synchronization 5 / deadlock

last time

monitor examples

- lock protecting all shared data
- condition variable (list of waiters) for each thing waited for
- while (need to wait) cond_wait
- if (reason to wait changed) broadcast/signal

counting semaphores

- up/post (increment) or down/wait (wait till non-zero, decrement)
- bookkeeping a count of something
- such that count = 0 if we need to wait

on the quiz

typos (from editing the question at the last moment) on question 1C

- dropped, but...
- seems like a significant number students got this wrong despite knowing what was meant (despite very poor wording)

Question 4/5 unintentionally missing clear of current_pair

meant best answer was for the blank to include this (none of the above on Q4)

on the quiz (2)

```
LockMutex(Mutex *m) {
  LockSpinlock(&m->guard_spinlock);
  if (m->lock_taken) {
    put current thread on m->wait_queue
    mark current thread not runnable
    /* xv6: myproc()->state = SLEEPING; */
    UnlockSpinlock(&m->guard_spinlock);
    run scheduler
    /****/ m->lock_taken = true;
  } 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 that thread as runnable
    /* xv6: myproc()->state = RUNNABLE; */
  }
  m->lock_taken = false;
  UnlockSpinlock(&m->guard_spinlock);
}
```

on the quiz (3)

```
sem_t mutex;
sem_t make_pair;
sem_t finish_pair; /* initially 0 */
std::vector<string> current_pair;

std::vector<string> WaitForPair(string name) {
    std::vector<string> result;
    sem_wait(&make_pair);
    sem_wait(&mutex);
    current_pair.push_back(name);
    if (current_pair.size() == 2) {
        result = current_pair;
        sem_post(&mutex);
        sem_post(&finish_pair);
    } else { /* current_pair.size() == 1 */
        sem_post(&mutex);
        sem_wait(&finish_pair);
        sem_wait(&mutex);
        result = current_pair;
        sem_post(&mutex);
        /*** BLANK ONE ***/
        current_pair.clear(); /* <-- meant to include outside of blank */
        sem_post(&make_pair); sem_post(&make_pair);
    }
    return result;
}
```

on the quiz (4)

```
pthread_mutex_t lock;
pthread_cond_t global_cv;
list<StudentInfo*> waiting_students;
StudentInfo *GetNextStudent(TAInfo *ta) {
    StudentInfo *student = NULL;
    pthread_mutex_lock(&lock);
    while (waiting_students.size() == 0) {
        ___________________________ /* BLANK ONE */
    }
    student = waiting_students.front();
    waiting_students.pop_front();
    student->helped_by = ta;
    ___________________________ /* BLANK TWO */
    pthread_mutex_unlock(&lock);
    return student;
}

TAInfo *WaitForNextTA(StudentInfo *student) {
    TAInfo *ta;
    pthread_mutex_lock(&lock);
    student->helped_by = NULL;
    waiting_students.push_back(student);
    pthread_cond_signal(&global_cv);
    while (student->helped_by == NULL) {
        ______________________________ /* BLANK FOUR */
    }
```

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	Wr			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
		WriteLo k	ReadLock wait	1	0	0
		(read+w iting)	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	ReadLock 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);
}
```

rwlock exercise soln

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

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

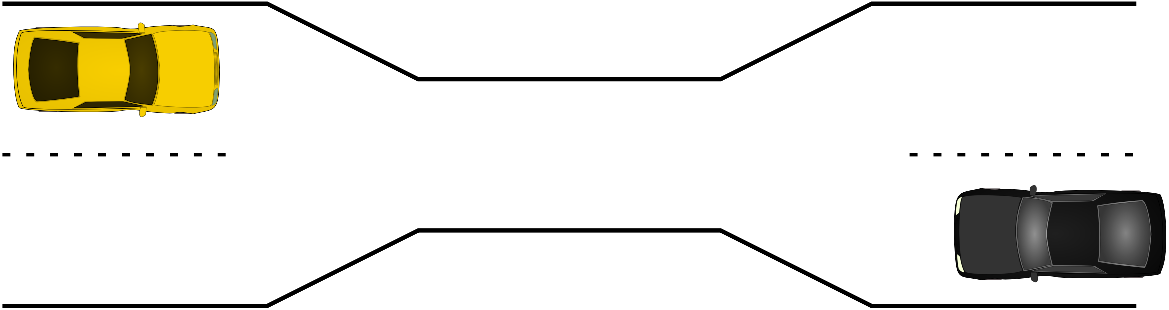
```
WriteLock() {
  mutex_lock(&lock);
  RecordStartWaiting();
  while (readers + writers != 0 ||
         (waiting_readers != 0 &&
         !WritersWaitingTooLong())) {
    cond_wait(&ok_to_write_cv);
  }
  RecordStopWaiting();
  ++writers;
  mutex_unlock(&lock);
}
WriteUnlock() {
  mutex_lock(&lock);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

rwlock exercise soln

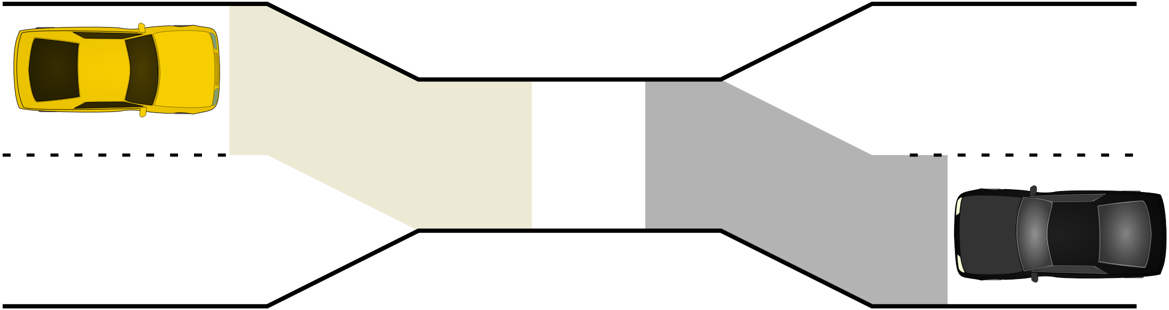
```
...
int waiting_readers = 0;
ReadLock() {
  mutex_lock(&lock);
  ++waiting_readers;
  while (writers != 0
     || WritersWaitingTooLong()) {
    cond_wait(&ok_to_read_cv, &lock);
  }
  --waiting_readers;
  ++readers;
  mutex_unlock(&lock);
}

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

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

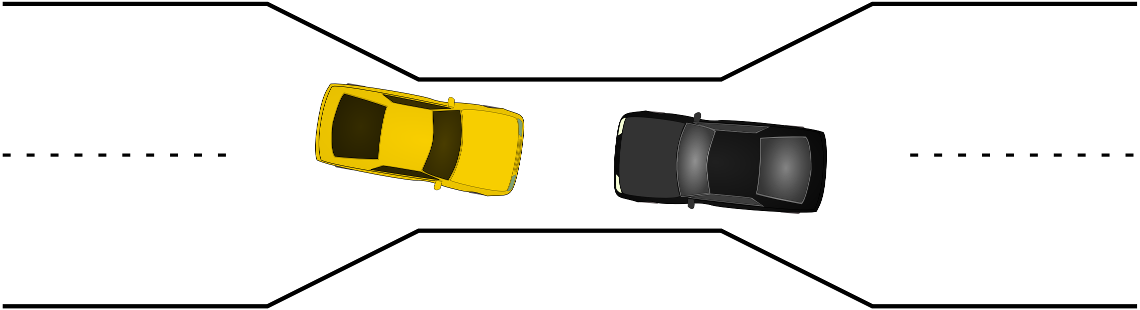
the one-way bridge



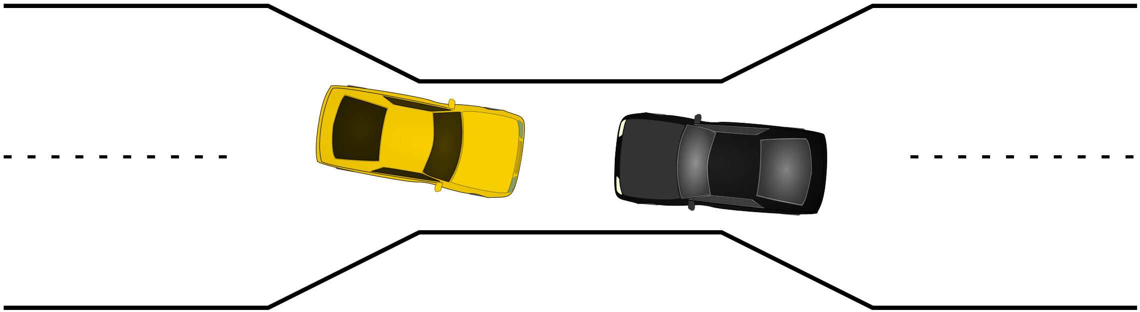
the one-way bridge



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_pipe[0], buffer, HUGE_SIZE);
}
```

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

deadlock waiting

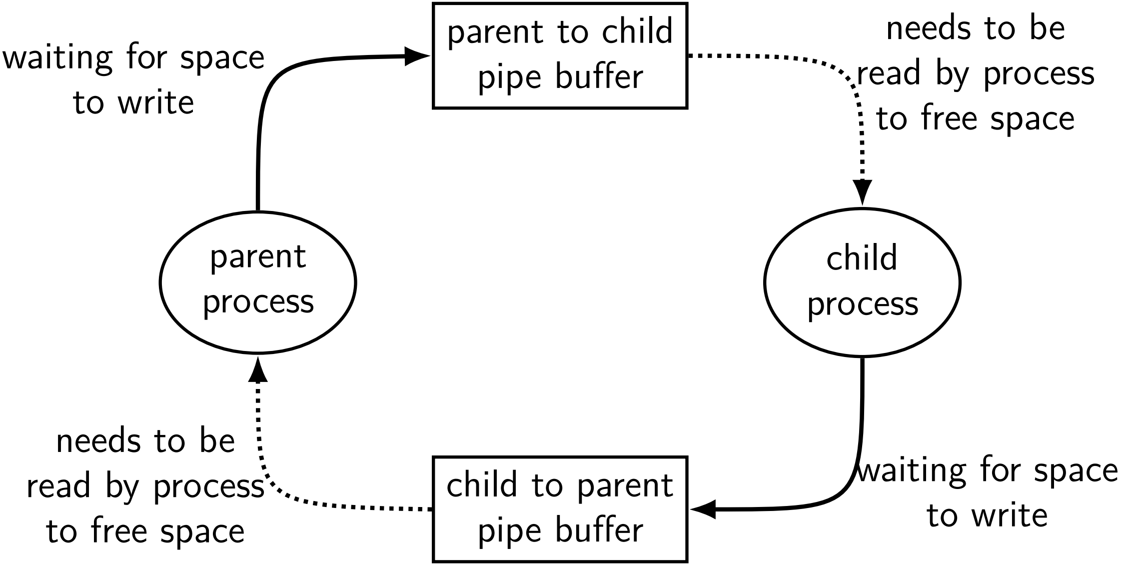
child writing to pipe waiting for free buffer space

…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



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

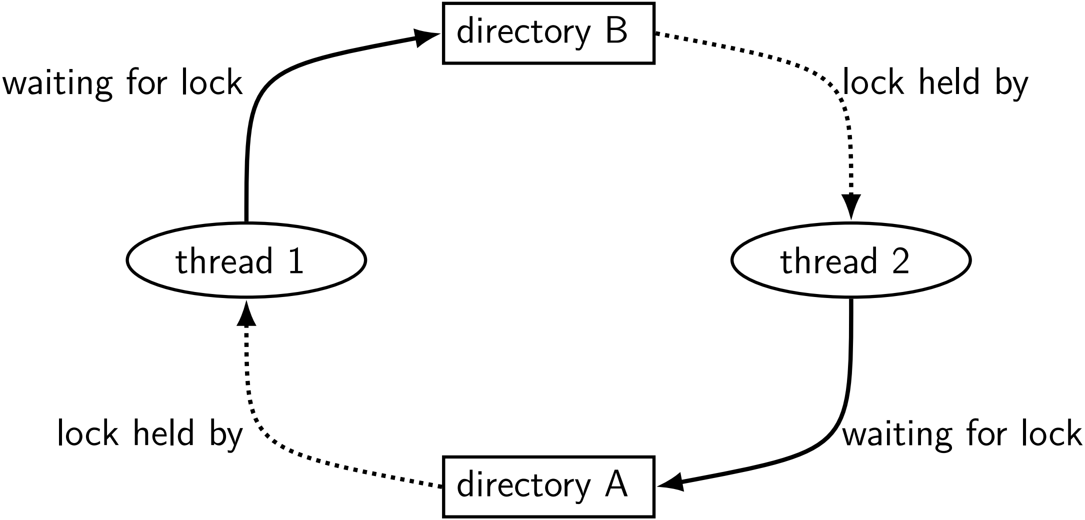
Thread 1 `MoveFile(A, B, "foo")`	**Thread 2** `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock…` stalled	
(waiting for lock on B)	`lock(&A->lock…` stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~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

moving two files: unlucky timeline

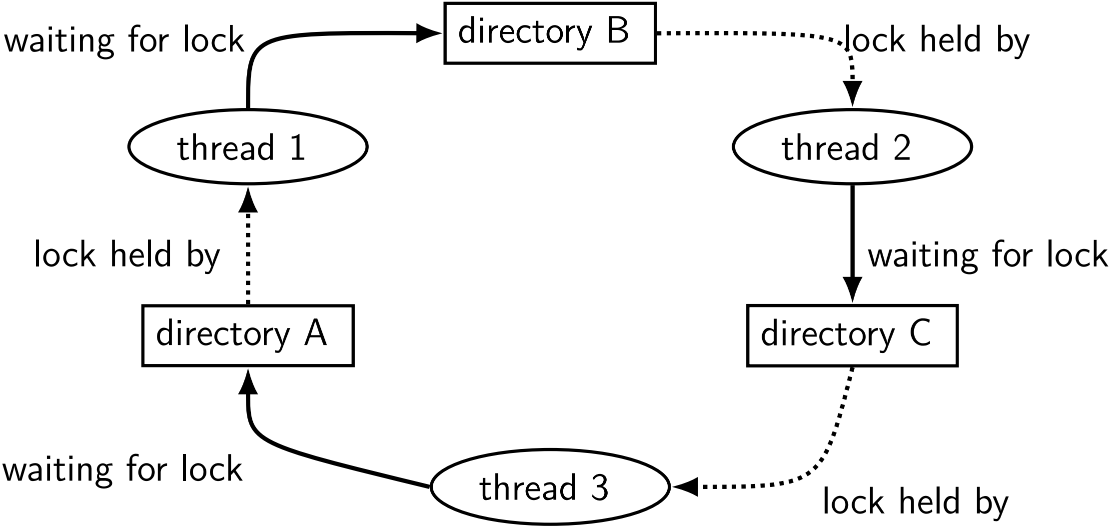
Thread 1 `MoveFile(A, B, "foo")`	Thread 2 `MoveFile(B, A, "bar")`
`lock(&A->lock);`	
	`lock(&B->lock);`
`lock(&B->lock`… stalled	
(waiting for lock on B)	`lock(&A->lock`… stalled
(waiting for lock on B)	(waiting for lock on A)
~~(do move)~~ unreachable	~~(do move)~~ unreachable
~~`unlock(&B->lock);`~~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



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

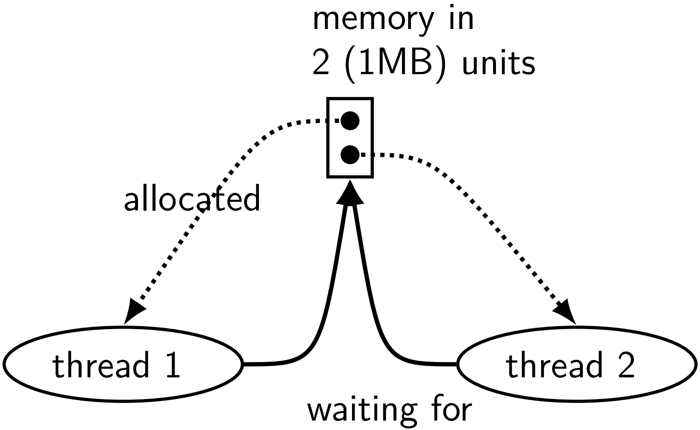
Thread 1	**Thread 2**
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`AllocateOrWaitFor(1 MB)`	`AllocateOrWaitFor(1 MB)`
`(do calculation)`	`(do calculation)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

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);`

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->prev->lock);
    pthread_mutex_unlock(&node->lock);
}
```

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

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

how is deadlock — solution

Remove B	Remove C
lock B	lock C
lock A (prev)	wait to lock B (prev)
wait to lock C (next)	

With B and D — only overlap in in node C — no circular wait possible

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources or at least enough that never run out	no *mutual exclusion*
no shared resources	no *mutual exclusion*
no waiting "busy signal" — abort and (maybe) retry revoke/preempt resources	no *hold and wait*/ *preemption*
acquire resources in **consistent order**	no *circular wait*
request **all resources at once**	no *hold and wait*

deadlock prevention techniques

infinite resources — 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
...

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 — no *mutual exclusion*

or at least enough that never run out

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 summary

backup slides

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 — 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*

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 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?

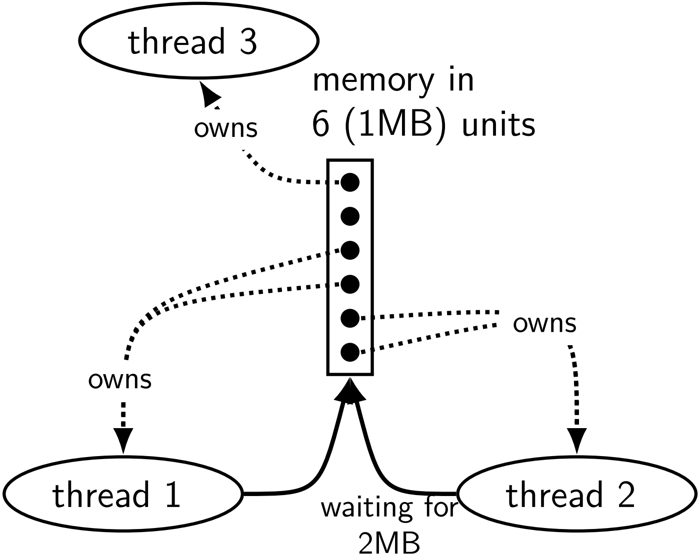
not a deadlock — thread 3 can still finish

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

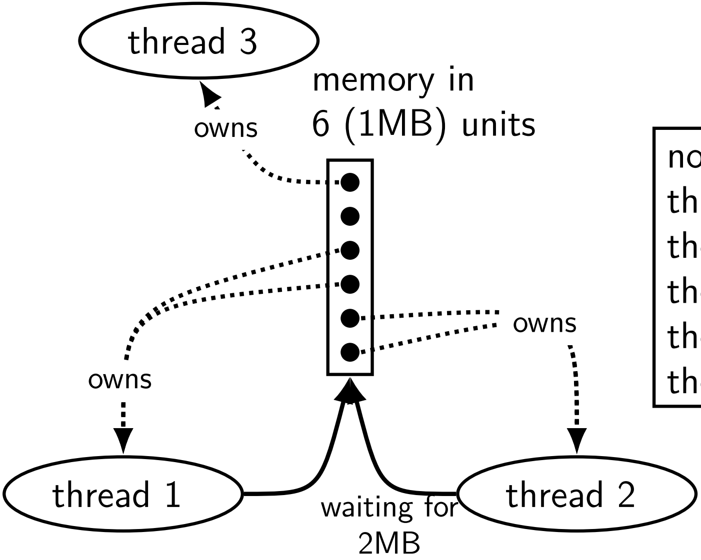
...but would be deadlock

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

divisible resources: not deadlock

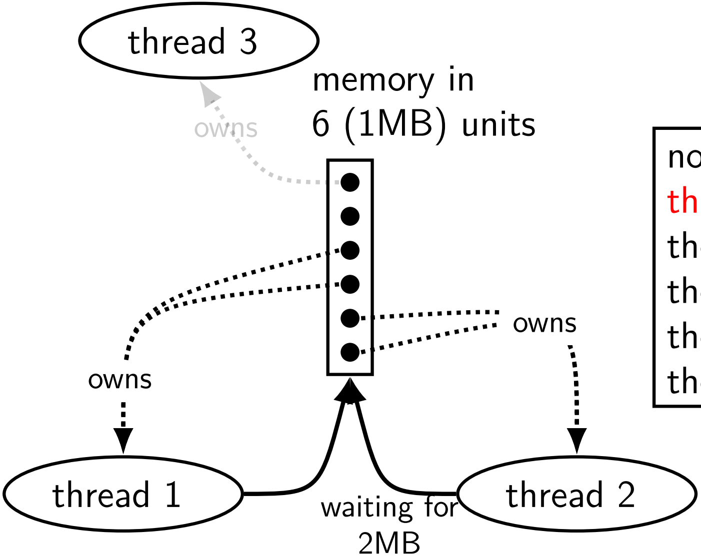


divisible resources: not deadlock



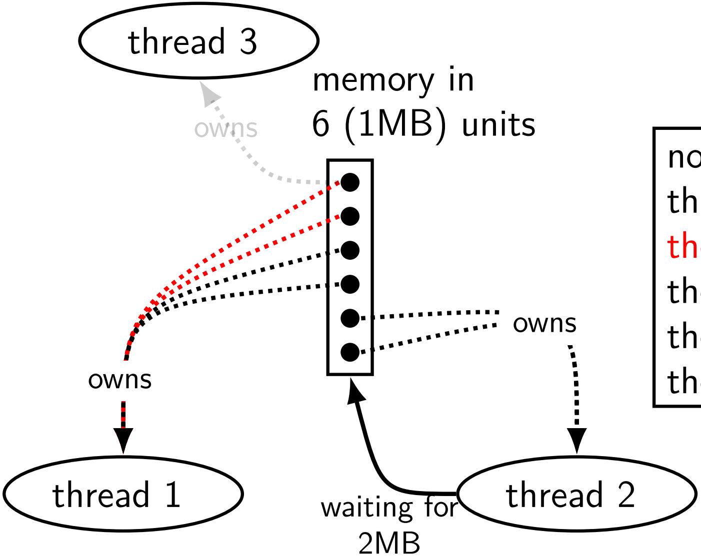
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



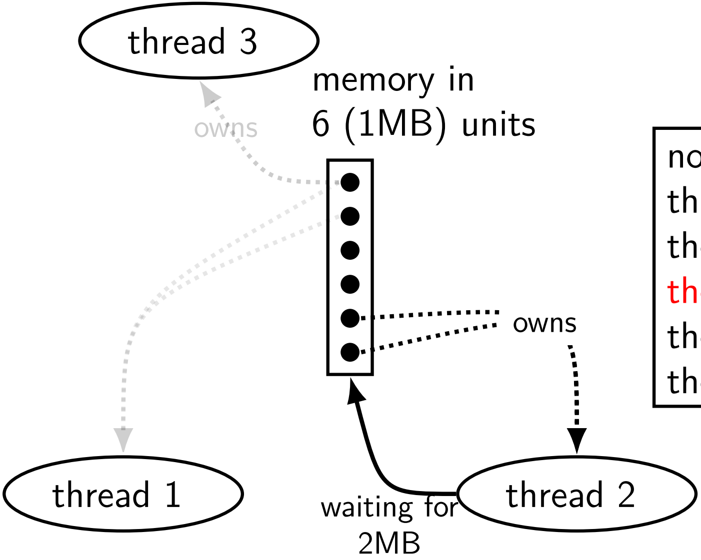
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



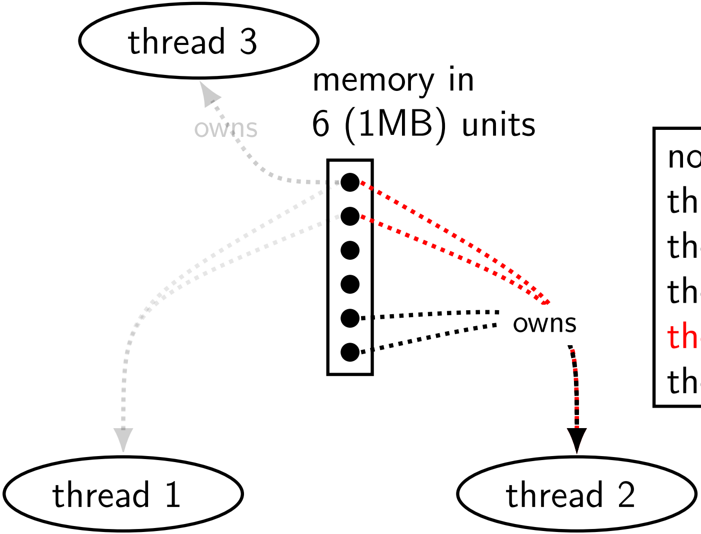
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



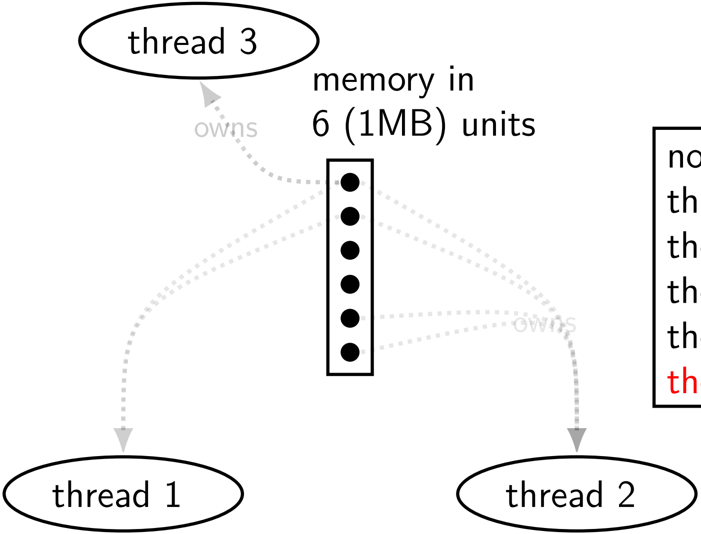
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock



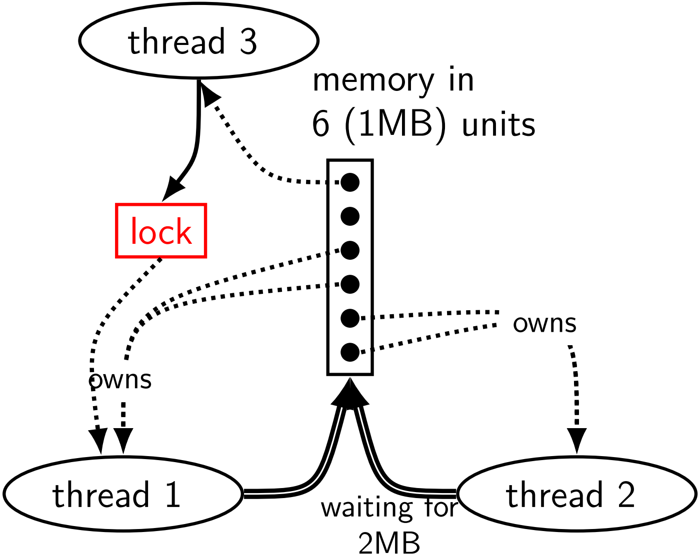
not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: not deadlock

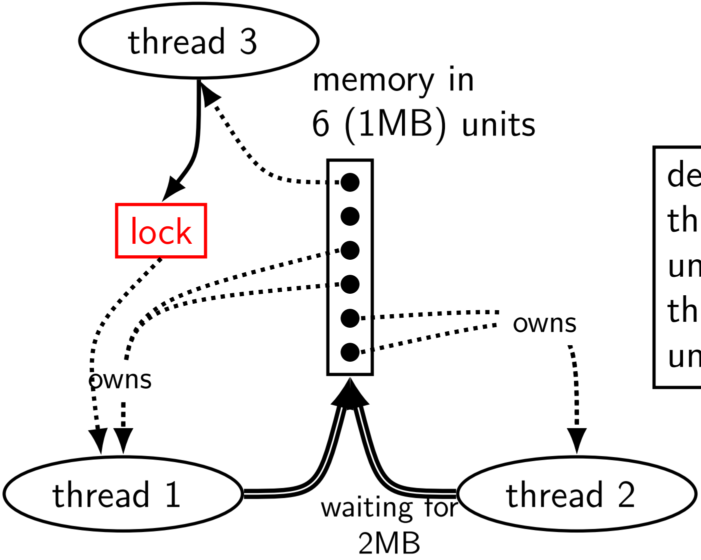


not deadlock:
thread 3 finishes
then thread 1 can get memory
then thread 1 finishes
then thread 2 can get resources
then thread 2 can finish

divisible resources: is deadlock

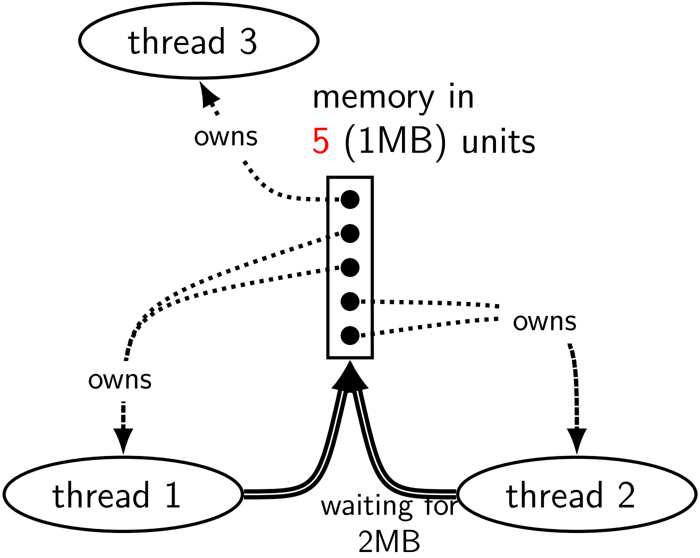


divisible resources: is deadlock

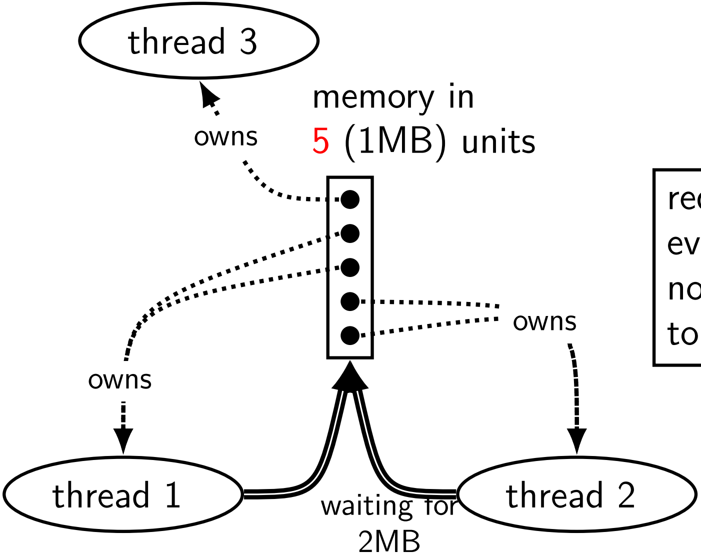


deadlock:
thread 3 can't finish
until thread 1 releases lock, but
thread 1 can't finish
until thread 3 releases memory

divisible resources: is deadlock

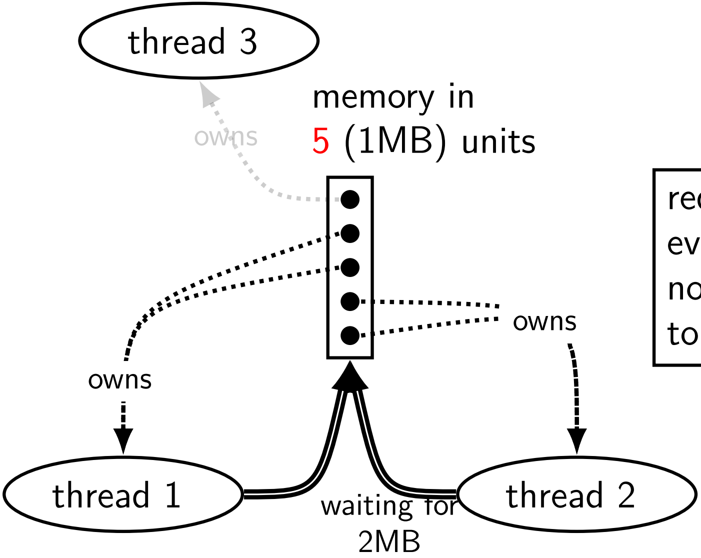


divisible resources: is deadlock



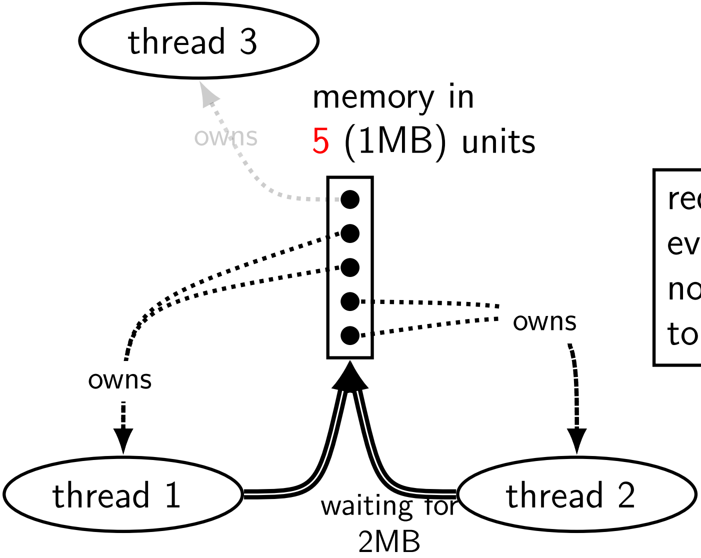
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



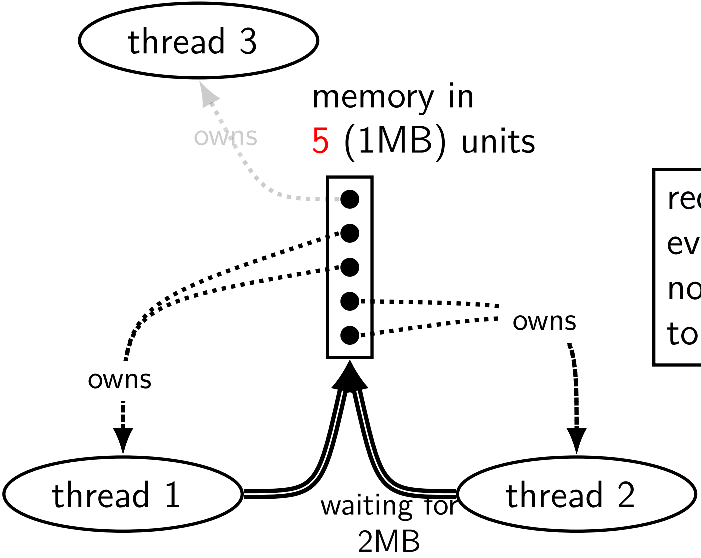
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

divisible resources: is deadlock



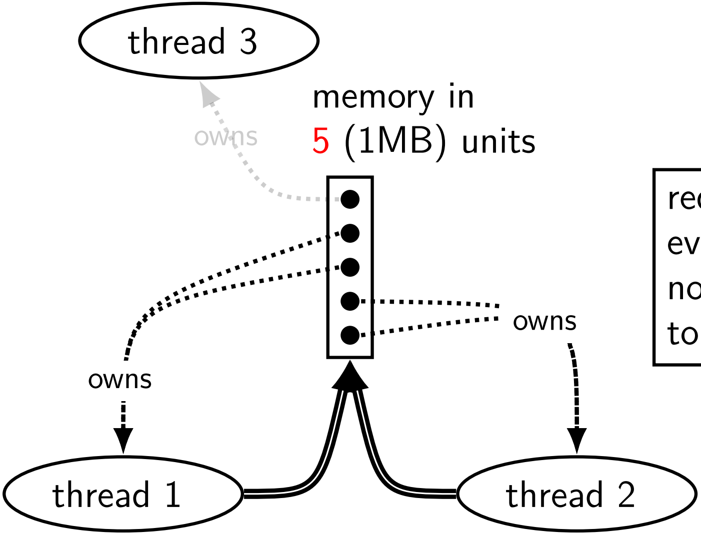
reducing memory: deadlock:
even after thread 3 finishes
no way for thread 1+2
to get what they want

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 divisibe resources

can't rely on cycles in graphs in this case

alternate algorithm exists

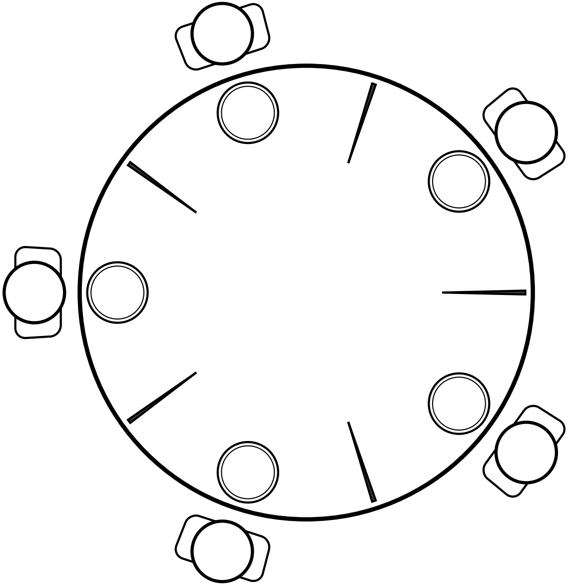
- similar technique to how we showed no deadlock

high-level intuition: simulate what could happen

- find threads that could finish based on resources available now

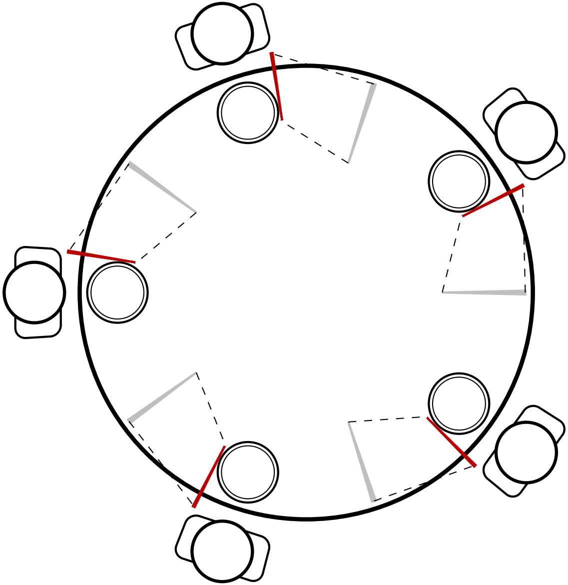
full details: look up Baker's algorithm

dining philosophers



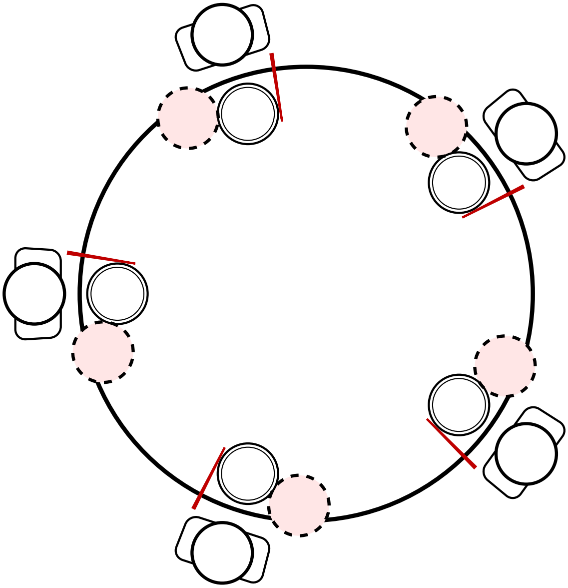
five philosophers either think or eat
to eat, grab chopsticks on either side

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

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?

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

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

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

semaphores with monitors: no condition

```
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;
    pthread_cond_signal(
        &count_is_positive_cv
    );
    pthread_mutex_unlock(&lock);
}
```

same as where we started...

semaphores with monitors: alt w/ signal

```
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;
    if (count > 0) {
        pthread_cond_signal(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

```
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    if (count == 1) {
        pthread_cond_signal(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

on signal/broadcast generally

whenever using signal need to ask
what if more than one thread is waiting?

need to explain why those threads will be signalled eventually

…even if next thread signalled doesn't run right away

another problem that would be avoided with Hoare scheduling

building semaphore with monitors (version B)

```
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;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

building semaphore with monitors (version B)

```
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;
    /* condition *just* became true */
    if (count == 1) {
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

before: signal every time

can check if condition just became true instead?

but do we really need to broadcast?

exercise: why broadcast?

```c
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);
}
```

```c
void up() {
    pthread_mutex_lock(&lock);
    count += 1;
    if (count == 1) { /* became > 0 */
        pthread_cond_broadcast(
            &count_is_positive_cv
        );
    }
    pthread_mutex_unlock(&lock);
}
```

exercise: why can't this be `pthread_cond_signal`?

hint: think of two threads calling down + two calling up?

brute force: only so many orders they can get the lock in

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

broadcast problem

Thread 1	Thread 2	Thread 3	Thread 4
Down()			
lock			
count == 0? yes			
unlock/wait			
	Down()		
	lock		
	count == 0? yes		
	unlock/wait		
		Up()	
		lock	
		count += 1 (now 1)	Up()
stop waiting on CV		signal	wait for lock
wait for lock		unlock	wait for lock
wait for lock			lock
wait for lock			count += 1 (now 2)
wait for lock			count != 1: don't signal
lock			unlock
count == 0? no			
count -= 1 (becomes 1)			
unlock			
	still waiting???		

Mesa-style monitors
signalling doesn't
"hand off" lock