CONCURRENCY BUGS

Kai Mast
CS 537
Fall 2022

(Book Chapter 32)

Recap: Semaphores

Semaphores are equivalent to locks + condition variables
- Can be used for both mutual exclusion and ordering

Semaphores contain **state**
- Initialization depends on how they will be used

`sem_wait()`:
- Waits until value > 0, then decrement (atomically)
- also called acquire or test, sometimes "P"

`sem_post()`:
- Increment value, then wake a single waiter (atomically)
- Also called release or notify, sometimes "V"

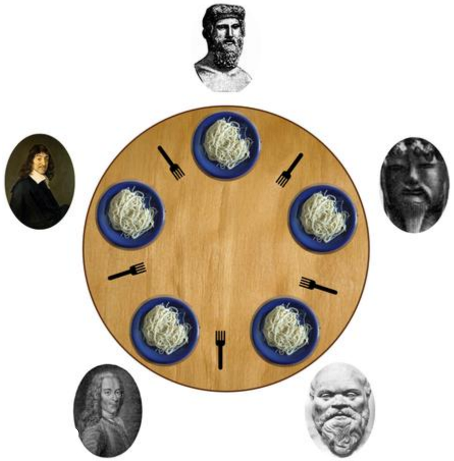
DINING PHILOSOPHERS

Problem Statement

- **N** philosophers are sitting at a round table
- They eat a strange pasta dish that requires two forks to eat
- Each philosopher shares a fork (or chopstick) with their neighbor
- Each philosopher must have both forks (or chopsticks) to eat
- Neighbors cannot eat simultaneously
- Philosophers alternate between thinking and eating

Each philosopher/thread **i** runs :

```
while (1) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
}
```



DINING PHILOSOPHERS: ATTEMPT #1

Two neighbors can't use fork at same time

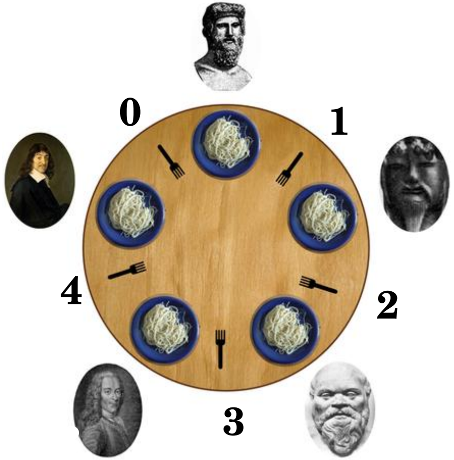
Must test if fork is there and grab it atomically

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

```
while (1) {
  think();
  take_forks(i);
  eat();
  put_forks(i);
}
```

Code for 5 philosophers:

```
sem_t forks[5]; // Initialize each to 1
void take_forks(int i) {
   sem_wait(&fork[i]);
   sem_wait(&fork[(i+1)%5]);
}
void put_forks(int i) {
   sem_post(&fork[i]);
   sem_post(&fork[(i+1)%5]);
}
```



Some orderings seem to work…

DINING PHILOSOPHERS: ATTEMPT #1

Grab right fork (from the philosophers point of view), then left fork

Code for 5 philosophers:

```
sem_t fork[5]; // Initialize each to 1
void take_forks(int i) {
   sem_wait(&fork[i]);
   sem_wait(&fork[(i+1)%5]);
}
void put_forks(int i) {
   sem_post(&fork[i]);
   sem_post(&fork[(i+1)%5]);
}
```

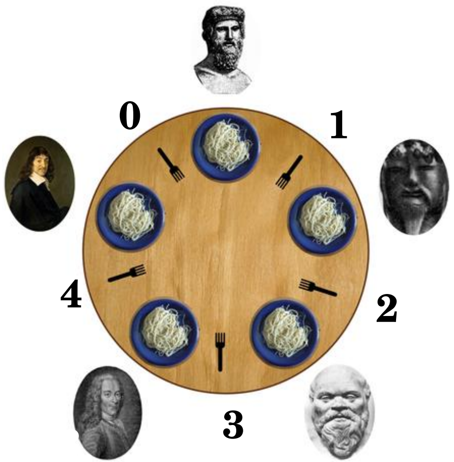
Which orderings deadlock?

All threads acquire the first lock and then wait (and get stuck) on the second

DINING PHILOSOPHERS: ATTEMPT #2

Grab lower-numbered fork first, then higher-numbered

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



No deadlock: Philosopher 3 finishes `take_forks()` eventually calls `put_forks()`

Who can run then? What is wrong with this solution?

DINING PHILOSOPHERS: HOW TO APPROACH

Introduce state variable for each philosopher i
`state[i] =` THINKING, HUNGRY, or EATING

Assume N=5 for the following

Safety:
No two adjacent philosophers eat simultaneously

```
for_all i: state[i] != EATING || state[i+1%5] != EATING
```

Liveness:
Not the case that a philosopher is hungry and both their neighbors are not eating

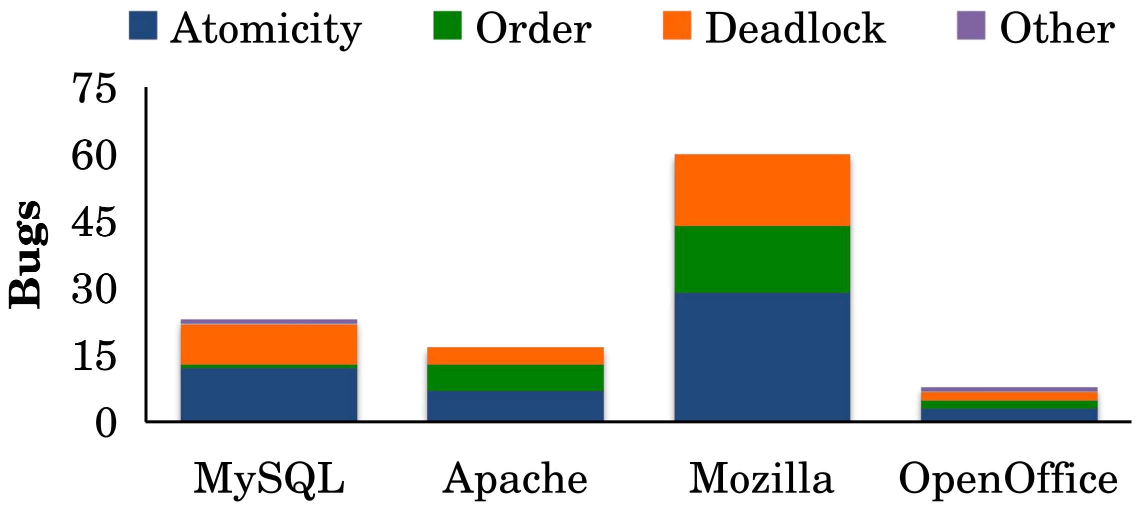
```
for_all i: state[i] != HUNGRY ||
    state[i+4%5] == EATING || state[i+1%5] == EATING
```

```c
while (1) {
  think();
  take_forks(i);
  eat();
  put_forks(i);
}
```

```c
sem_t may_eat[5]; // how to initialize?
sem_t mutex;      // how to init?
int state[5] = {THINKING};

void take_forks(int i) {
    sem_wait(&mutex); // enter critical section
    state[i] = HUNGRY;
    test_safety_and_liveness(i); // check if I can run
    sem_post(&mutex); // exit critical section
    sem_wait(&may_eat[i]);
}

void put_forks(int i) {
    sem_wait(&mutex); // enter critical section
    state[i] = THINKING;
    // check if neighbor can run now
    test_safety_and_liveness(i+1%5);
    test_safety_and_liveness(i+4%5);
    sem_post(&mutex); // exit critical section
}
```

```c
void test_safety_and_liveness(int i) {
    if (state[i] == HUNGRY
      && state[i+4%5] != EATING
      && state[i+1%5] != EATING) {
        state[i] = EATING;
        sem_post(&may_eat[i]);
    }
}
```

CONCURRENCY STUDY



Lu *et al.* [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?

Test (`thd->proc_info != NULL`) and write (`fputs(thd->proc_info)`) should be atomic

SOLUTION: FIX ATOMICITY BUGS WITH LOCKS

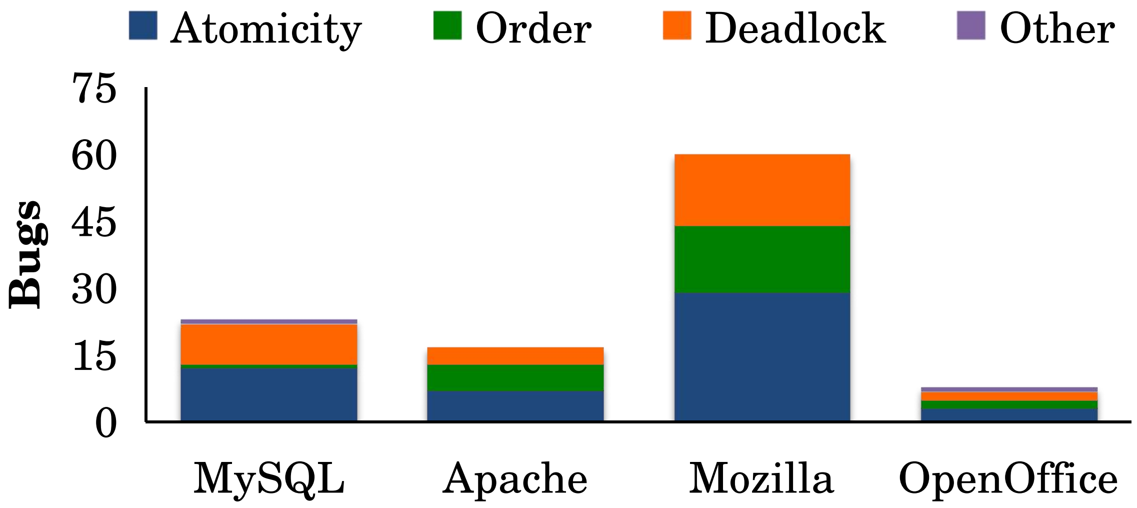
Thread 1:

```
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 *et al.* [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?

Thread 1 sets value of mThread needed by Thread 2
How to ensure reading mThread happens **after** mThread initialization?

SOLUTION: FIX ORDERING BUGS WITH C.V.S

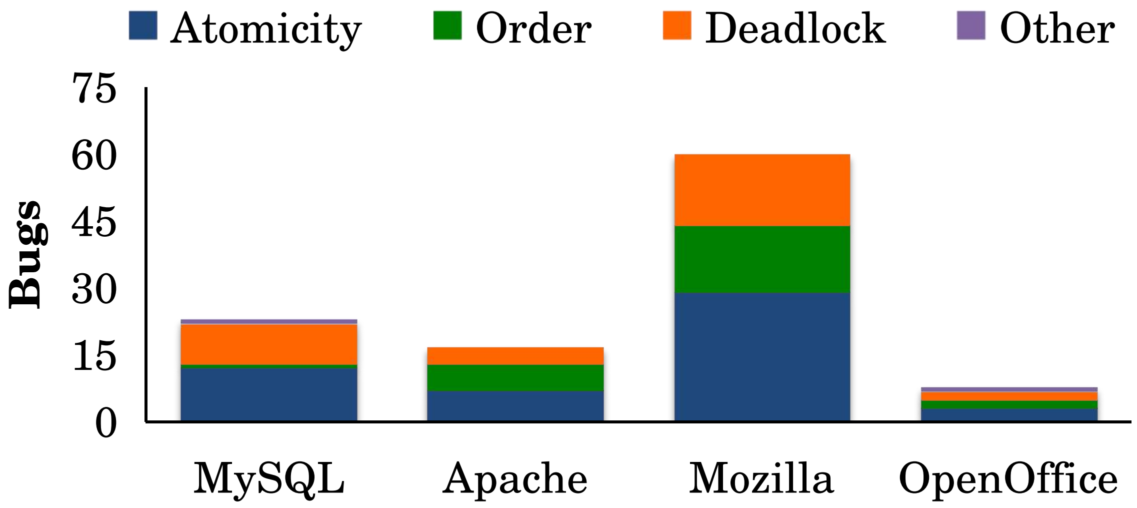
Thread 1:

```
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 *et al.* [ASPLOS 2008]:
For four major projects, search for concurrency bugs among >500K bug reports. Analyze small sample to identify common types of concurrency bugs.

DEADLOCKS

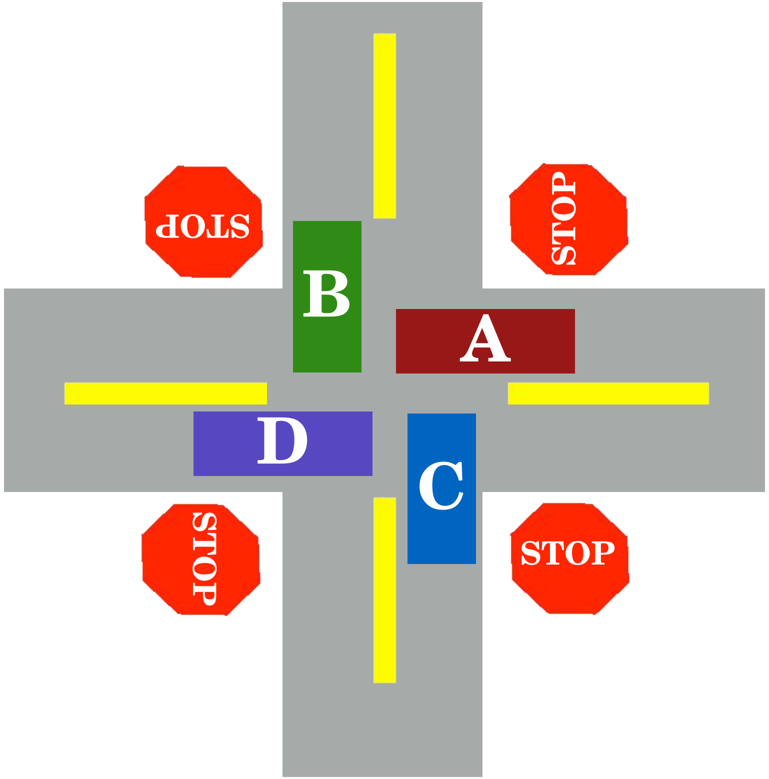
No progress can be made because **two or more** threads are each waiting for another to take some action and thus none ever does

DEADLOCK THEORY

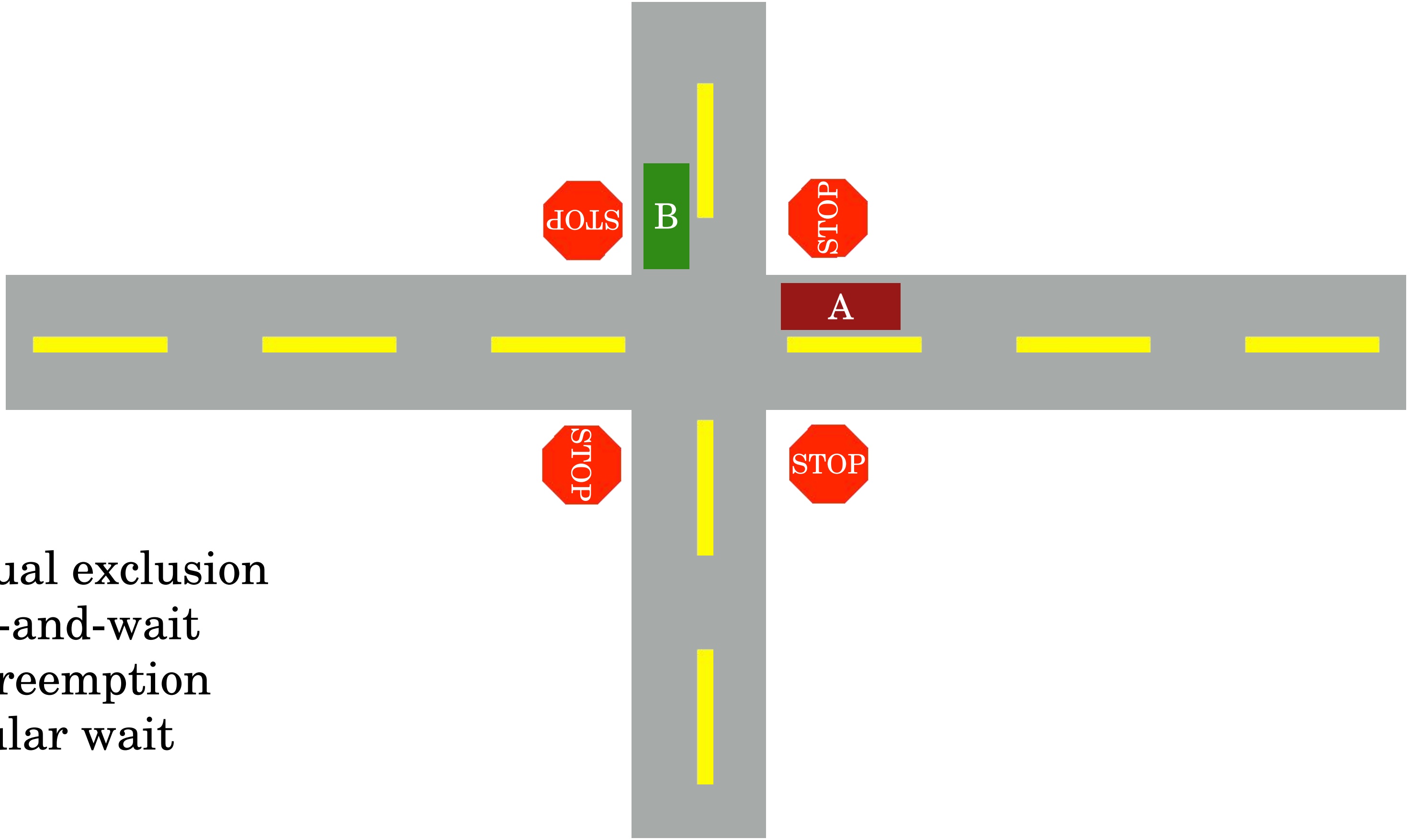
Deadlocks can only happen when these four conditions hold:

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

Can eliminate deadlock by eliminating any one condition

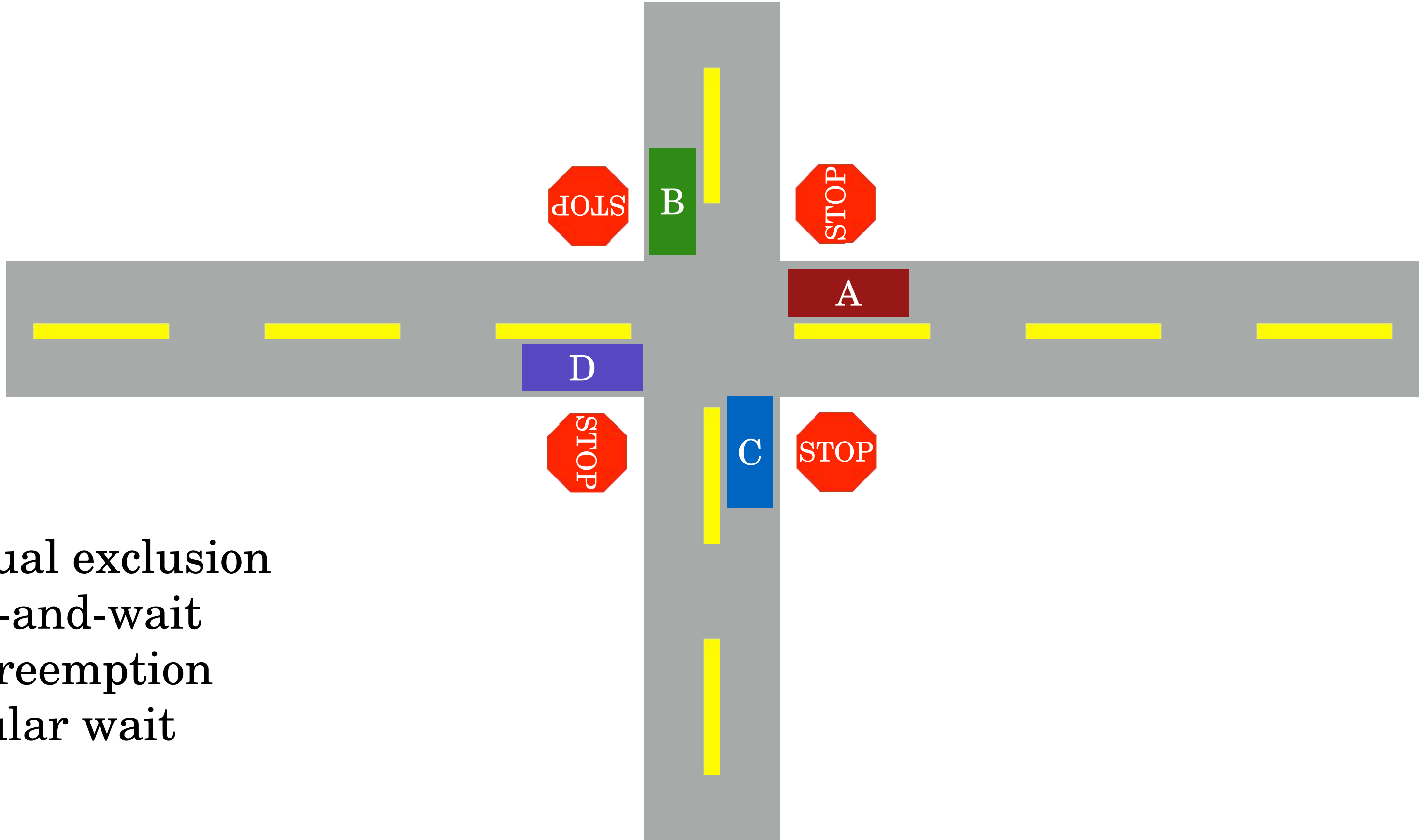


Both cars arrive at same time
Is this deadlocked?



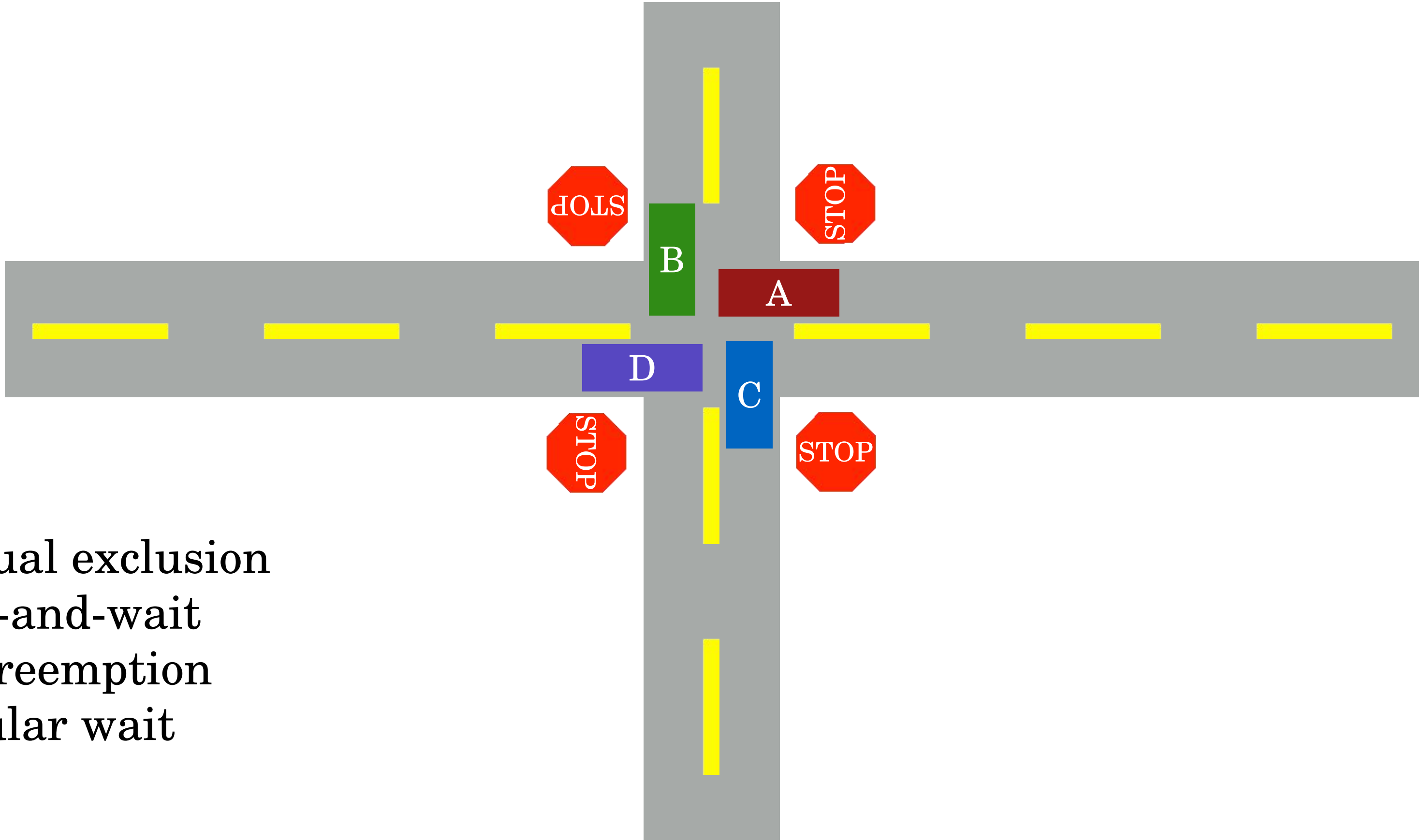
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

4 cars arrive at same time
Is this deadlocked?



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

4 cars move forward same time
Is this deadlocked?



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

CODE EXAMPLE

Can deadlock happen with these two threads?

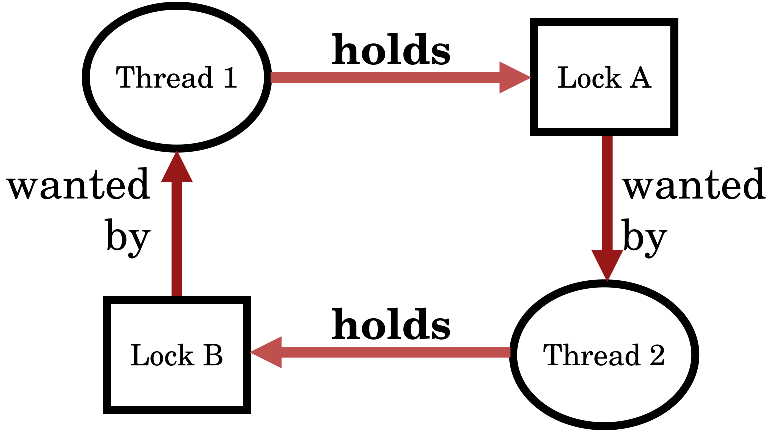
Thread 1

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

Thread 2

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

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



FIX DEADLOCKED CODE

Thread 1

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

Thread 2

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

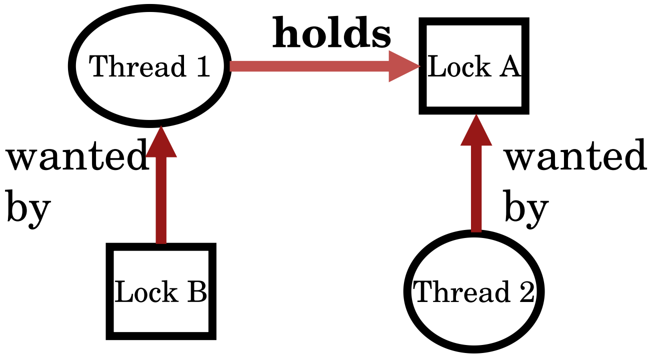
How would you fix this code?

Thread 1

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

Thread 2

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



```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);
    return rv;
}
```

How could deadlock occur?

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

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

ENCAPSULATION

Modularity can make it harder to see deadlocks

Solution?

```
void lock_both(mutex_t *m1, mutex_t *m2) {
    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?

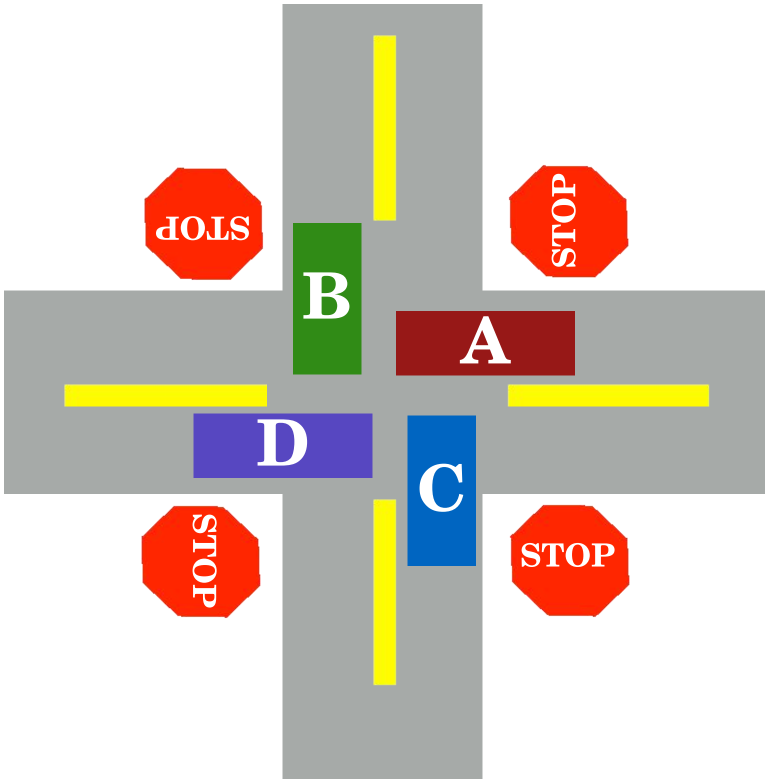
Code assumes m1 != m2 (not same lock)

DEADLOCK THEORY

Deadlocks can only happen when these four conditions hold:

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:

```
// returns 0 on failure, 1 on success
int cmp_and_swap(int *addr, int expected, int new) {
      // Does this atomically using hardware (for example, cmpxchg on x86)
      if *addr == expected {
           *addr = new;
           return 1;
       } else {
           return 0;
       }
}
```

LOCK-FREE ALGORITHMS

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

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

T1: add(&val, 2);

val = 10;

T2: add(&val, 3);

```
T1:
old = 10;

T2:
old = 10;
*val == 10 => success; *val = 13;
return true;

T1:
*val != 10 => fail
old = 13;
*val == 13 => success; *val=15;
```

LOCK-FREE ALGORITHM: LINKED LIST INSERT

```
void insert(list_t *l, int val) {
  node_t *n = malloc(sizeof(*n));
  n->val = val;
  lock(&l->mutex);
  n->next = l->head;
  l->head = n;
  unlock(&l-<m);
}
```

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

LOCK-FREE ALGORITHM: LINKED LIST INSERT

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

Assume scheduling: T1, T2, T2, T1… (one line each)

T1: insert(2); initially head = 0x0 **T2: insert(3);**

```
node_t *n = malloc(...); // 0x100
                                          node_t *n = malloc(...); // 0x200
n->val = val; // n->val == 2
                                          n->val = val; // n->val == 3
n->next = l->head; // n->next == 0x0
                                          n->next = l->head; // n->next == 0x0
cmp_and_swap => success! (l->head == 0x100)
                                          cmp_and_swap => fail (n->next != l->head)
                                          n->next = l->head; // n->next == 0x100
                                          cmp_and_swap => success (l->head == 0x200)
```

2. HOLD-AND-WAIT

Problem: Threads hold resources while waiting for additional resources

Strategy: Acquire all locks atomically

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

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 (reduces concurrency)

```
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);
```

3. NO PREEMPTION

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

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

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

Disadvantages?

Potential Livelock

No processes make progress, but state of involved processes constantly changes

Classic solution: Exponential random back-off

4. CIRCULAR WAIT

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

Practical Solution:

- Decide which locks must be acquired before others
- If A before B, never acquire A if B is already held!
- Document 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

PRACTICE PROBLEM 1

Global variables

```
int x, y, z;
pthread_mutex a, b, c;
```

Is there a deadlock possible?

What causes the deadlock?

- Thread A: locks A, B
- Thread B: locks C
- Thread A: tries to lock C
- Thread B: try to lock A

Thread A

```
lock(a);
x = x + 1;
lock(b);
y = y + x;
lock(c);
z = z + x;
unlock(c);
unlock(b);
unlock(a);
```

Thread B

```
lock(b);
y = y - 1;
lock(c);
z = z – 1;
unlock(b);
lock(a);
x = x + z;
unlock(c);
unlock(a);
```

Practice Problem 2

```
void add_to_stack(void *data, elem_t *stack)
{
    elem_t *elem;
    elem = malloc(sizeof(elem_t));
    elem->value = data;
    elem->next = stack->head;
    stack->head = elem;
}
```

```
void* pop_from_stack(elem_t *stack)
{
    elem_t *top = stack->head;
    stack->head = top->next;
    void *data = top->value;
    free(top);
    return data;
}
```

Where can a race happen? Where should we add lock/unlock?
When reading/writing stack

PRACTICE PROBLEM 3

```
volatile int balance = 0;

void *mythread (void *arg) {
    balance = balance + 200;
    printf(“Balance is %i\n”,
        balance);
    return NULL;
}
```

```
int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL,
        mythread, “A”);
    pthread_create(&p2, NULL,
        mythread, “B”);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf(“Final Balance is %i\n”,
        balance);
}
```

How many threads are created? 2 (in addition to the main thread)

Why do we need to call `pthread_join`? Wait for balance to be updated successfully

PRACTICE PROBLEM 3

```
volatile int balance = 0;

void *mythread (void *arg) {
    balance = balance + 200;
    printf(“Balance is %i\n”,
        balance);
    return NULL;
}
```

```
int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL,
        mythread, “A”);
    pthread_create(&p2, NULL,
        mythread, “B”);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf(“Final Balance is %i\n”,
        balance);
}
```

What is the final balance printed when calling `printf(“Final balance is %i\n”, balance)`?

Practice Problem 3

```
volatile int balance = 0;

void *mythread (void *arg) {
    balance = balance + 200;
    printf("Balance is %i\n",
        balance);
    return NULL;
}
```

```
int main (int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_create(&p1, NULL,
        mythread, "A");
    pthread_create(&p2, NULL,
        mythread, "B");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final Balance is %i\n",
        balance);
}
```

How can I make this code correct?

Practice Problem 4

```
typedef struct _lock_t {
     int flag;
} lock_t

void init(lock_t *lock) {
     lock->flag = ABC;
}
```

```
void acquire(lock_t *lock) {
   while (atomic_exchange(&lock->flag, XYZ) == XYZ)
   { /* spin */ }
}

void release (lock_t *lock) {
   lock->flag = 0;
}
```

What should the value of ABC be while initialization? 0

What should the value of XYZ be? 1