CONCURRENCY: SEMAPHORES, DEADLOCK

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

- Project 3 is due Monday 3/11

- Midterm is next Wednesday 3/13 at 5.15pm, details on Piazza
- Discussion: Midterm review, Q&A → Mon
Tue at 5:30 pm Noland Hall

- Fill out mid semester course evaluation https://aefis.wisc.edu/

AGENDA / LEARNING OUTCOMES

Concurrency abstractions

- How to implement semaphores?
- What are common pitfalls with concurrent execution?

RECAP

CONCURRENCY OBJECTIVES

Mutual exclusion (e.g., A and B don't run at same time)

solved with *locks*

Ordering (e.g., B runs after A does something)

solved with *condition variables* and *semaphores*

SUMMARY: CONDITION VARIABLES

wait(cond_t *cv, mutex_t *lock)

- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)

- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return, doing nothing

wait
sleep()
release lock
acquire lock

signal
wakeup()

SUMMARY: RULES OF THUMB FOR CVS

done boolean

1. Keep state in addition to CV's

2. Always do wait/signal with lock held

3. Whenever thread wakes from waiting, recheck state

if (buffer.empty())
wait

while (buffer.empty())
wait

SEMAPHORE OPERATIONS

state

Allocate and Initialize

```
sem_t sem;
sem_init(sem_t *s, int initval) {
  s->value = initval;
}
```

User cannot read or write value directly after initialization

see impl slides behind

Wait or Test (sometime P() for Dutch) sem_wait(sem_t*)

Decrements sem value, Waits ~~until~~ if value of sem is >= 0

Signal or Post (sometime V() for Dutch) sem_post(sem_t*)

Increment sem value, then wake a single waiter

PRODUCER CONSUMER: EXAMPLE PIPES

A pipe may have many writers and readers

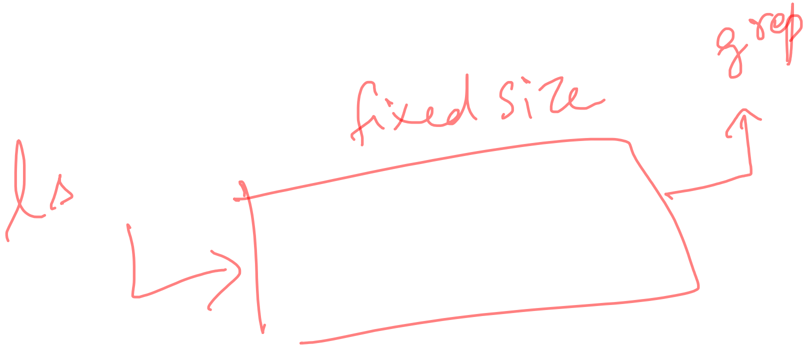
Internally, there is a finite-sized buffer

Writers add data to the buffer

- Writers have to wait if buffer is full

Readers remove data from the buffer

- Readers have to wait if buffer is empty



PRODUCER/CONSUMER: SEMAPHORES #1

Single producer thread, single consumer thread

Single shared buffer between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to ___1___
- fullBuffer: Initialize to ___0___

Producer

```
while (1) {

    sem_wait(&emptyBuffer);
    Fill(&buffer);

    sem_signal(&fullBuffer);
}
```

until this is > 0

signal consumers

Consumer

```
while (1) {

    sem_wait(&fullBuffer);
    Use(&buffer);

    sem_signal(&emptyBuffer);
}
```

	E	F
P	0	1
C	1	0

PRODUCER/CONSUMER: SEMAPHORES #2

buffer empty
size = 3

Single producer thread, single consumer thread

Shared buffer with **N elements** between producer and consumer

Use 2 semaphores

- emptyBuffer: Initialize to ____N____
- fullBuffer: Initialize to ____0____

	E	F
c1		-1
c2		-2
P1	2	-1
c1	3	

	E	F
P1	2	1
P2	1	2
P3	0	3
P4	-1	
C1	0	2
C2	1	1
D4	0	2

Producer

```
i = 0;
while (1) {
    sem_wait(&emptyBuffer);
    Fill(&buffer[i]);
    i = (i+1)%N;
    sem_signal(&fullBuffer);
}
```

→ circular

Consumer

```
j = 0;
While (1) {
    sem_wait(&fullBuffer);
    Use(&buffer[j]);
    j = (j+1)%N;
    sem_signal(&emptyBuffer);
}
```

PRODUCER/CONSUMER: SEMAPHORE #3

Final case:

- Multiple producer threads, multiple consumer threads
- Shared buffer with N elements between producer and consumer

Requirements

- Each consumer must grab unique filled element
- Each producer must grab unique empty element

PRODUCER/CONSUMER: MULTIPLE THREADS

P_1 0

P_2 1

Producer

```
while (1) {
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_signal(&fullBuffer);
}
```

$i = (i+1) \% N$

Consumer

```
while (1) {
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_signal(&emptyBuffer);
}
```

Are my_i and my_j private or shared? Where is mutual exclusion needed???

PRODUCER/CONSUMER: MULTIPLE THREADS

Does this work?

Empty buffer
Consumer grabs mutex
wait full buff

" lock
mutex"

```
Producer #1
    sem_wait(&mutex);
    sem_wait(&emptyBuffer);
    my_i = findempty(&buffer);
    Fill(&buffer[my_i]);
    sem_signal(&fullBuffer);
    sem_signal(&mutex);
```

```
Consumer #1
    sem_wait(&mutex);
    sem_wait(&fullBuffer);
    my_j = findfull(&buffer);
    Use(&buffer[my_j]);
    sem_signal(&emptyBuffer);
    sem_signal(&mutex);
```

PRODUCER/CONSUMER: MULTIPLE THREADS

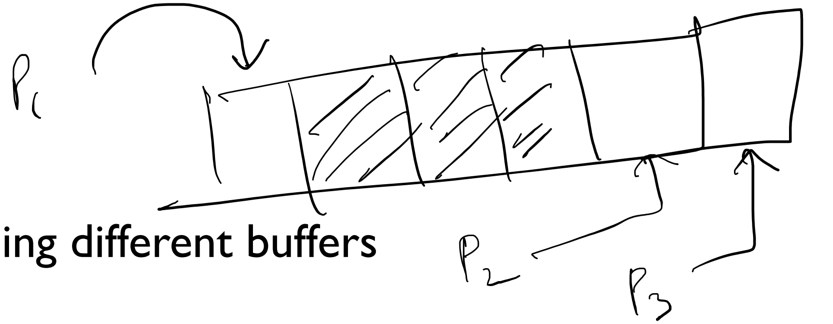
```
Producer #2
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    Fill(&buffer[myi]);
    sem_signal(&mutex);
    sem_signal(&fullBuffer);
```

critical section

```
Consumer #2
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    Use(&buffer[myj]);
    sem_signal(&mutex);
    sem_signal(&emptyBuffer);
```

critical section

3 Producers



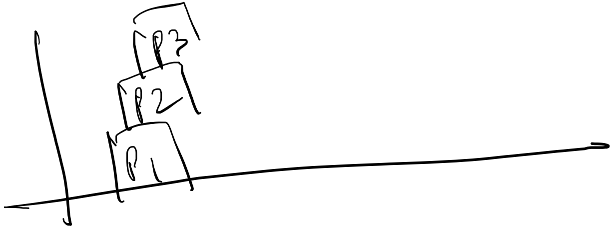
Works, but limits concurrency:
Only 1 thread at a time can be using or filling different buffers

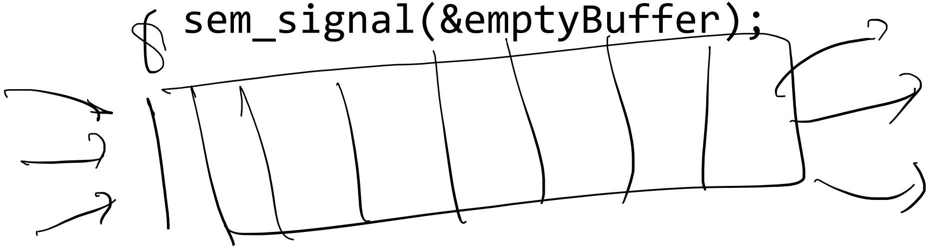
PRODUCER/CONSUMER: MULTIPLE THREADS

P1 0 my_i
0 my_j
C1 0 my_i
P2 0 my_j
C2 0

```
Producer #3
    sem_wait(&emptyBuffer);
    sem_wait(&mutex);
    myi = findempty(&buffer);
    sem_signal(&mutex);
    Fill(&buffer[myi]);
    sem_signal(&fullBuffer);
```

```
Consumer #3
    sem_wait(&fullBuffer);
    sem_wait(&mutex);
    myj = findfull(&buffer);
    sem_signal(&mutex);
    Use(&buffer[myj]);
    sem_signal(&emptyBuffer);
```





Works and increases concurrency; only finding a buffer is protected by mutex;
Filling or Using different buffers can proceed concurrently

READER/WRITER LOCKS

Let multiple reader threads grab lock (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Let us see if we can understand code

Data structures

OK for multi readers

1 writer

READER/WRITER LOCKS

```
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 1);
    sem_init(&rw->writelock, 1);
}
```

"reader"

initialize lock to 1

similar to mutex init

READER/WRITER LOCKS

```
void rwlock_acquire_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);
      rw->readers++;
      if (rw->readers == 1)
          sem_wait(&rw->writelock);
      sem_post(&rw->lock);
}
void rwlock_release_readlock(rwlock_t *rw) {
      sem_wait(&rw->lock);
      rw->readers--;
      if (rw->readers == 0)
          sem_post(&rw->writelock);
      sem_post(&rw->lock);
}
rwlock_acquire_writelock(rwlock_t *rw) {  sem_wait(&rw->writelock); }
rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

let other writers

	L	WL	Readers
	1	1	0
T1.readL	∅ 0	0	1
T2.readL	∅ 1		2
T3. writeL		-1	
T1.unlock	∅ 1		1
T2.unlock	∅ 1	0	0

↳ T3 signal

mutex


BUNNY
https://tinyurl.com/cs537-sp19-bunny8

READER WRITER LOCKS

tinyurl.com/cs537-sp19-bunny8

T1: acquire_readlock() → RUNNING
T2: acquire_readlock() → RUNNING
T3: acquire_writelock() → BLOCKED WRITE LOCK

What is the status of T2 ?
RUNNING

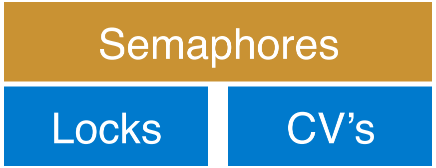
T6: acquire_writelock() WRITE LOCK & RUNNING
T4: acquire_readlock() WAITING FOR READ LOCK
T5: acquire_readlock() WAITING FOR READ LOCK

What is the status of T4?

BUILD SEMAPHORE FROM LOCK AND CV

```
Typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```



sem_wait(): ~~Decrement and~~ waits until value >~~=~~ 0, decrement

sem_post(): Increment value, then wake a single waiter

BUILD SEMAPHORE FROM LOCK AND CV

Acquire lock

```
sem_wait(sem_t *s) {
    lock_acquire(&s->lock);
    s->value--;
    while (s->value <= 0)
        cond_wait(&s->cond);
    lock_release(&s->lock);
}
```

checking if value is positive

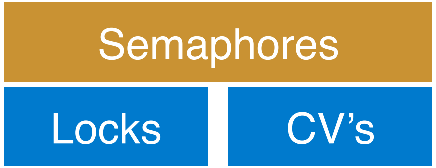
s->value--;

```
sem_post(sem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

wakes up single waiter

Locks using semaphore

Semaphore using Locks + CV => equally powerful



sem_wait(): ~~Decrement and~~ waits until value >~~=~~ 0, decrement

sem_post(): Increment value, then wake a single waiter

SUMMARY: SEMAPHORES

Semaphores are equivalent to locks + condition variables

– Can be used for both mutual exclusion and ordering

Semaphores contain **state**

– How they are initialized depends on how they will be used

– Init to 0: Join (1 thread must arrive first, then other)

– Init to N: Number of available resources

sem_wait(): Decrement and waits until value >= 0

sem_post(): Increment value, then wake a single waiter (atomic)

Can use semaphores in producer/consumer and for reader/writer locks

CONCURRENCY BUGS

CONCURRENCY IN MEDICINE: THERAC-25 (1980'S)

"The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a race condition."

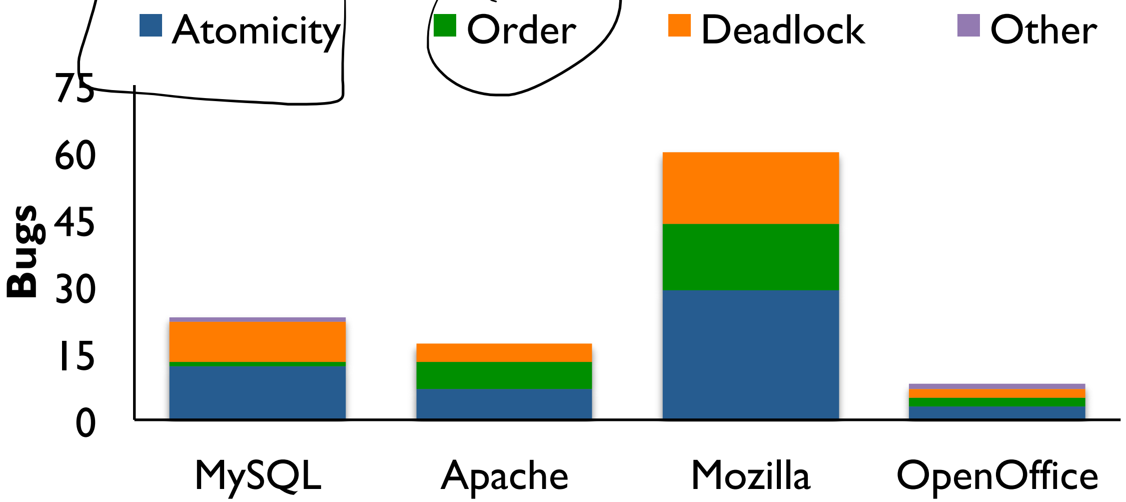
"…in three cases, the injured patients later died."

Source: http://en.wikipedia.org/wiki/Therac-25

CONCURRENCY STUDY

Mutual exclusion

ordering



Database

Lu *etal.* [ASPLOS 2008]:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

ATOMICITY: MYSQL

if not null

Thread 1:
```
if (thd->proc_info) {
   …
   fputs(thd->proc_info, …);
   …
}
```

Thread 2:
```
thd->proc_info = NULL;
```

T1 reads not null

T2 sets it to null

T1 passes in NULL!

What's wrong?

FIX ATOMICITY BUGS WITH LOCKS

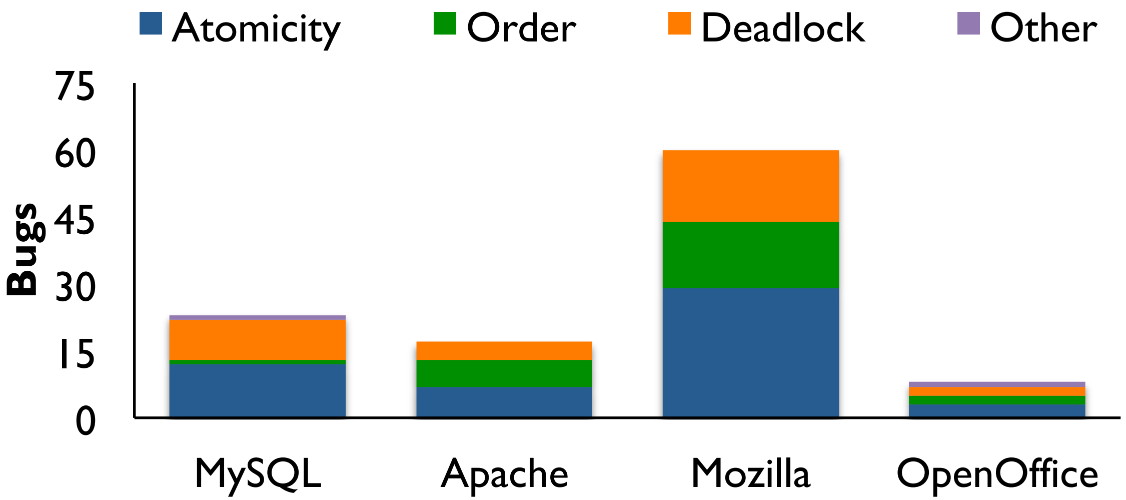
Thread 1:

```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
    …
    fputs(thd->proc_info, …);
    …
}
pthread_mutex_unlock(&lock);
```

Thread 2:

```
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

CONCURRENCY STUDY



Lu *etal.* [ASPLOS 2008]:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

ORDERING: MOZILLA

Thread 1:

```
void init() {
  …
  mThread =
    PR_CreateThread(mMain, …);
  …
}
```

T1 runs before
T2 mThread→

Thread 2:

```
void mMain(…) {
  …
  mState = mThread->State;
  …
}
```

T2 runs before T1
T2 mthread→state
but mThread not init

What's wrong?

FIX ORDERING BUGS WITH CONDITION VARIABLES

Thread 1:

```
void init() {
    …

    mThread =
    PR_CreateThread(mMain, …);

    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);

    …
}
```

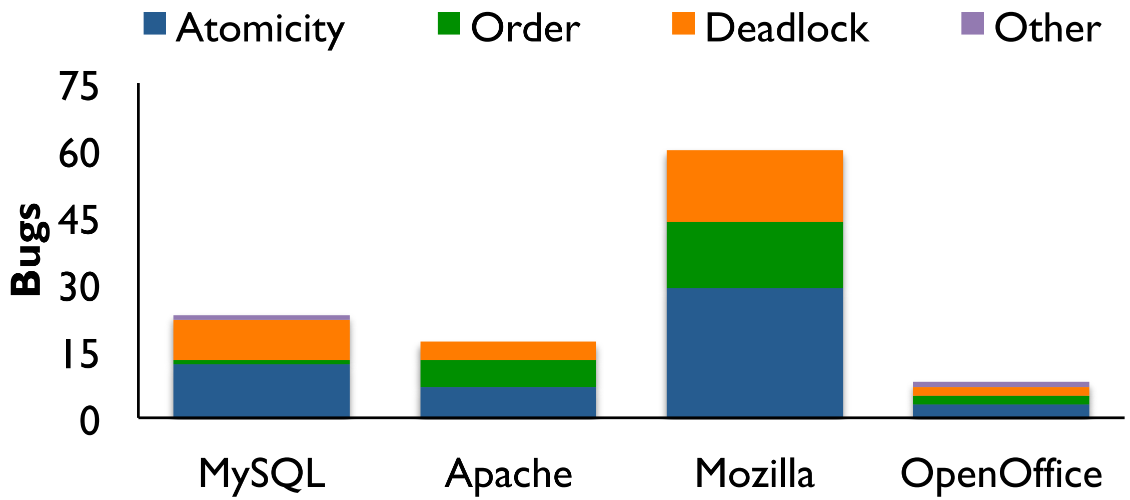
Thread 2:

```
void mMain(…) {
    …

    mutex_lock(&mtLock);
    while (mtInit == 0)
      Cond_wait(&mtCond, &mtLock);
    Mutex_unlock(&mtLock);

    mState = mThread->State;
    …
}
```

CONCURRENCY STUDY



Lu *etal.* [ASPLOS 2008]:

For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

DEADLOCK

No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

CODE EXAMPLE

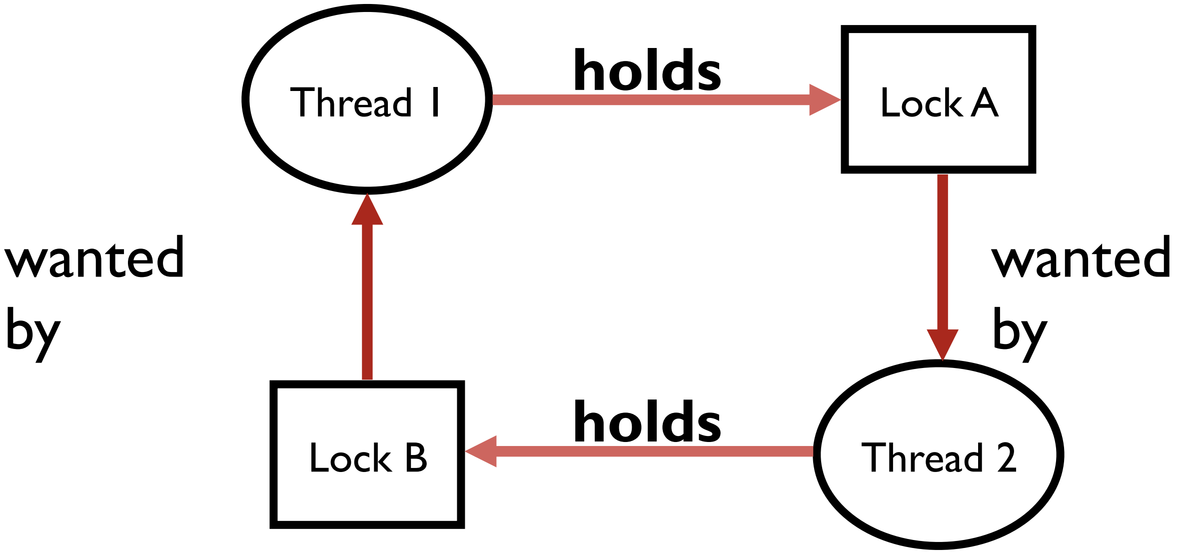
Thread 1:

```
lock(&A);
lock(&B);
```

Thread 2:

```
lock(&B);
lock(&A);
```

CIRCULAR DEPENDENCY



FIX DEADLOCKED CODE

```
Thread 1:

lock(&A);
lock(&B);
```

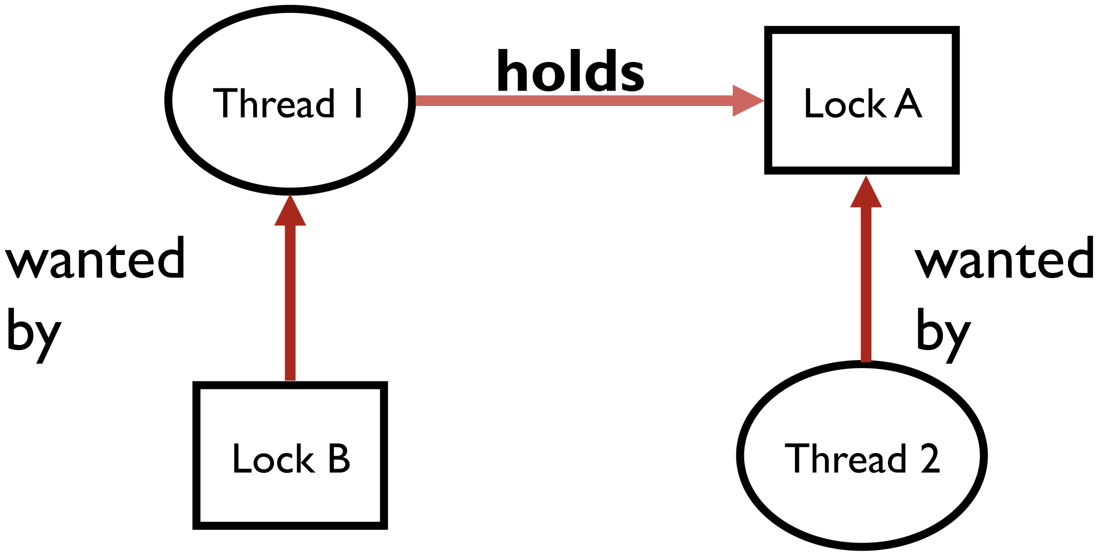
```
Thread 2:

lock(&B);
lock(&A);
```

```
Thread 1
```

```
Thread 2
```

NON-CIRCULAR DEPENDENCY



```c
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = malloc(sizeof(*rv));
    mutex_lock(&s1->lock);
    mutex_lock(&s2->lock);
    for(int i=0; i<s1->len; i++) {
        if(set_contains(s2, s1->items[i])
            set_add(rv, s1->items[i]);
    mutex_unlock(&s2->lock);
    mutex_unlock(&s1->lock);
}
```

```
Thread 1: rv = set_intersection(setA, setB);
Thread 2: rv = set_intersection(setB, setA);
```

ENCAPSULATION

Modularity can make it harder to see deadlocks

Solution?

```
if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

Any other problems?

DEADLOCK THEORY

Deadlocks can only happen with these four conditions:

1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Can eliminate deadlock by eliminating any one condition

1. MUTUAL EXCLUSION

Problem: Threads claim exclusive control of resources that they require

Strategy: Eliminate locks!

Try to replace locks with atomic primitive:

```
int CompAndSwap(int *addr, int expected, int new)
Returns 0 fail, 1 success
```

BUNNY

```
void add (int *val, int amt)
{
    Mutex_lock(&m);
    *val += amt;
    Mutex_unlock(&m);
}
```

```
void add (int *val, int amt) {
    do {
        int old = *value;
    } while(!CompAndSwap(val, ___, old+amt);
}
```

WAIT-FREE ALGORITHM: LINKED LIST INSERT

```
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    lock(&m);
    n->next = head;
    head = n;
    unlock(&m);
}
```

```
void insert (int val) {
    node_t *n = Malloc(sizeof(*n));
    n->val = val;
    do {
        n->next = head;
    } while (!CompAndSwap(&head,
                  n->next, n));
}
```

2. HOLD-AND-WAIT

Problem: Threads hold resources allocated to them while waiting for additional resources

Strategy: Acquire all locks atomically **once.** Can release locks over time, but cannot acquire again until all have been released

How to do this? Use a meta lock:

Disadvantages?

3. NO PREEMPTION

Problem: Resources (e.g., locks) cannot be forcibly removed from threads that are

Strategy: if thread can't get what it wants, release what it holds

```
top:
    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }
    …
```

Disadvantages?

4. CIRCULAR WAIT

Circular chain of threads such that each thread holds a resource (e.g., lock) being requested by next thread in the chain.

Strategy:

- decide which locks should be acquired before others
- if A before B, never acquire A if B is already held!
- document this, and write code accordingly

Works well if system has distinct layers

NEXT STEPS

Project 3: Out now!

Midterm details posted

Next class: Midterm review