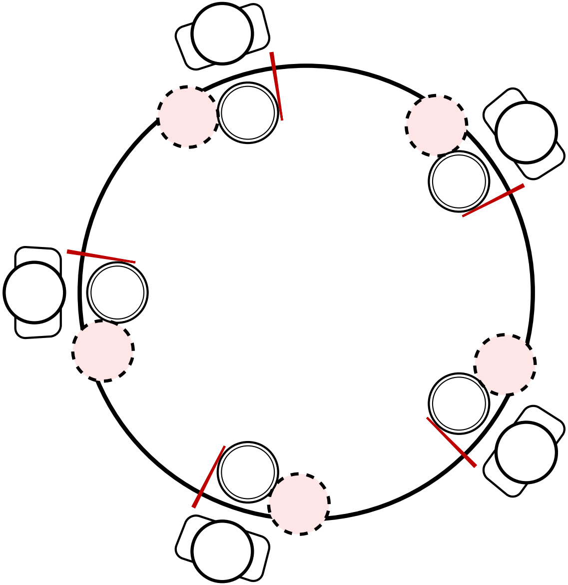
dining philosophers



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

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();
}
```

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;
}
```

monitor ordering exercise: solution

(one of many possible solutions)

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

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

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

allocating all at once?

for resources like disk space, memory

figure out maximum allocation when starting thread

- “only” need conservative estimate

only start thread if those resources are available

okay solution for embedded systems?