CONCURRENCY: READER/WRITER LOCKS + DEADLOCK

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Project 4 turned in – no significant problems
- Project 5 available now (xv6 Memory)
  - Greatly simplified! ☺
  - Due next Monday 11/4 (5pm)
  - Request new project partner if desired (web form)
- Midterm 2: Nov 11/6 (Wed) from 7:30-9:30pm
  - Two "quizzes" on race conditions in Canvas
  - Next lecture some review, some sample problems

AGENDA / LEARNING OUTCOMES

Concurrency abstractions

- How to implement reader/writer locks with semaphores?
- What types of concurrency bugs occur?
- How to fix **atomicity bugs** (with locks)?
- How to fix **ordering bugs** (with condition variables)?
- How does **deadlock** occur?
- How to prevent deadlock (with waitfree algorithms, grab all locks atomically, trylocks, and ordering across locks)?

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*

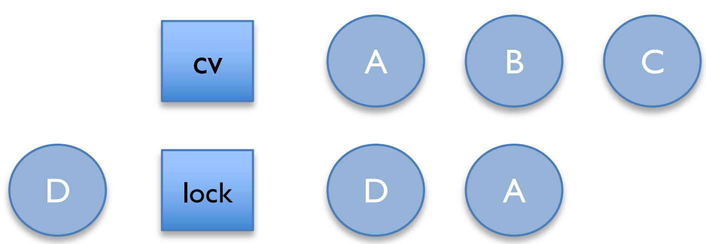
CONDITION VARIABLES

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



signal(cv) - what happens?

release(lock) - what happens?

INTRODUCING SEMAPHORES

Condition variables have no state (other than waiting queue)

- Programmer must track additional state

Semaphores have state: track integer value

- State cannot be directly accessed by user program, but state determines behavior of semaphore operations

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

DINING PHILOSOPHERS

Problem Statement

- **N** Philosophers sitting at a round table
- Each philosopher shares a chopstick (or fork) with neighbor
- Each philosopher must have both chopsticks to eat
- Neighbors can't eat simultaneously
- Philosophers alternate between thinking and eating

Each philosopher/thread **i** runs :

```
while (1) {
    think();
    take_chopsticks(i);
    eat();
    put_chopsticks(i);
  }
```

DINING PHILOSOPHERS: ATTEMPT #1

Two neighbors can't use chopstick at same time

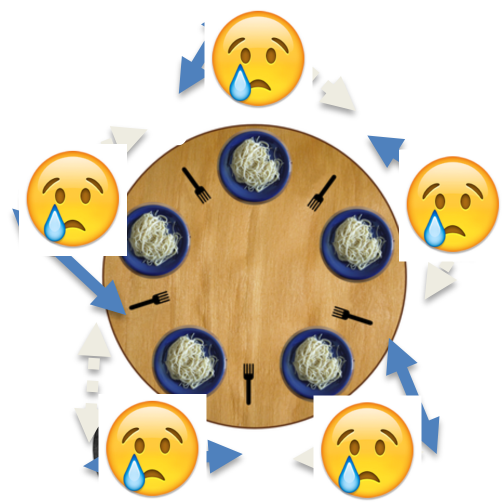
Must test if chopstick is there and grab it atomically

- Represent each chopstick with a semaphore
- Grab right chopstick then left chopstick

Deadlocked!

Code for 5 philosophers:

```
sem_t chopstick[5]; // Initialize each to 1
take_chopsticks(int i) {
    wait(&chopstick[i]);
    wait(&chopstick[(i+1)%5]);
}
put_chopsticks(int i) {
    signal(&chopstick[i]);
    signal(&chopstick[(i+1)%5]);
}
```



DINING PHILOSOPHERS

- Two solutions
  - Having one philosopher grab resource in other order; Break circular dependencies
  - Phrase in terms of safety and liveness criteria; Don't hold on to one resource while waiting for next

DINING PHILOSOPHERS: ATTEMPT #2

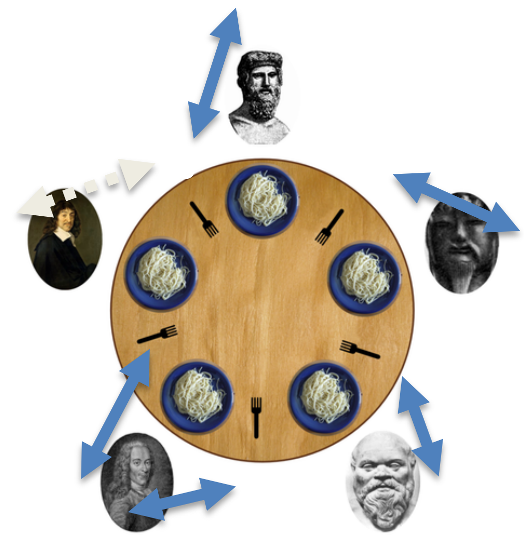
Grab lower-numbered chopstick first, then higher-numbered

```
sem_t chopstick[5]; // Initialize to 1
take_chopsticks(int i) {
  if (i < 4) {
      wait(&chopstick[i]);
      wait(&chopstick[i+1]);
  } else {
      wait(&chopstick[0]);
      wait(&chopstick[4]);
}
```

Philosopher 3 finishes take_chopsticks() and eventually calls put_chopsticks();

Who can run then?

What is wrong with this solution???



```
sem_t mayEat[5]; // how to initialize?
sem_t mutex;     // how to init?
int state[5] = {THINKING};
take_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = HUNGRY;
  testSafetyAndLiveness(i); // check if I can run
  signal(&mutex); // exit critical section
  wait(&mayEat[i]);
}
put_chopsticks(int i) {
  wait(&mutex); // enter critical section
  state[i] = THINKING;
  test(i+1 %5); // check if neighbor can run now
  test(i+4 %5);
  signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
  if(state[i]==HUNGRY&&state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
      state[i] = EATING;
      signal(&mayEat[i]);
  } }
```

READER/WRITER LOCKS

Protect shared data structure; Goal:

Let multiple reader threads grab lock with other readers (shared)

Only one writer thread can grab lock (exclusive)

- No reader threads
- No other writer threads

Two possibilities for priorities – different implementations

1) No reader waits unless writer in critical section
    - How can writers starve?
2) No writer waits longer than absolute minimum
    - How can readers starve?

Let us see if we can understand code…

VERSION 1

Readers have priority

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/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); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T2: release_readlock()
T1: release_readlock()
    // who runs?
T4: acquire_readlock()
    // what happens?
T5: acquire_readlock()
    // where blocked?
T3: release_writelock()
    // what happens next?

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); }
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: acquire_readlock()
// what happens?

VERSION 2

Writers have priority
Three semaphores

- Mutex
- OKToRead
- OKToWrite
- Semaphores used in similar way to myEat[i] in Dining Philosphers, but one for all readers, one for all writers)

How to initialize?
Shared variables

Waiting Readers,
ActiveReaders
WaitingWriters
ActiveWriters

```
Acquire_readlock() {
   Sem_wait(&mutex);
   If (ActiveWriters +
      WaitingWriters==0) {
         sem_post(OKToRead);
         ActiveReaders++;
   } else WaitingReaders++;
   Sem_post(&mutex);
   Sem_wait(OKToRead);
}
Release_readlock() {
   Sem_wait(&mutex);
   ActiveReaders--;
   If (ActiveReaders==0 &&
     WaitingWriters > 0) {
         ActiveWriters++;
         WaitingWriters--;
         Sem_post(OKToWrite);
   }
   Sem_post(&mutex);
}
```

```
Acquire_writelock() {
   Sem_wait(&mutex);
   If (ActiveWriters + ActiveReaders + WaitingWriters==0) {
      ActiveWriters++;
      sem_post(OKToWrite);
   } else WaitingWriters++;
   Sem_post(&mutex);
   Sem_wait(OKToWrite);
}
```

T1: acquire_readlock()
T2: acquire_readlock()
T3: acquire_writelock()
T4: acquire_readlock()
// what happens?
… release_readlock() x 2
T3: release_writelock()

```
Release_writelock() {
   Sem_wait(&mutex);
   ActiveWriters--;
   If (WaitingWriters > 0) {
      ActiveWriters++;
      WaitingWriters--;
      Sem_post(OKToWrite);
   } else while(WaitingReaders>0) {
      ActiveReaders++;
      WaitingReaders--;
      sem_post(OKToRead);
   }
   Sem_post(&mutex);
```

CONCURRENCY BUGS

CORRECTNESS IS IMPORTANT

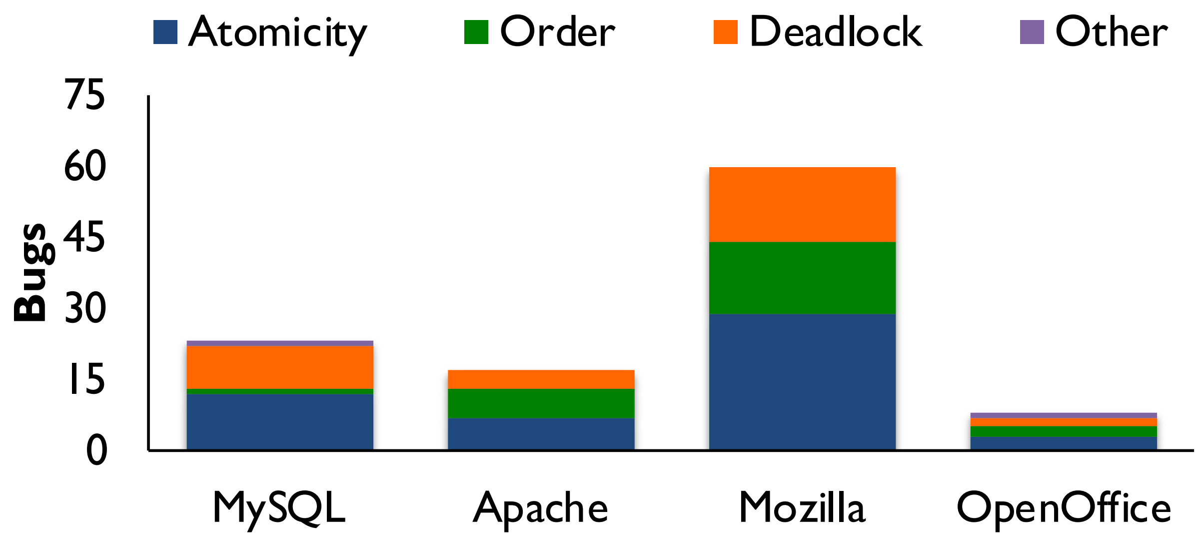
Concurrency in Medicine: Therac-25 (1980's)

"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



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

Thread 1:

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

Thread 2:

```
thd->proc_info = NULL;
```

What's wrong?

FIX ATOMICITY BUGS WITH LOCKS

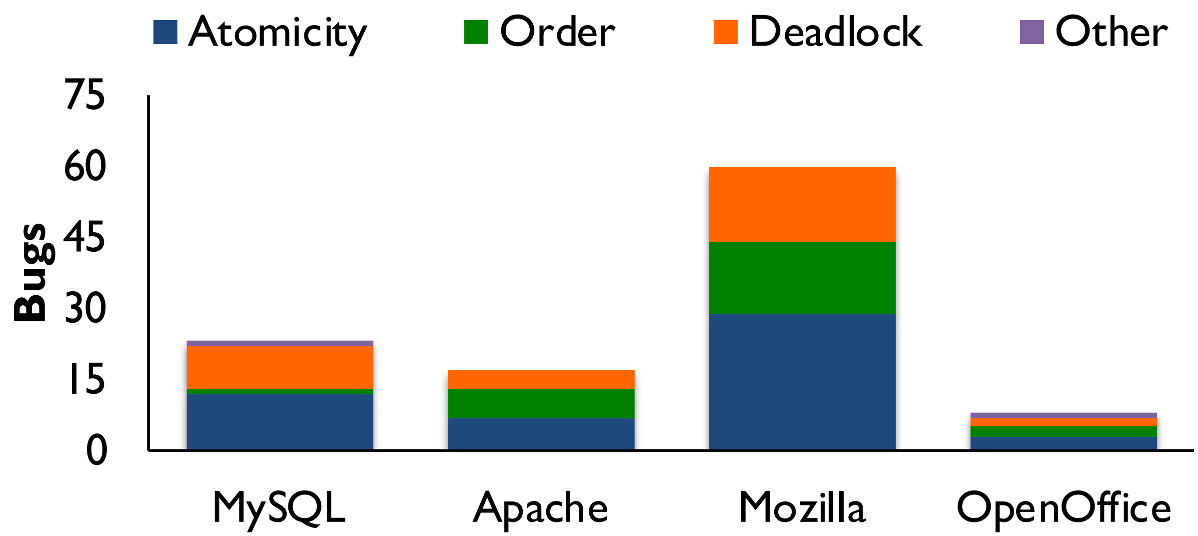
Thread 1:

```
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, …);
  …
}
```

Thread 2:

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

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);

    …
}
```

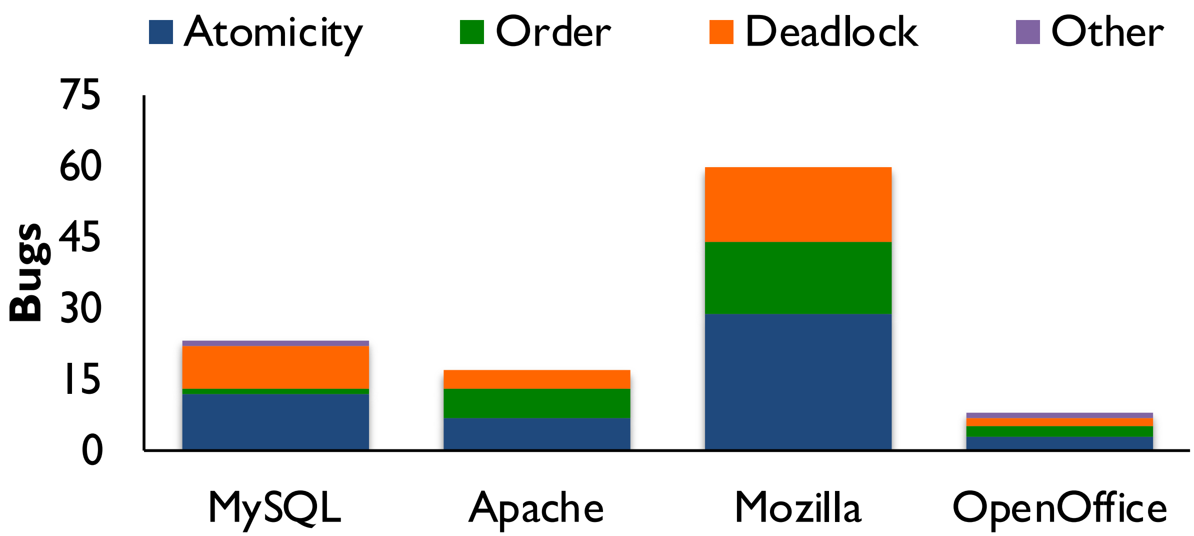
Thread 2:

```
void mMain(…) {
  …

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

  mState = mThread->State;
  …
}
```

CONCURRENCY STUDY

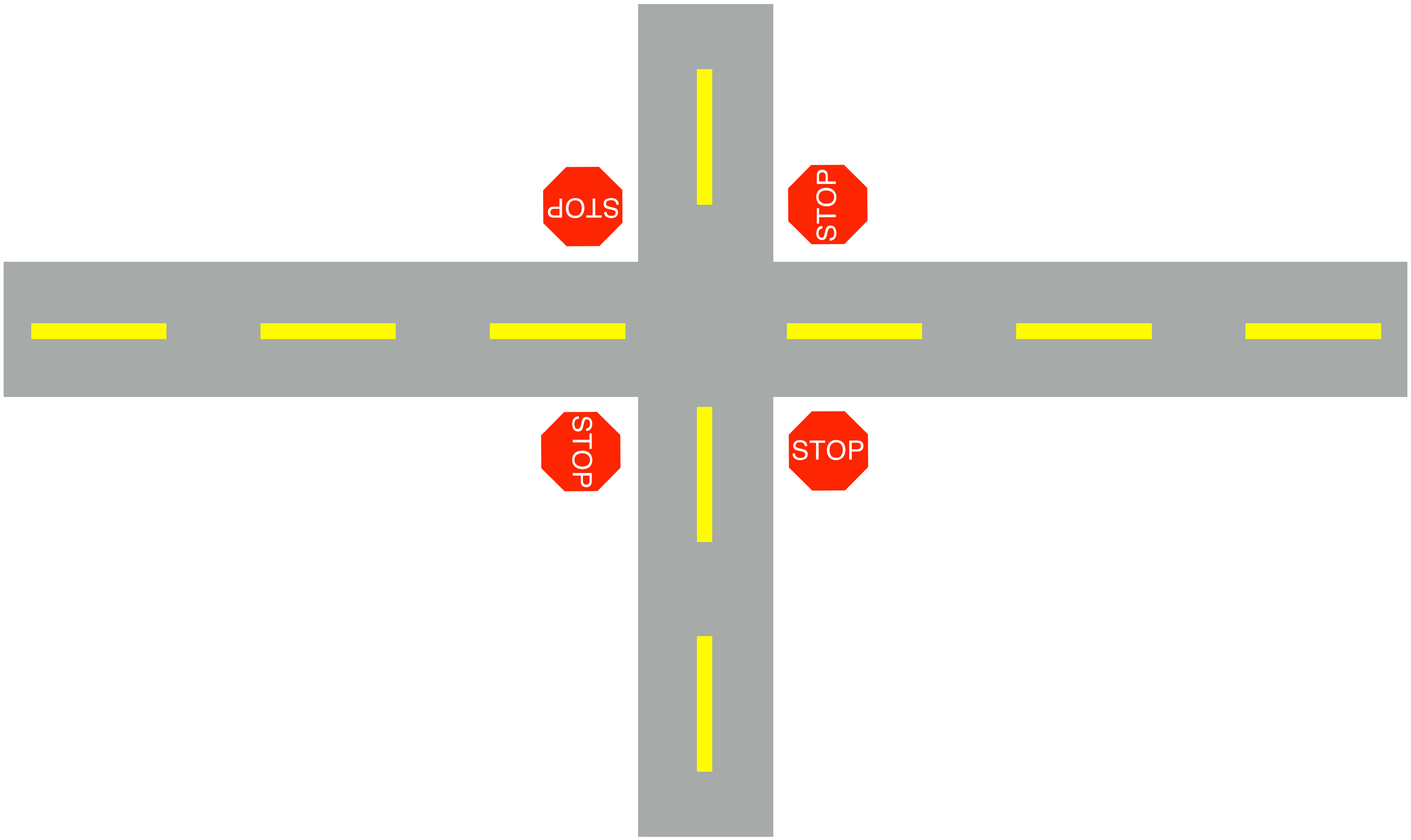


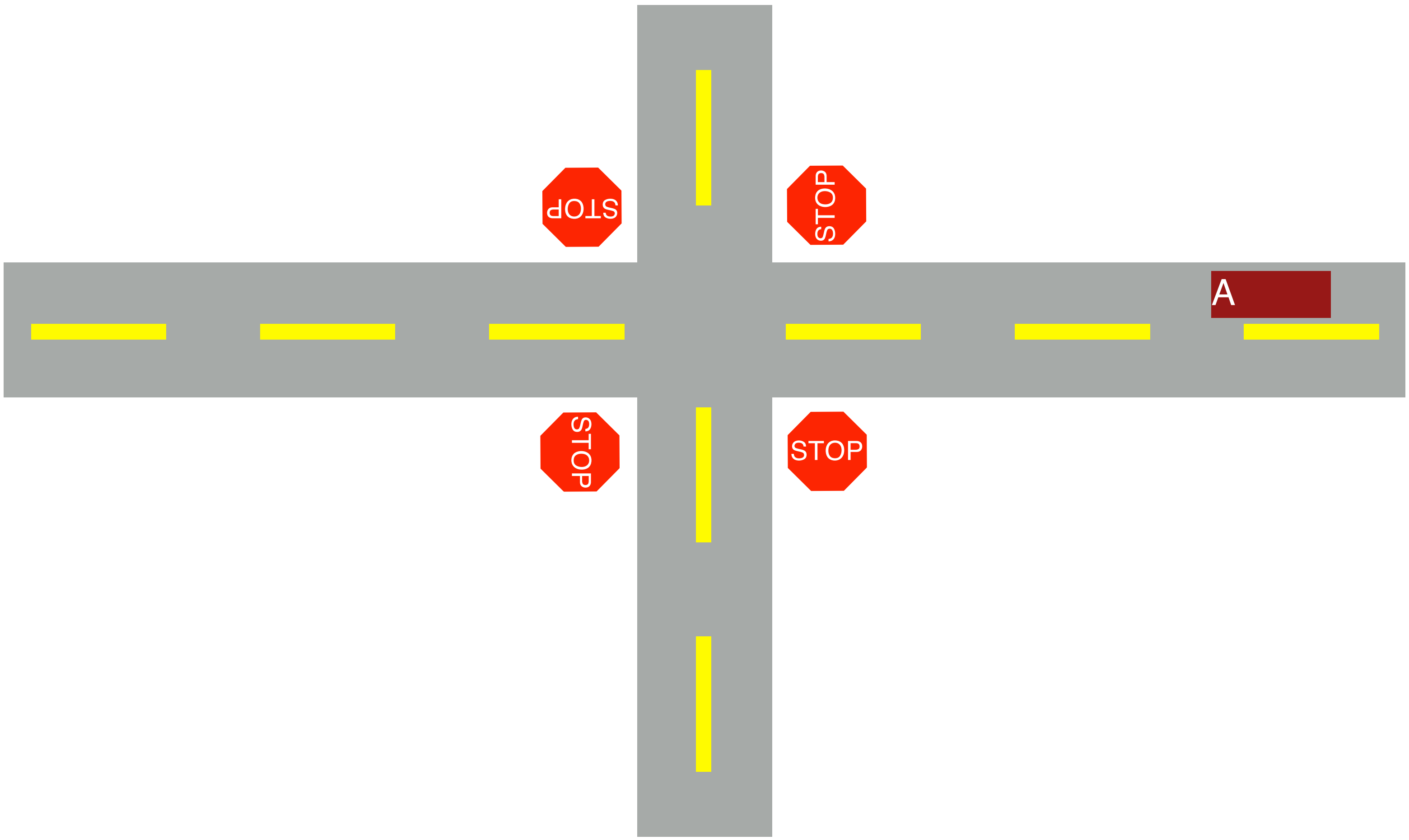
Lu *etal.* [ASPLOS 2008]:

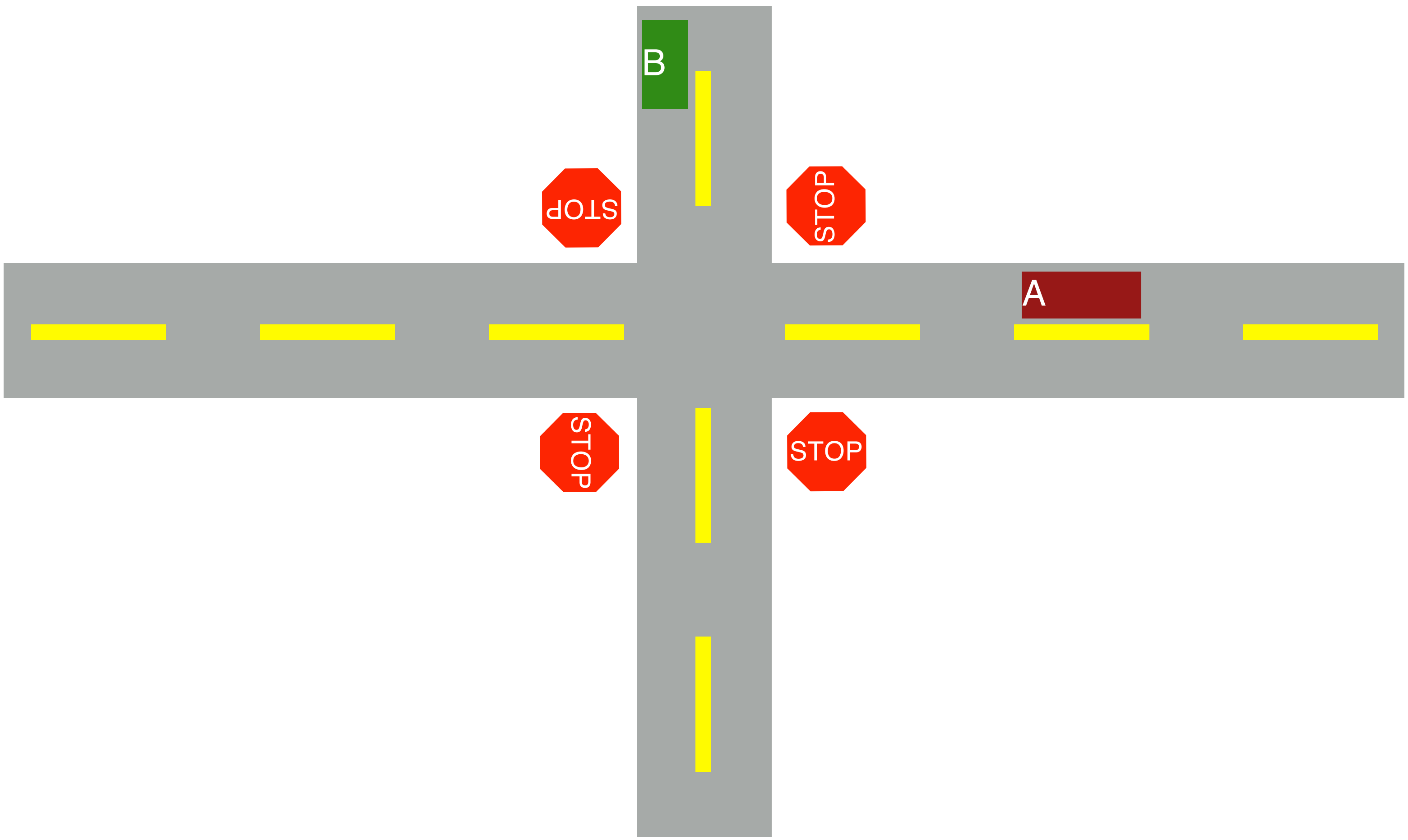
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

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


STOP
STOP
STOP
STOP

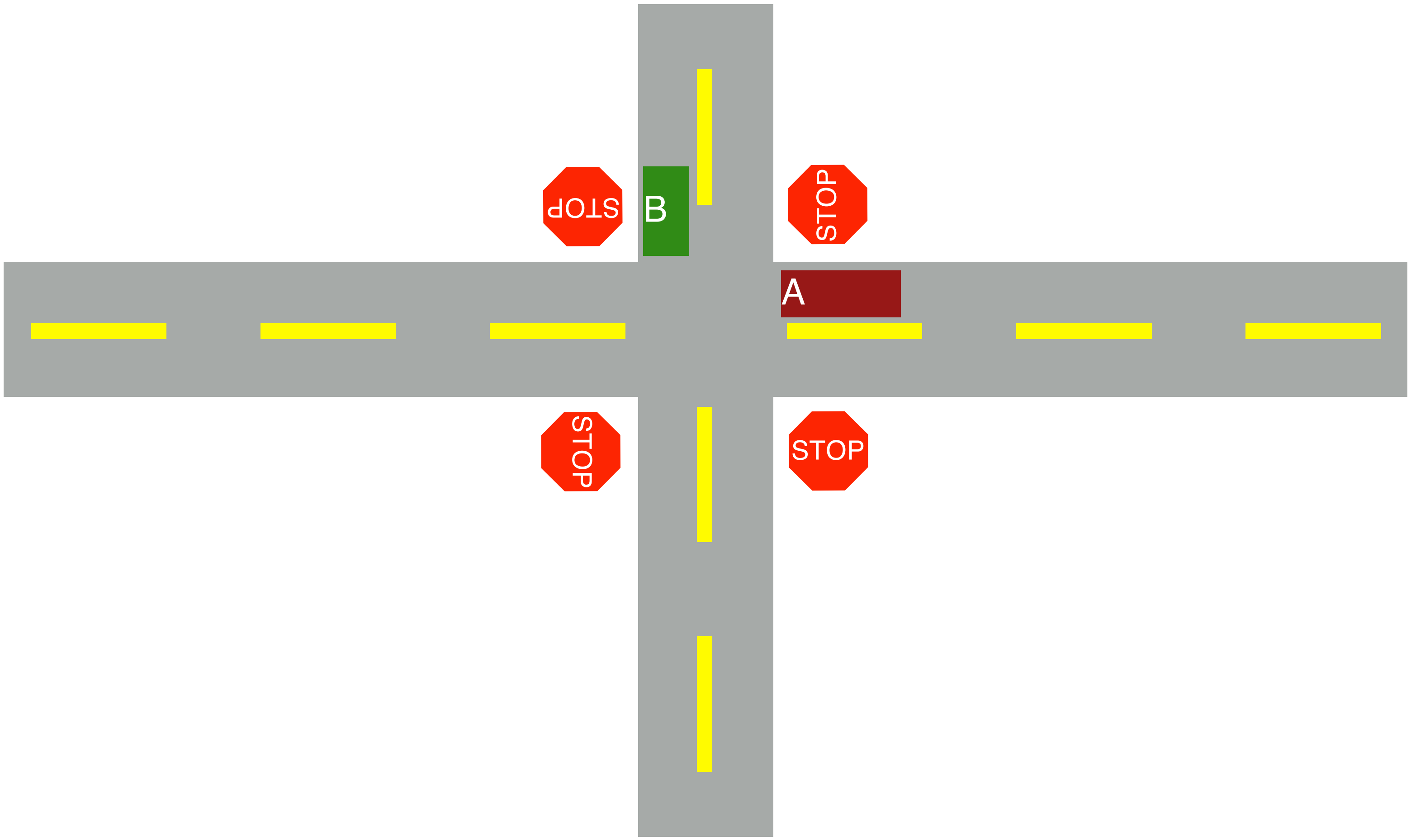

STOP
STOP
A
STOP
STOP

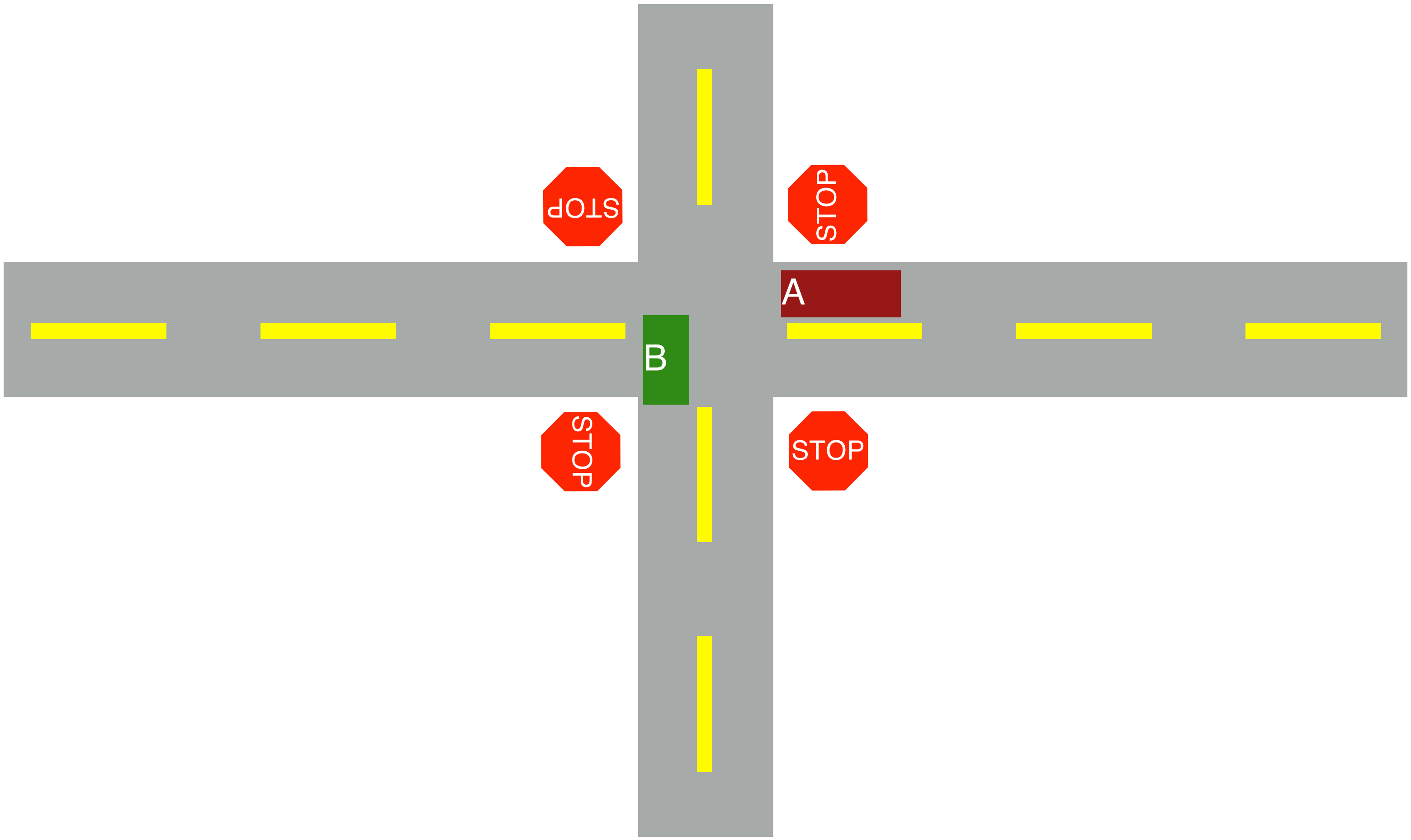


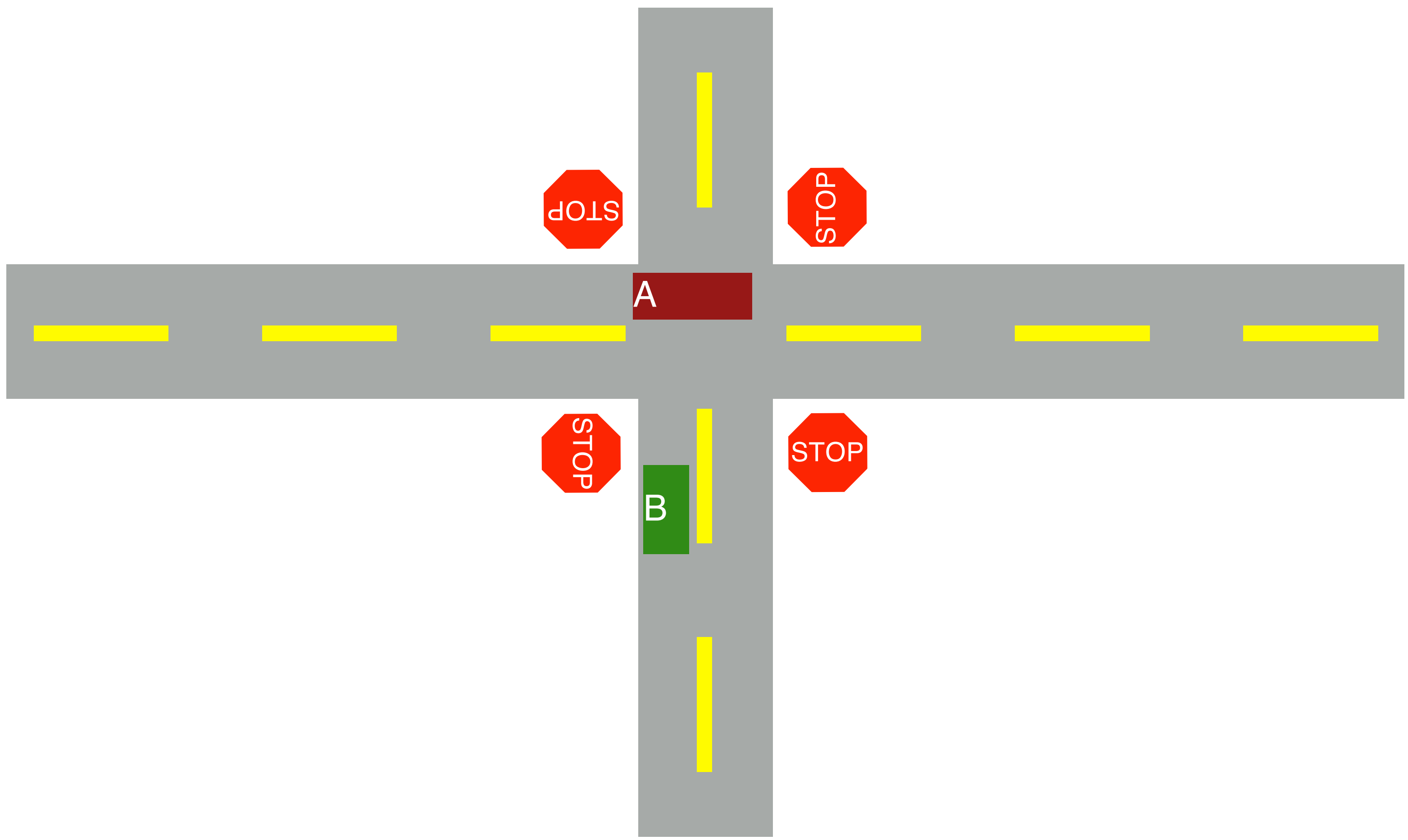
B
STOP
STOP
A
STOP
STOP

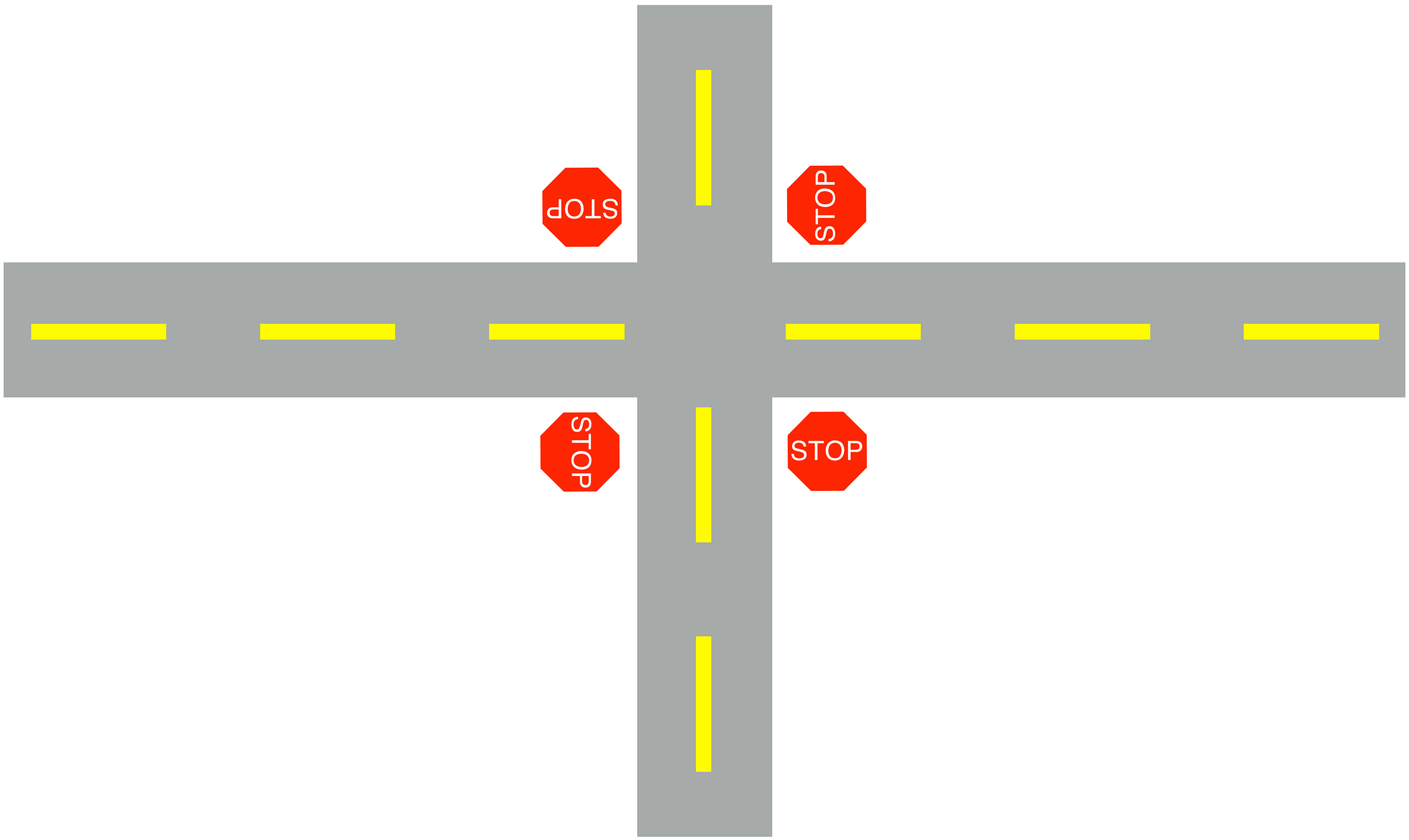
Both cars arrive at same time

Is this deadlocked?

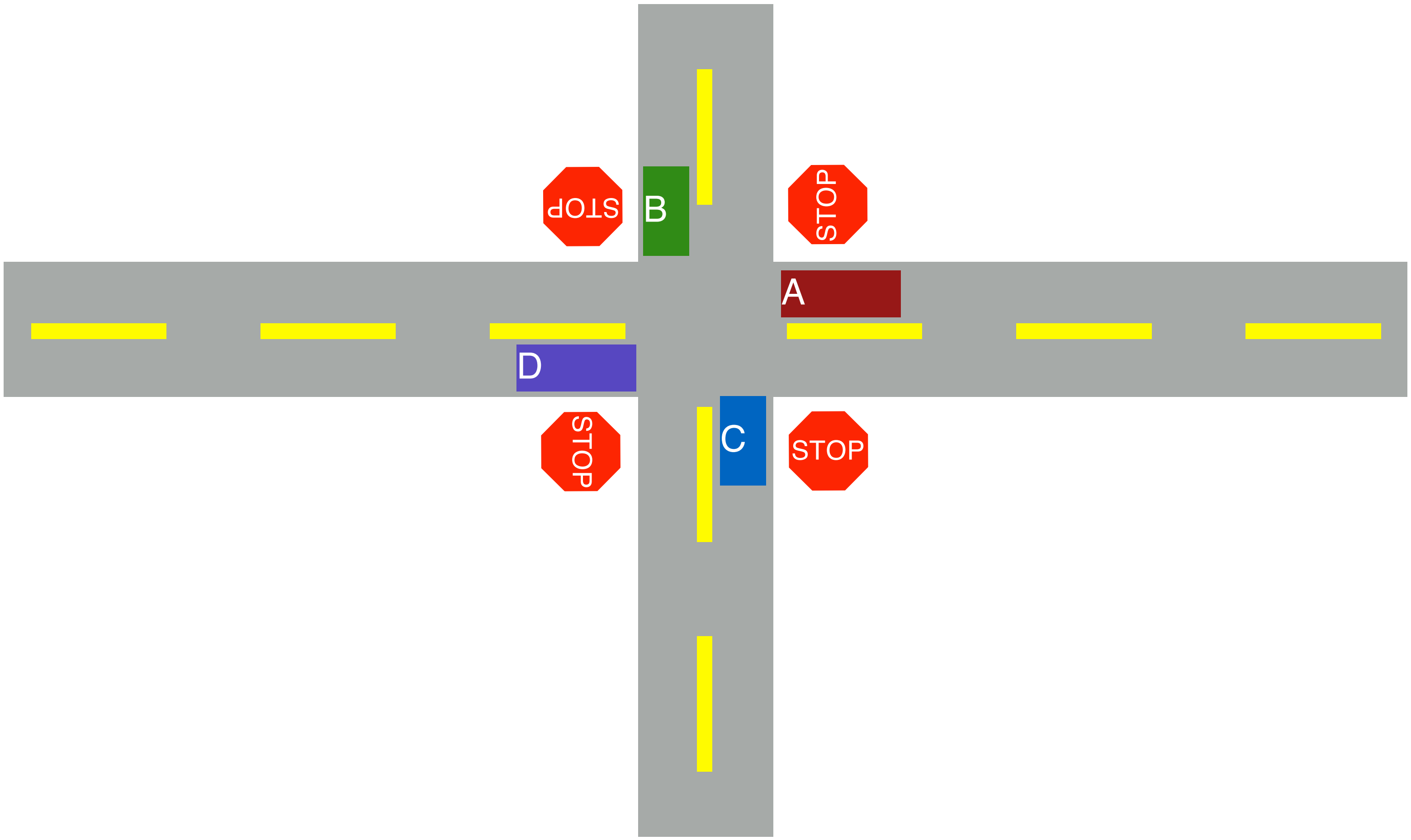



STOP
STOP
A
B
STOP
STOP


STOP
STOP
A
STOP
STOP
B

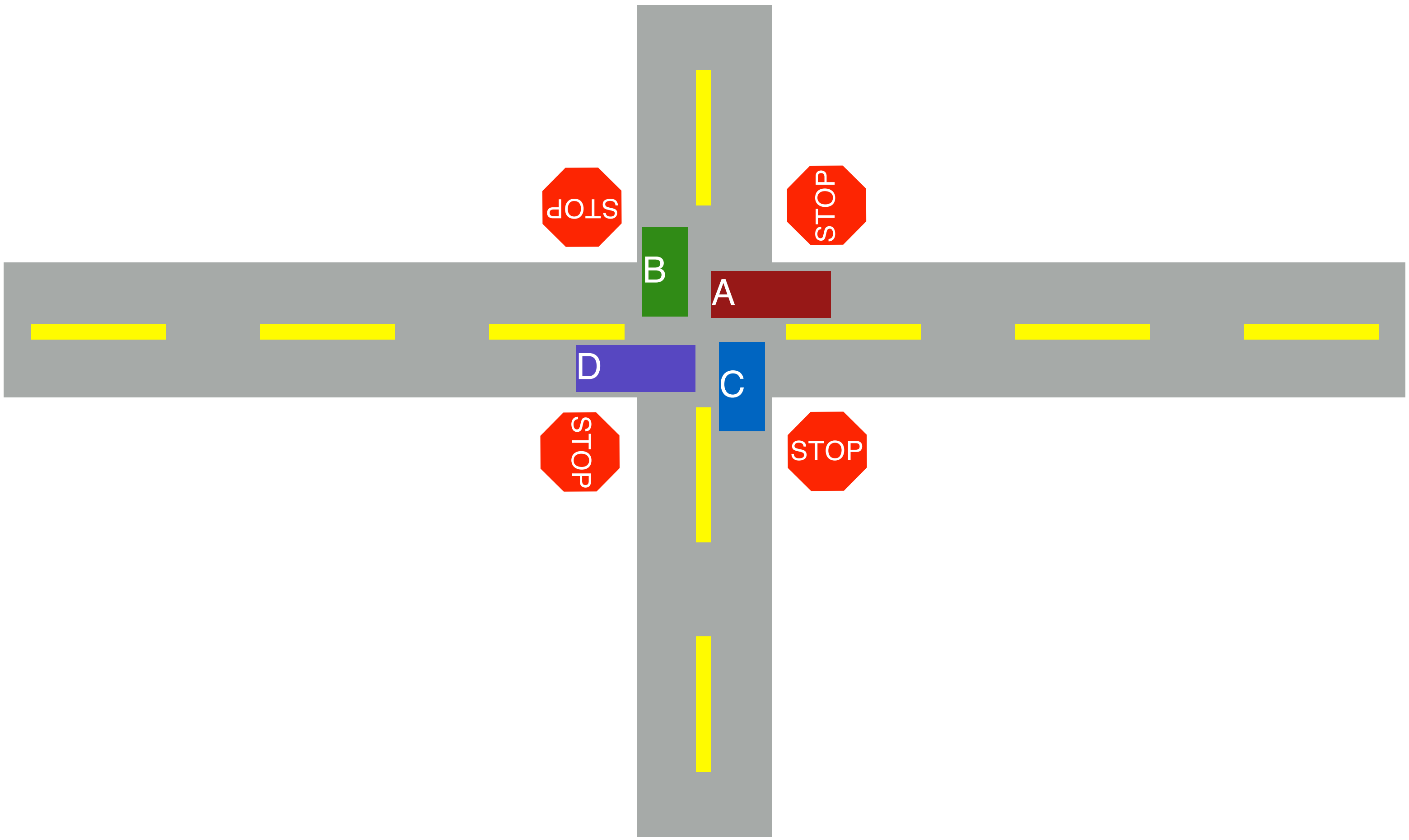

STOP
STOP
STOP
STOP

4 cars arrive at same time
Is this deadlocked?

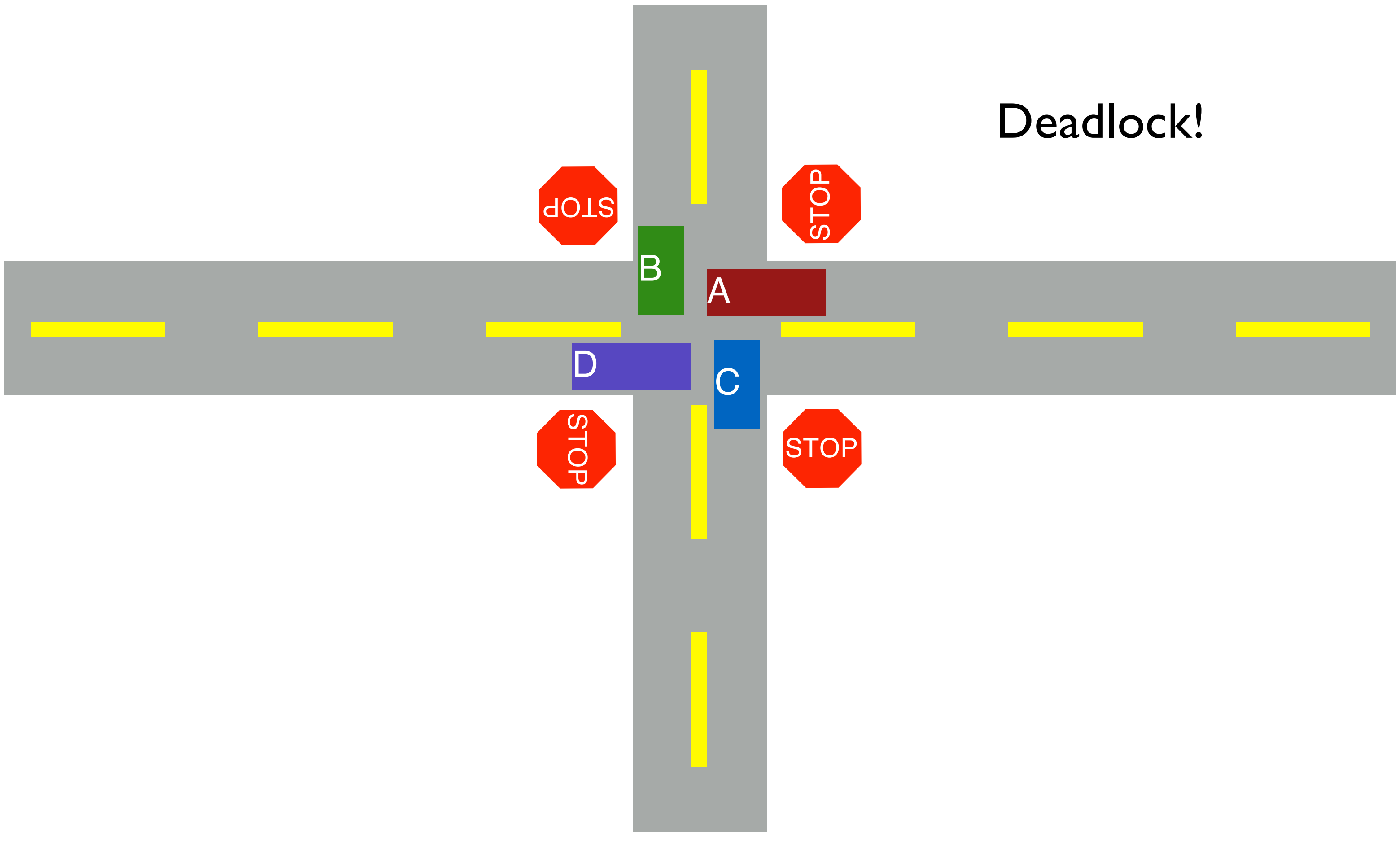


4 cars move forward same time

Is this deadlocked?



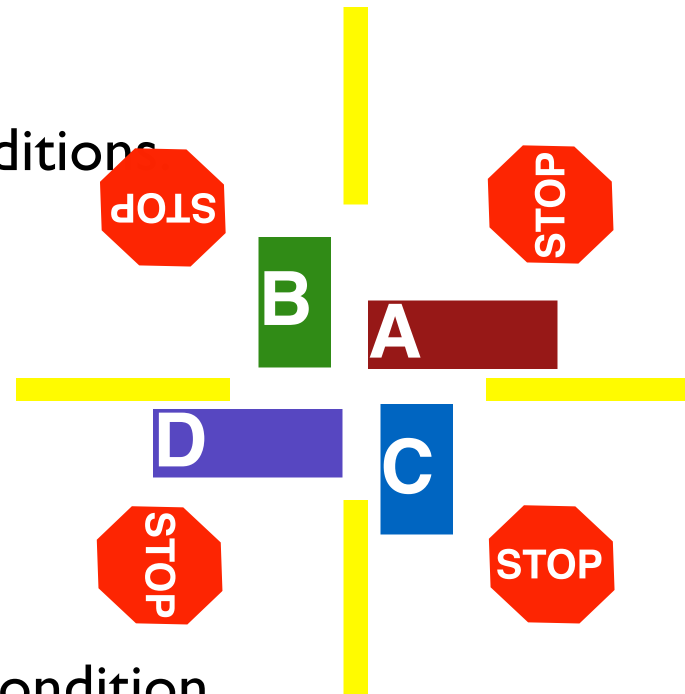
Deadlock!



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

CODE EXAMPLE

Thread 1:

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

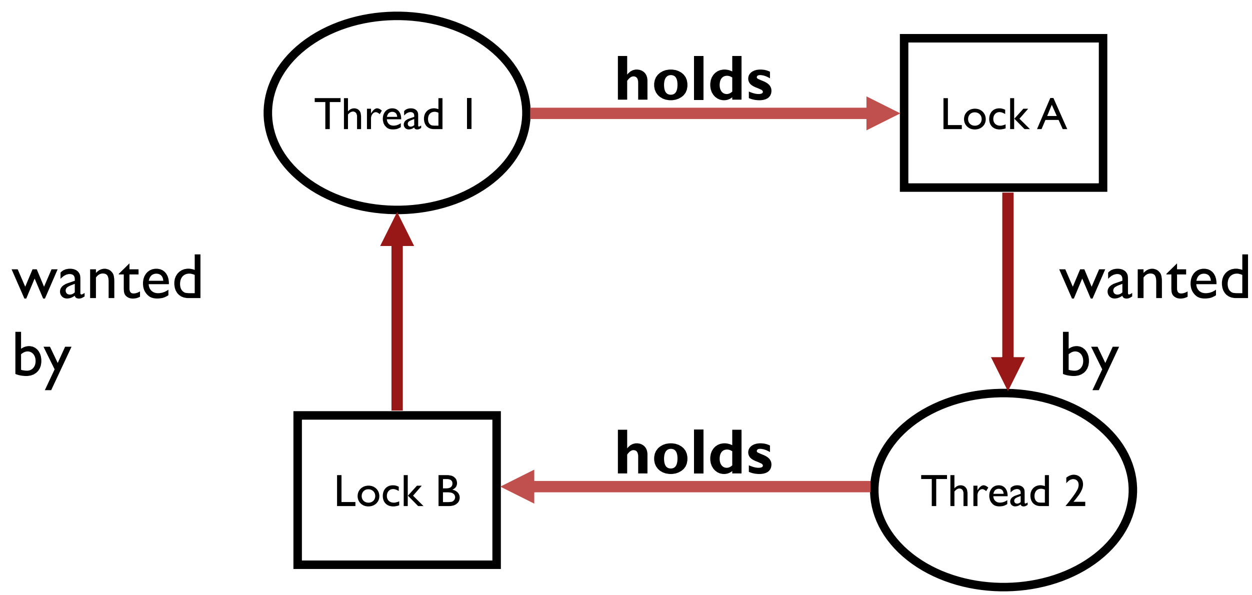
Thread 2:

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

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

Can deadlock happen with these two threads?

CIRCULAR DEPENDENCY



FIX DEADLOCKED CODE

Thread 1:

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

Thread 2:

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

How would you fix this code?

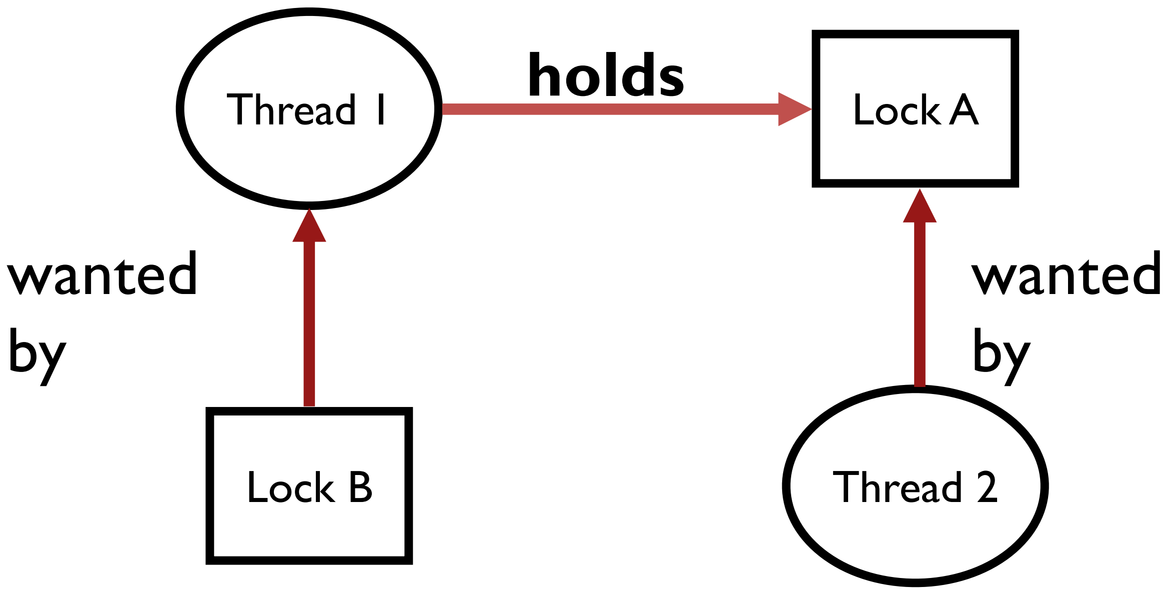
Thread 1

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

Thread 2

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

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?

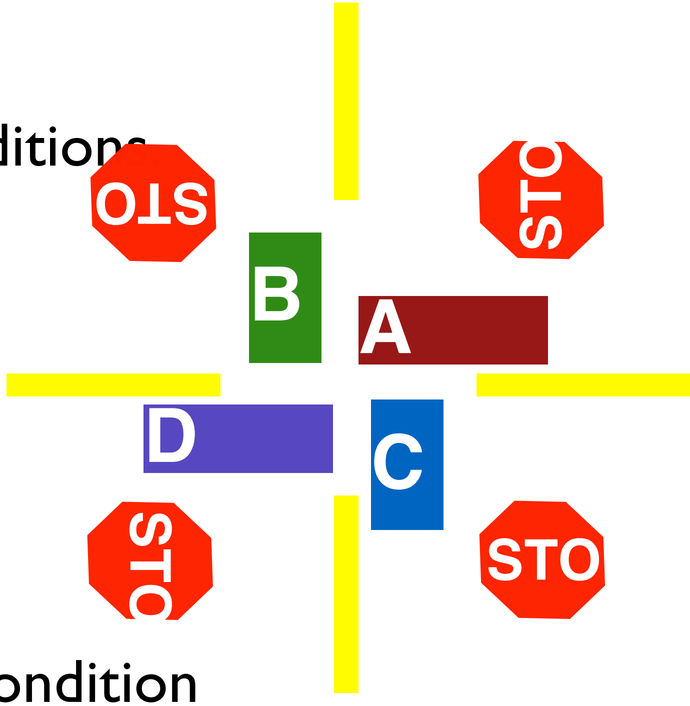
```
if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

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
```

WAIT-FREE ALGORITHMS

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

```
void add (int *val, int amt) {
    do {
        int old = *val;
    } 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 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? Use a meta lock:

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

2. HOLD-AND-WAIT

Problem: Threads hold resources 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?

Must know ahead of time which locks will be needed
Must be conservative (acquire any lock possibly needed)
Degenerates to just having one big lock

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?

Livelock:
no processes make progress, but the state of involved processes constantly changes
Classic solution: Exponential back-off

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

LOCK ORDERING IN XV6

Creating a file requires simultaneously holding:

- a lock on the directory,
- a lock on the new file's inode,
- a lock on a disk block buffer,
- idelock,
- ptable.lock

Always acquires locks in order listed

SUMMARY

When in doubt about **correctness**, better to limit concurrency
(i.e., add unnecessary locks, one big lock)

Concurrency is hard, encapsulation makes it harder!

Have a strategy to avoid deadlock and stick to it

Choosing a lock order is probably most practical for reasonable performance