Concurrency Bugs

Questions answered in this lecture:

- Why is concurrent programming difficult?
- What type 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)?

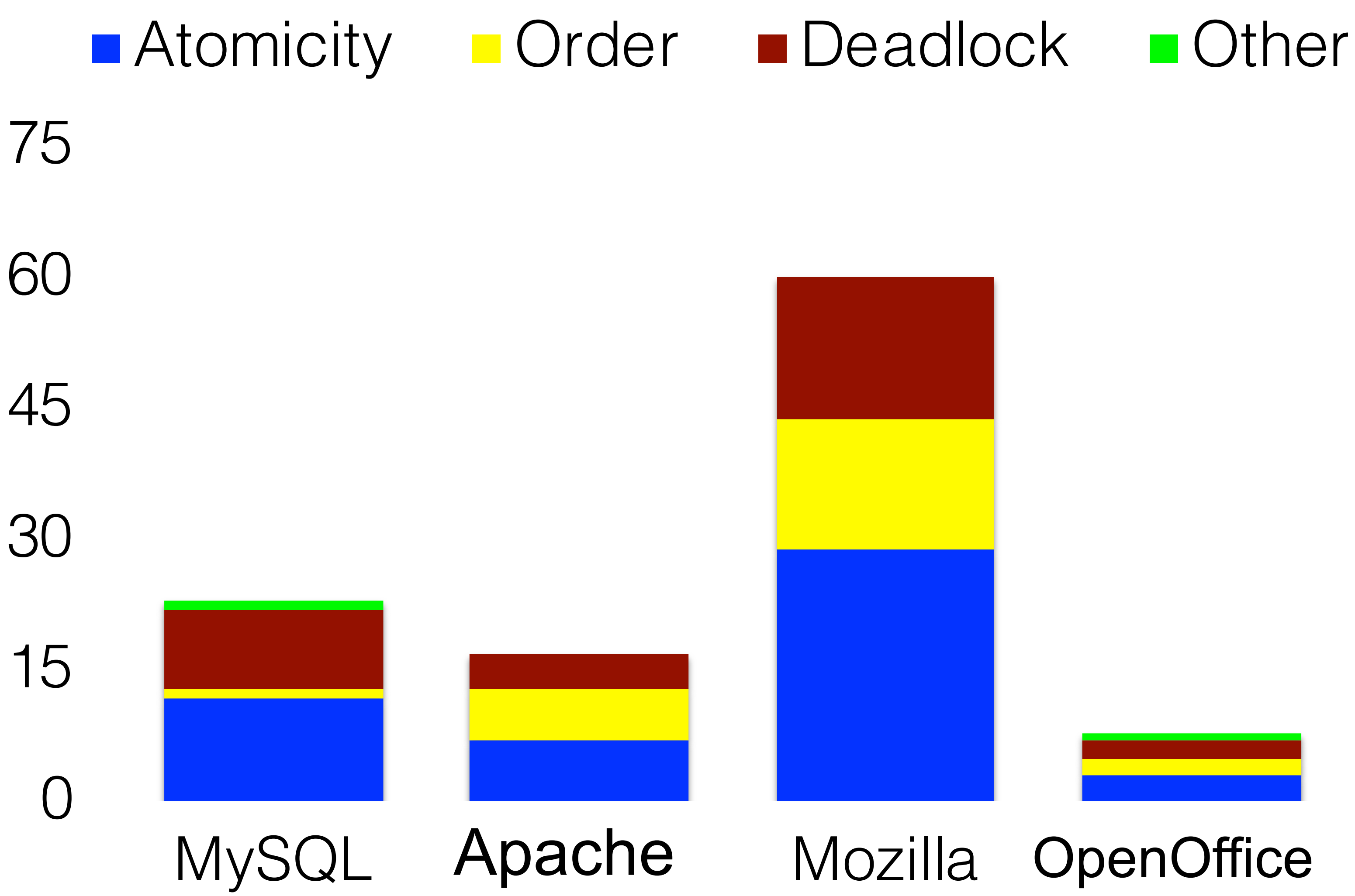
Concurrency in Medicine: Therac-25 (1980's)

"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**."

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

Concurrency Study from 2008



Lu *etal.* Study:

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

Source: http://pages.cs.wisc.edu/~shanlu/paper/asplos122-lu.pdf

Atomicity: MySQL

Thread 1:

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

Thread 2:

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

Test (thd->proc_info != NULL) and set (writing to thd->proc_info) should be atomic

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);
```

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 Thread2

How to ensure that reading MThread happens after mThread initialization?

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);

      …
}
```

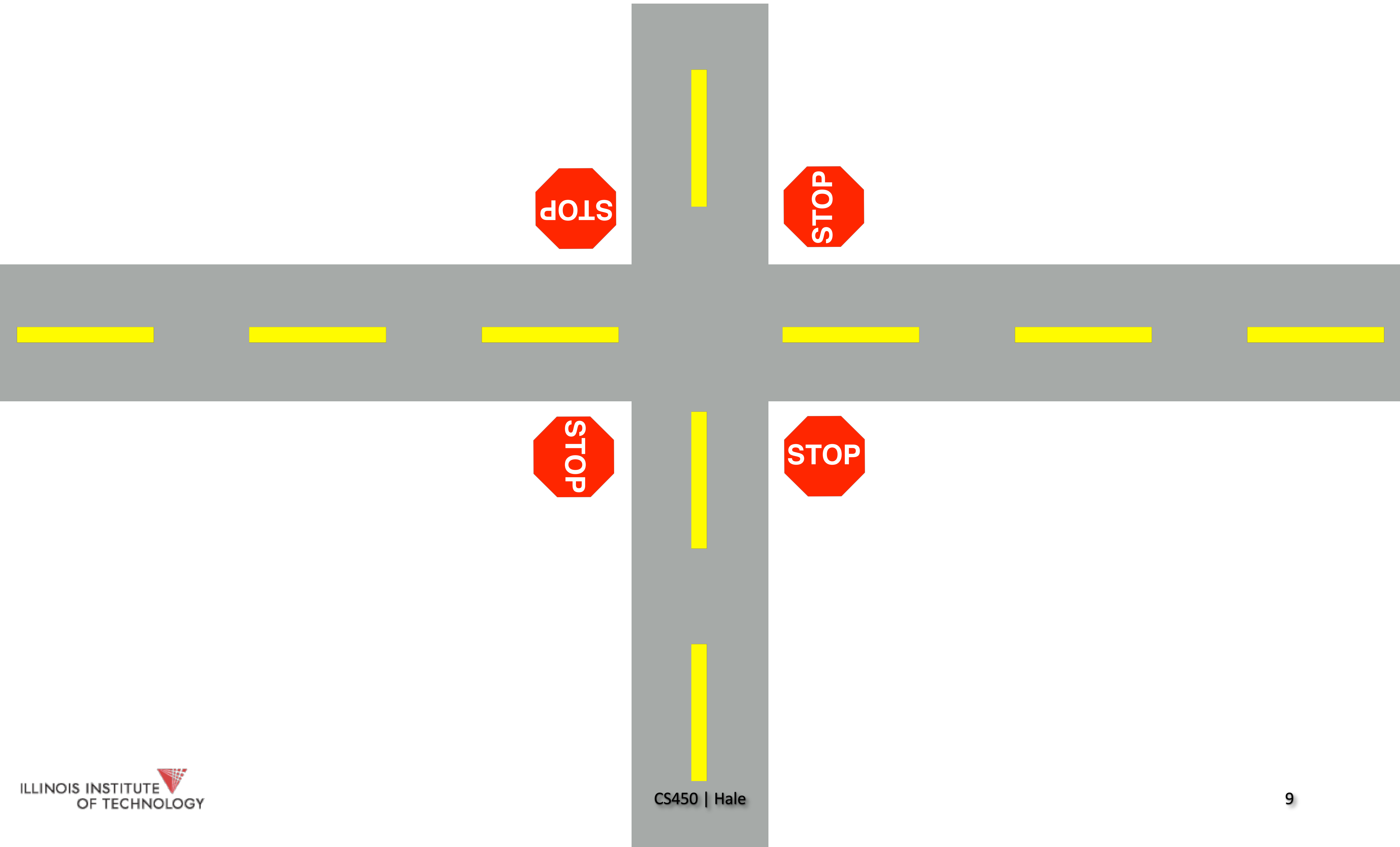
Thread 2:

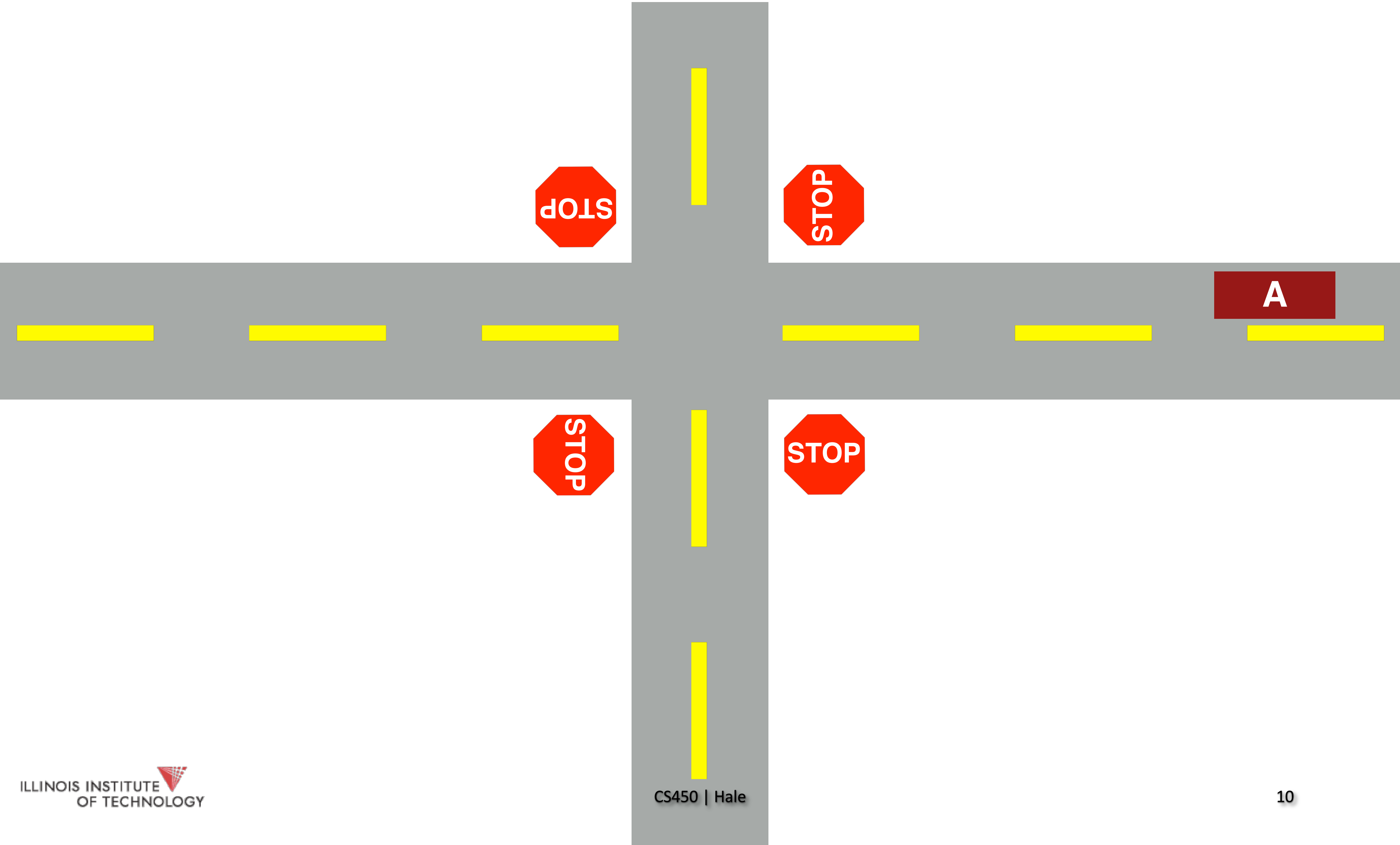
```
void mMain(…) {
  …

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

  mState = mThread->State;
  …
}
```

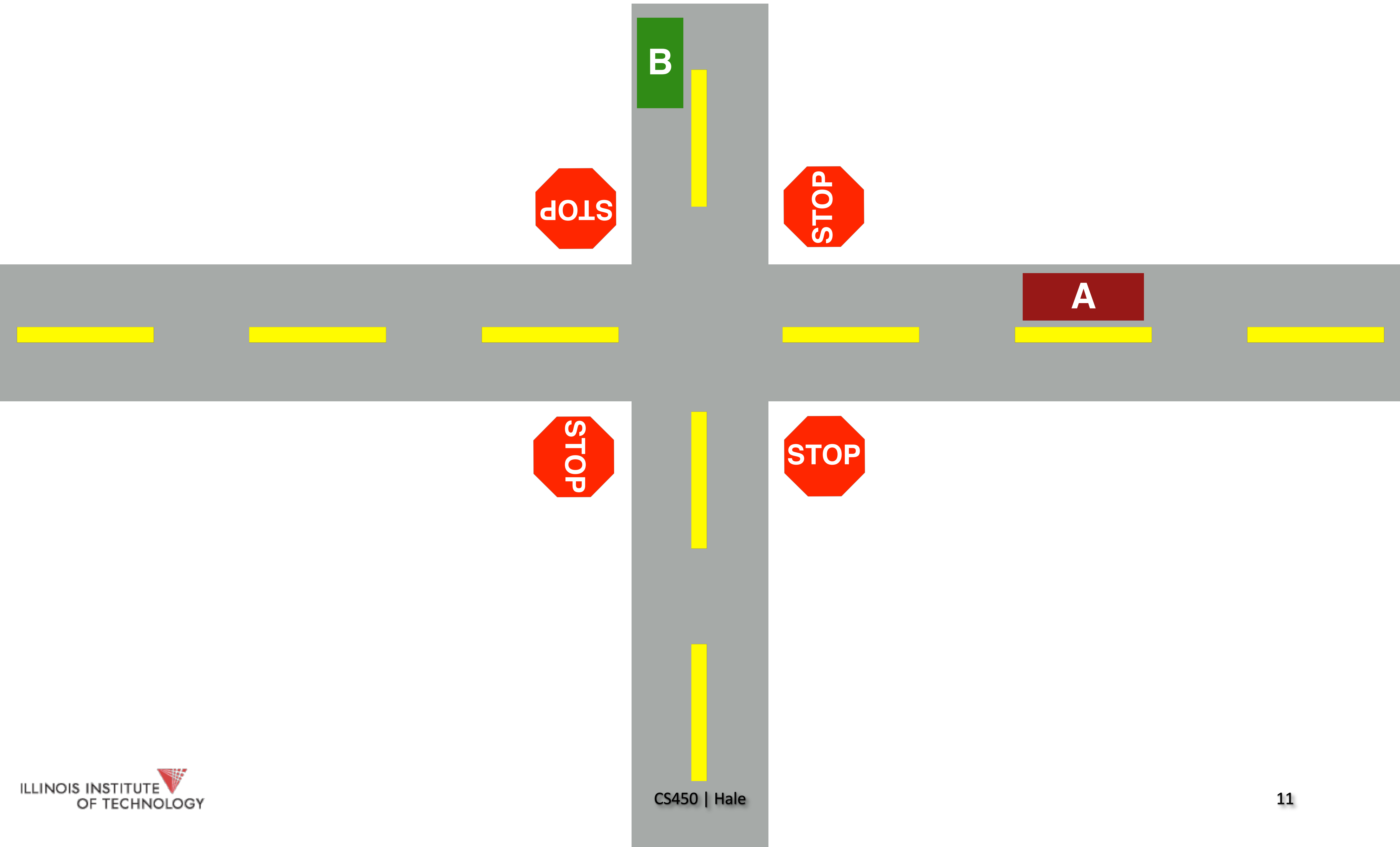
Deadlock

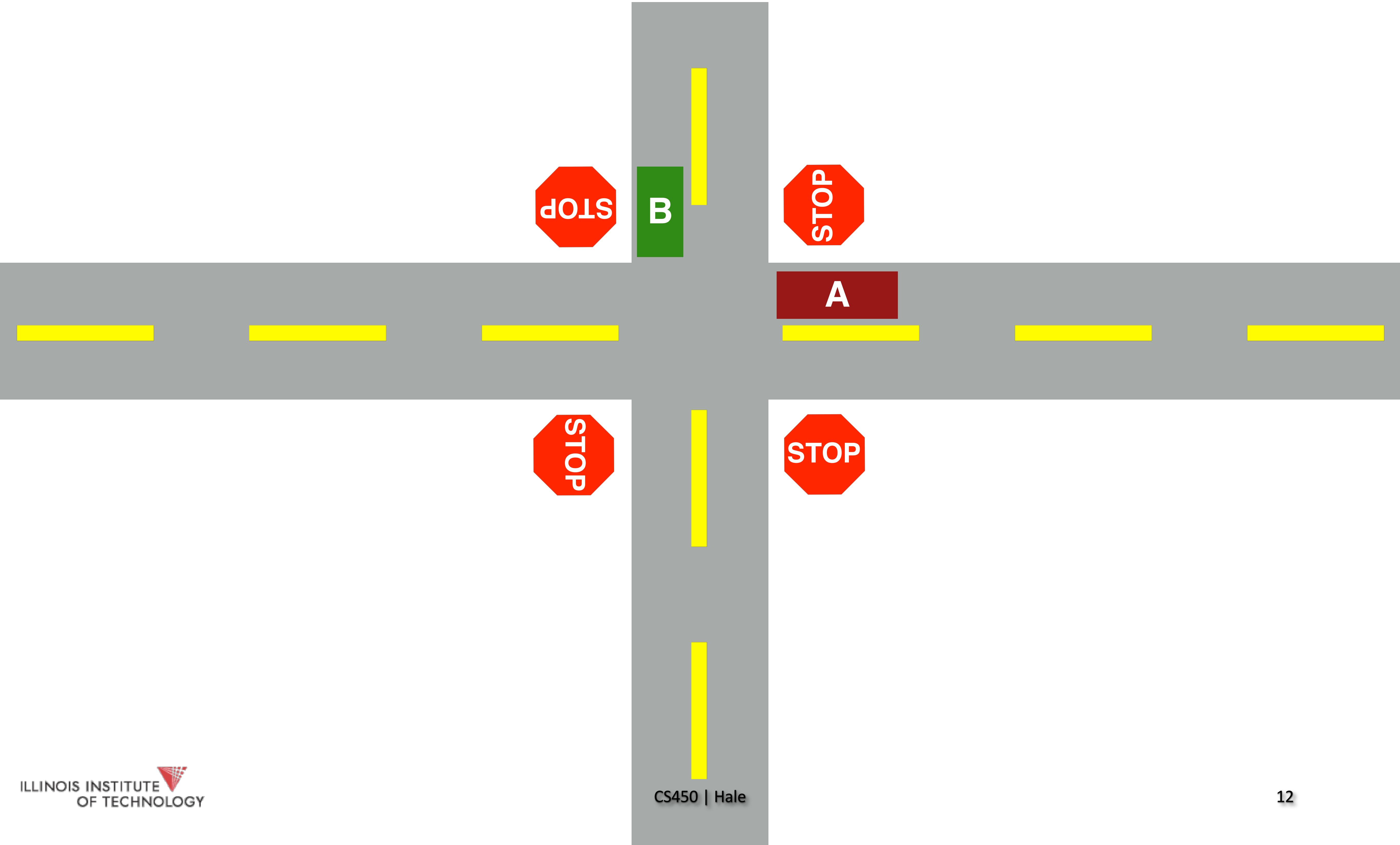
Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

“Cooler" name: the **deadly embrace** (Dijkstra)

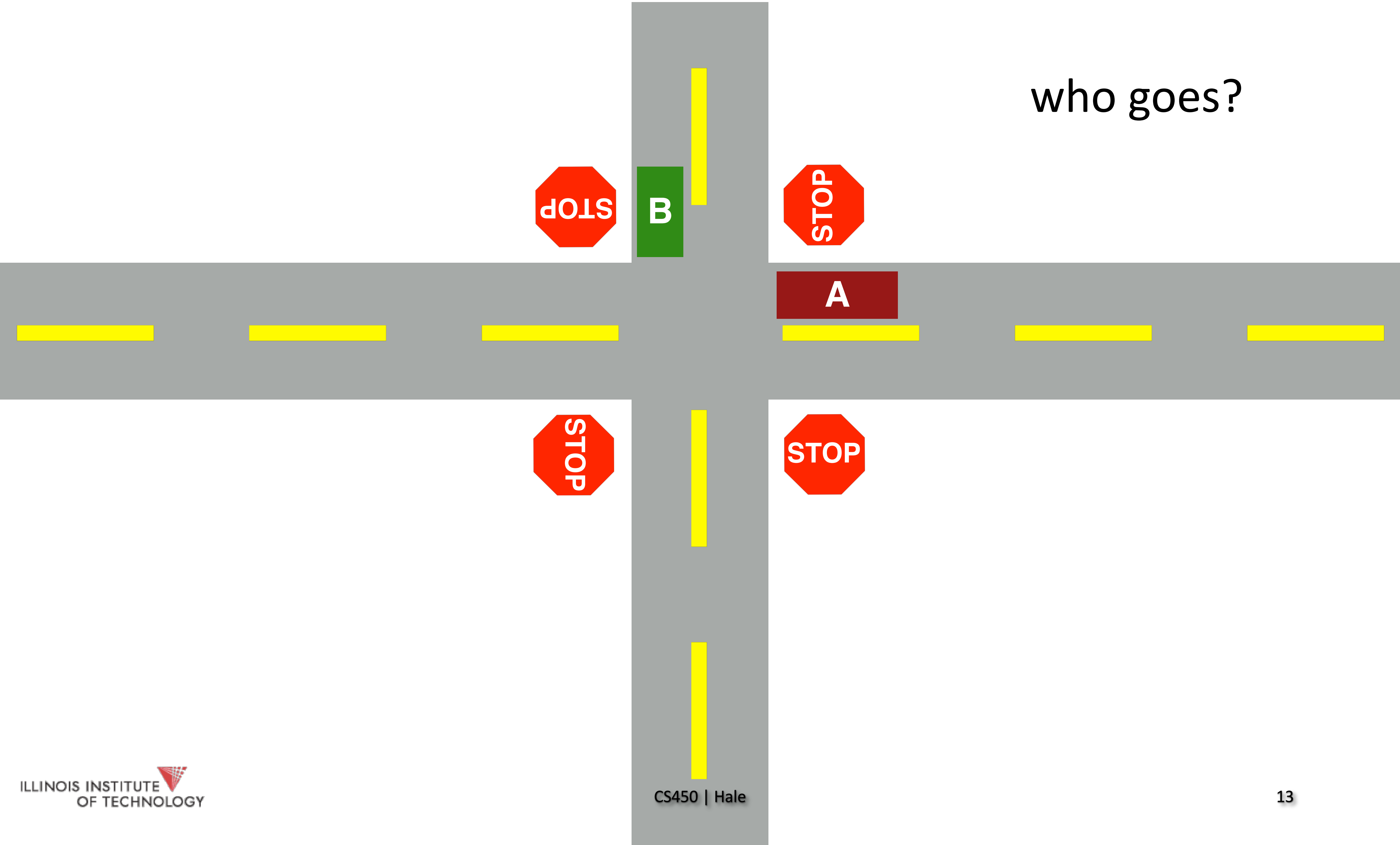

STOP
STOP
STOP
STOP

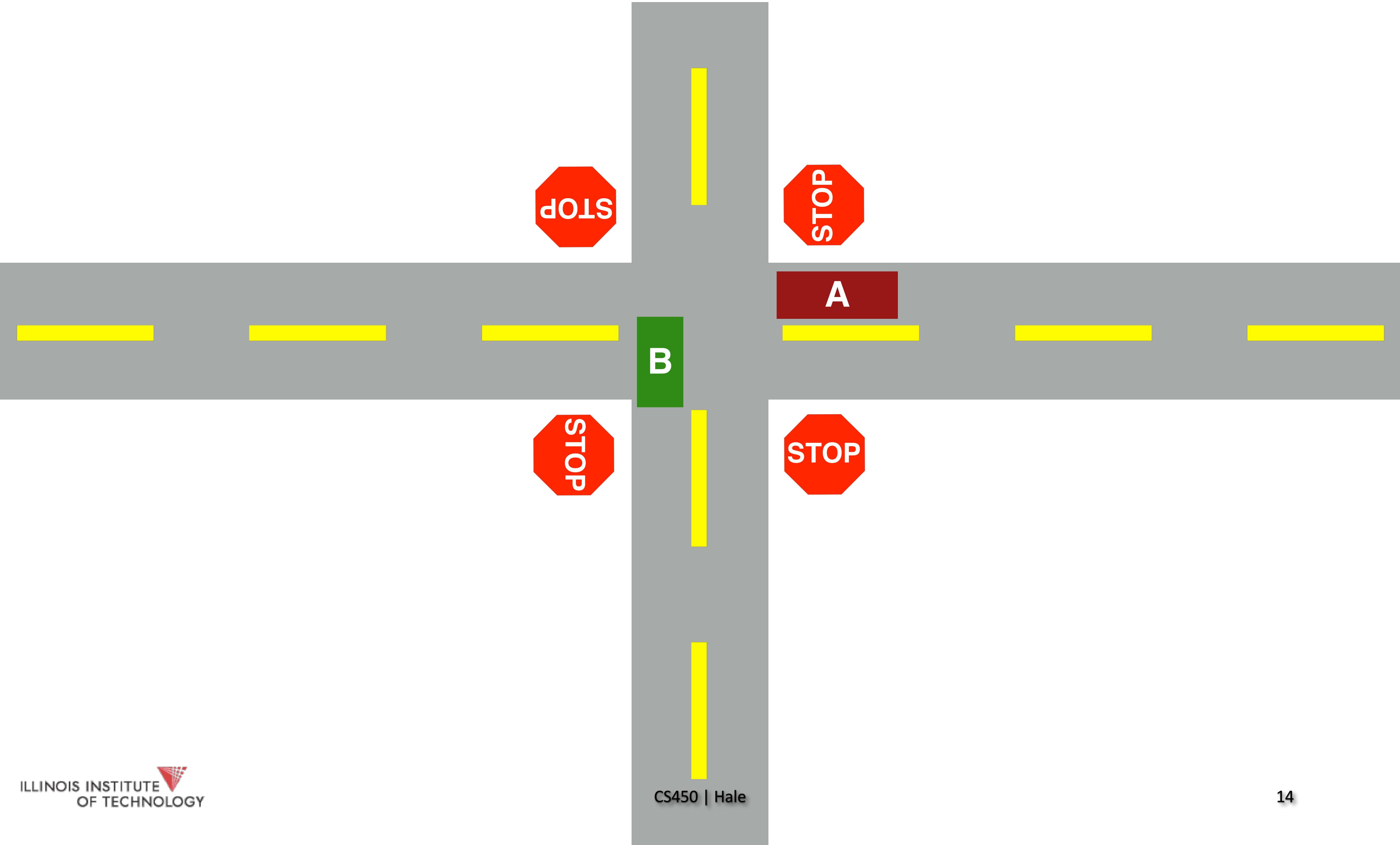

STOP
STOP
STOP
STOP
A

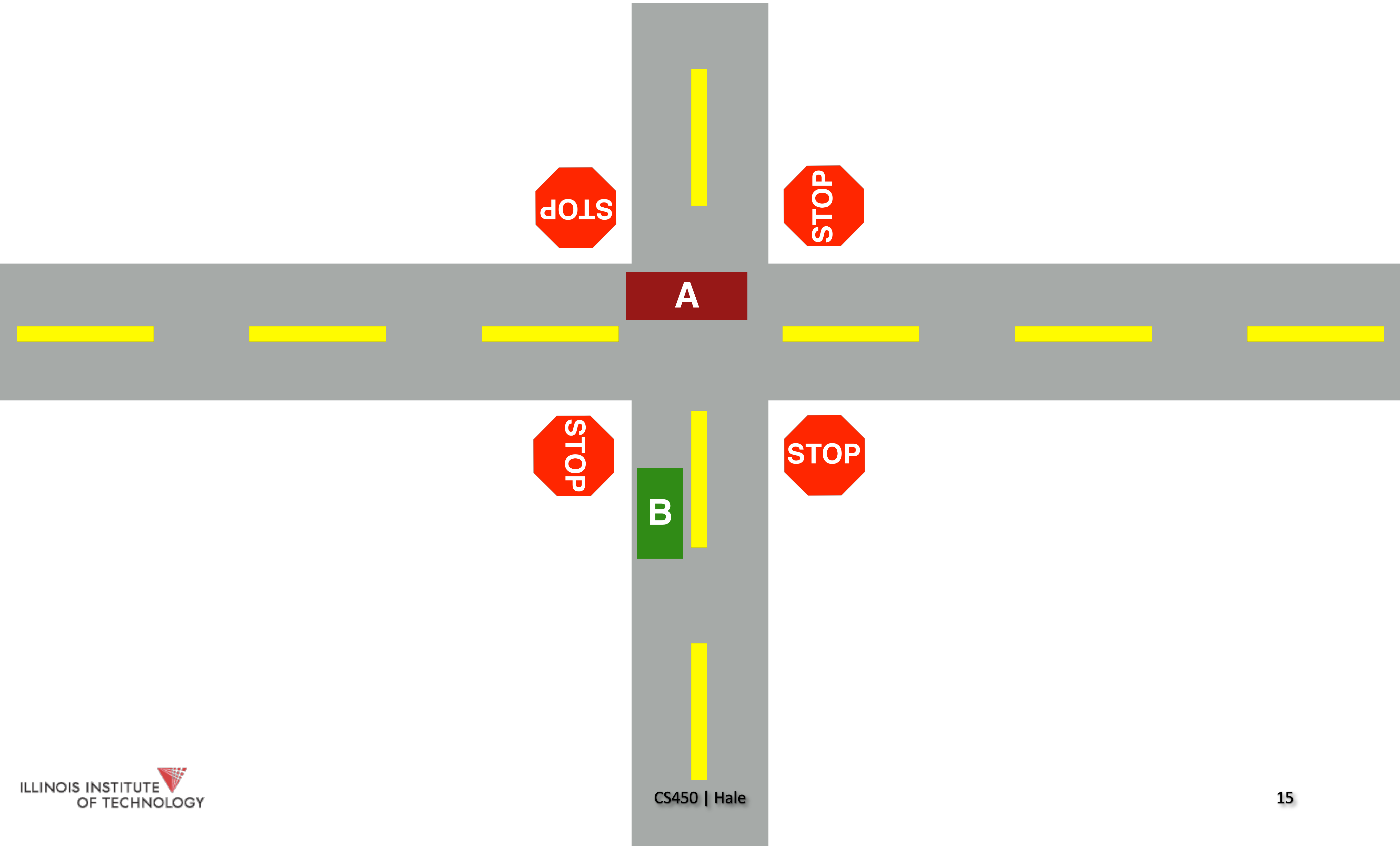

B
STOP
STOP
A
STOP
STOP

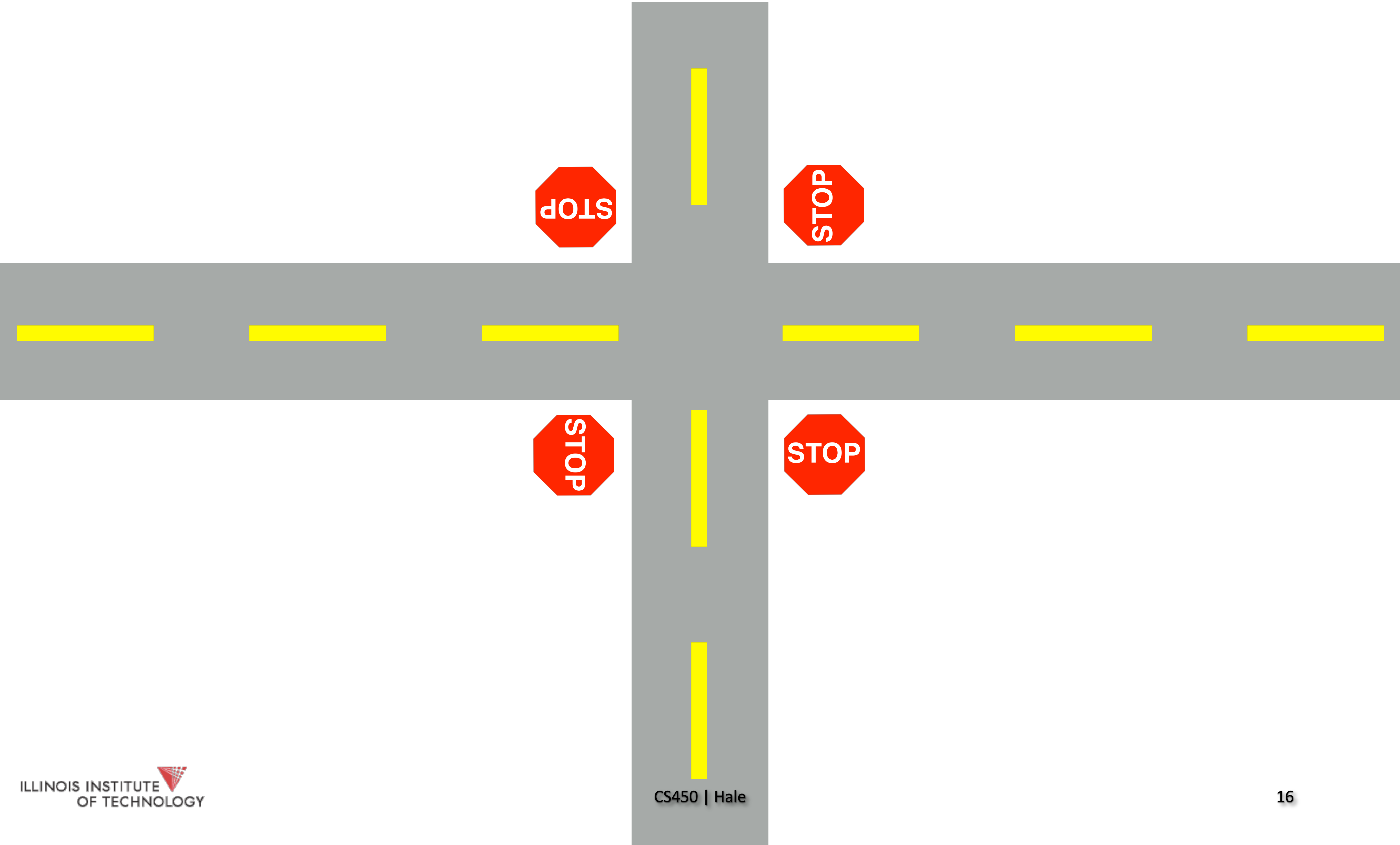

STOP
B
STOP
A
STOP
STOP

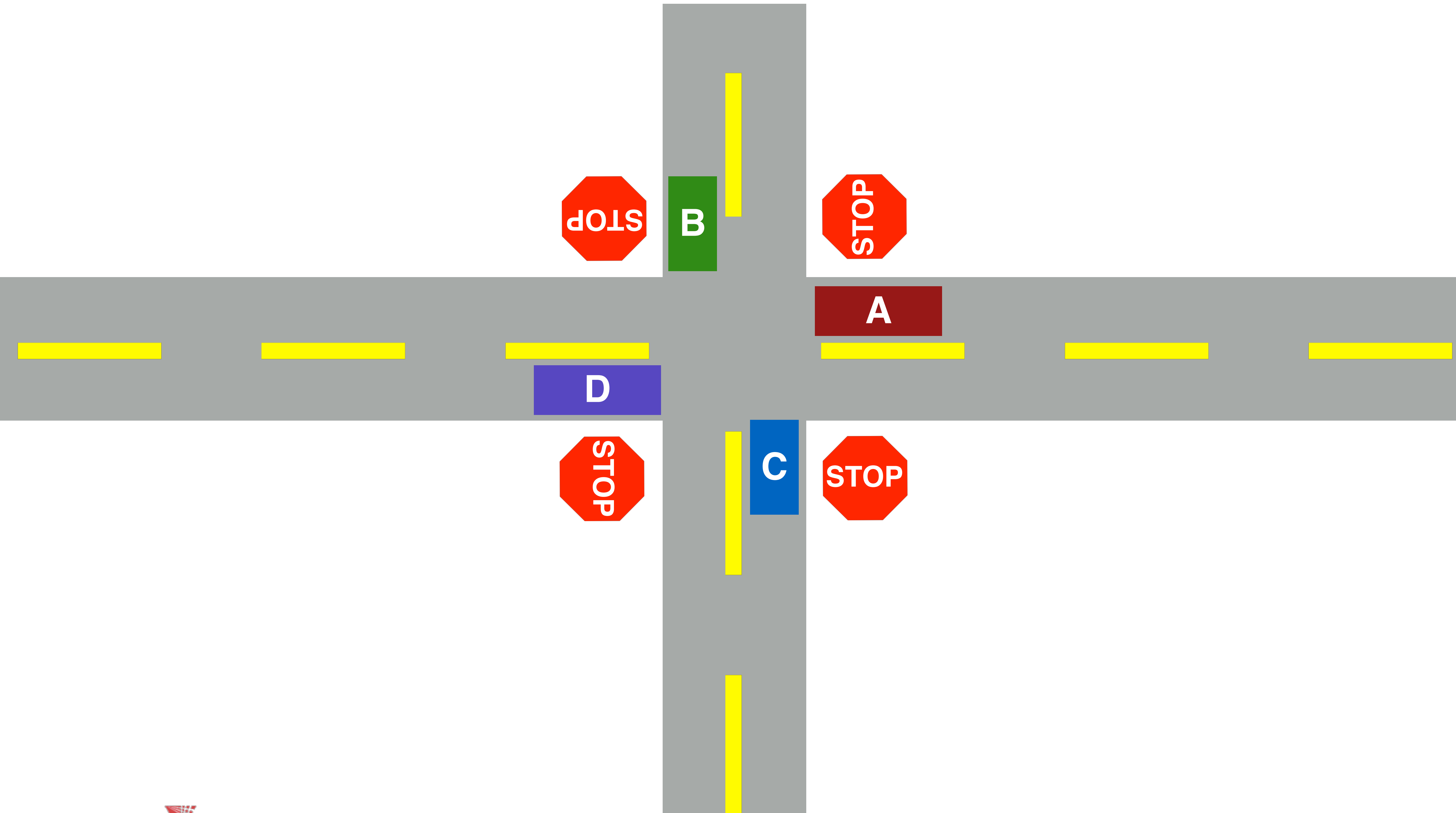
who goes?

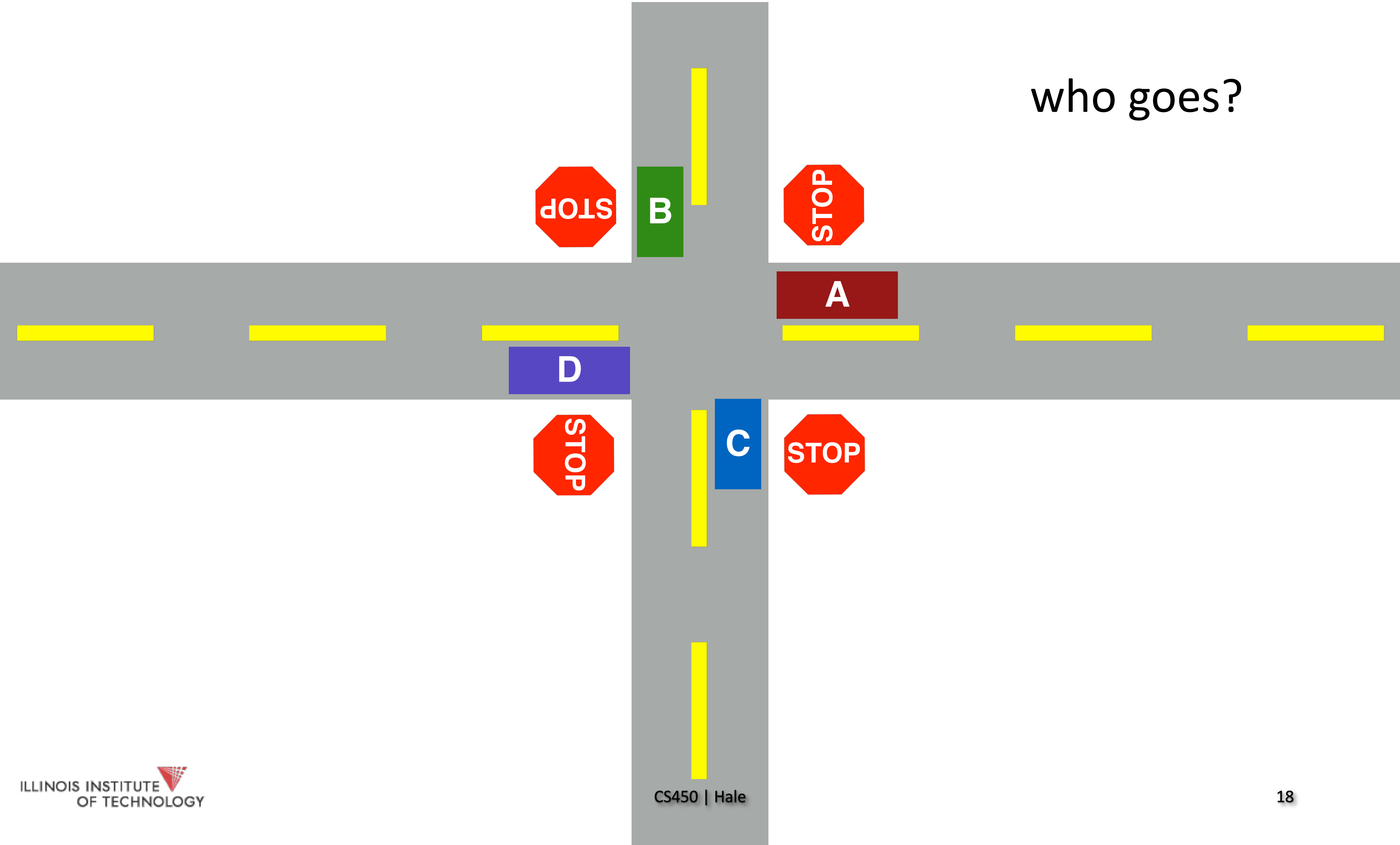



STOP
STOP
A
B
STOP
STOP

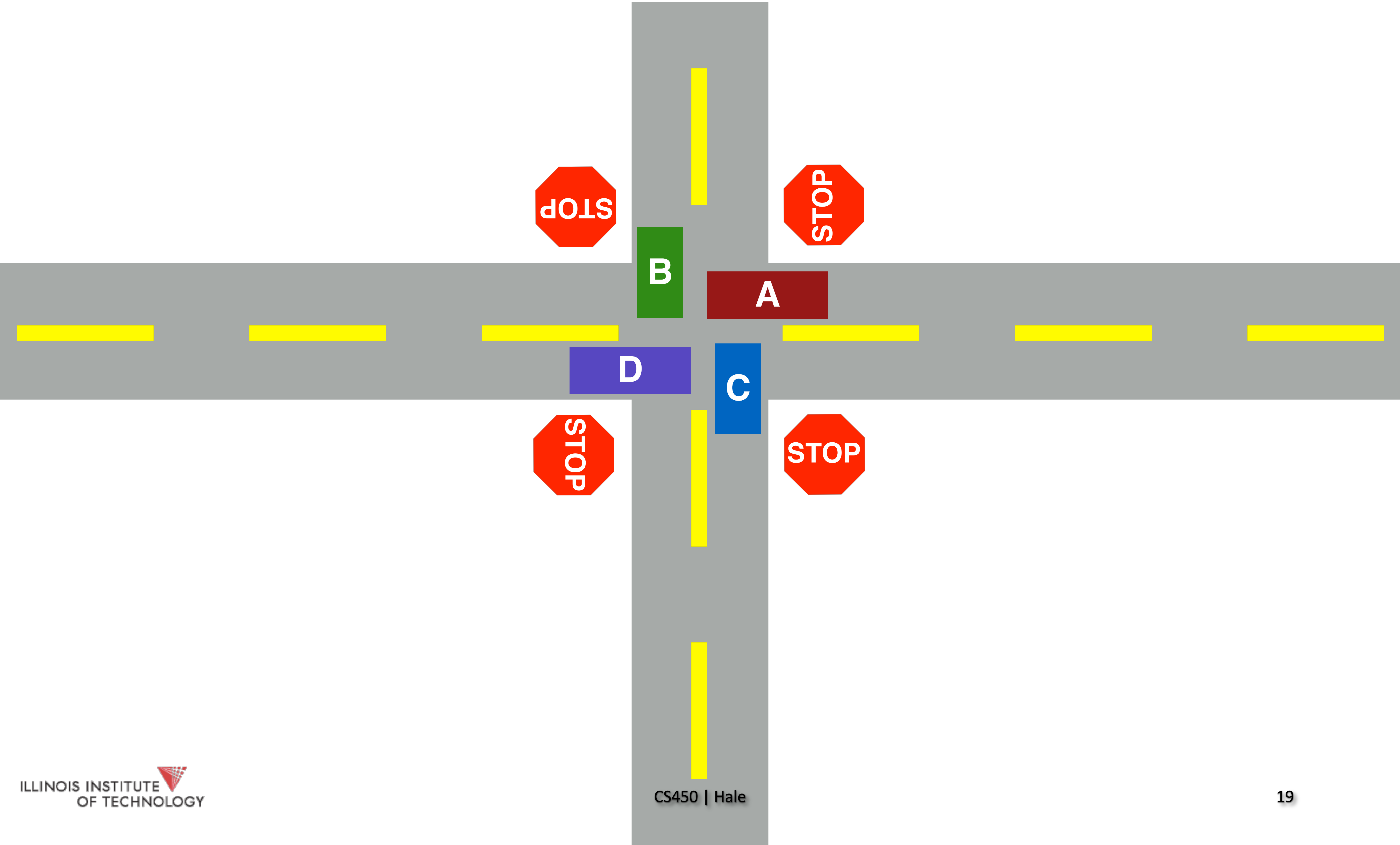

STOP
STOP
A
STOP
B
STOP

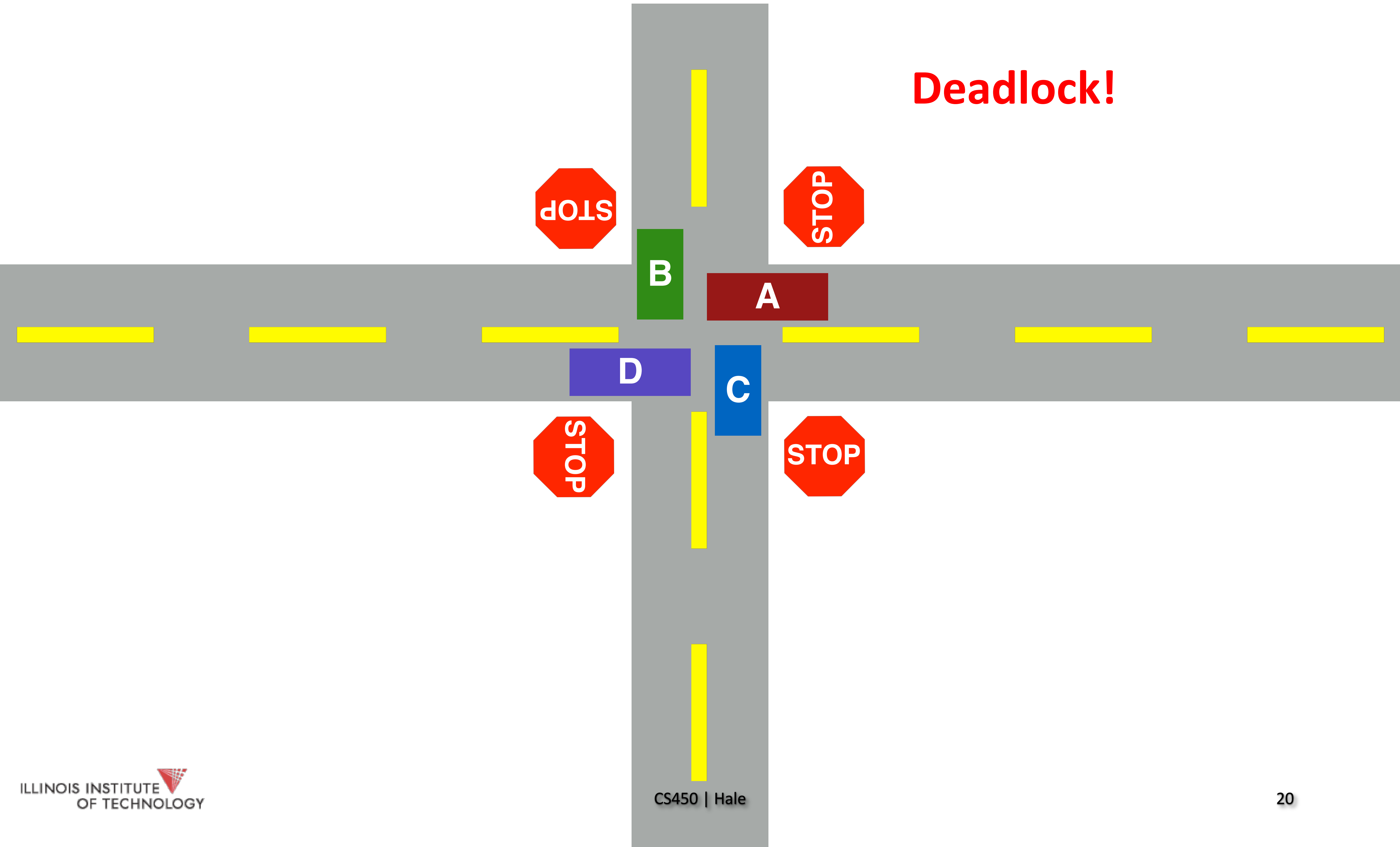

STOP
STOP
STOP
STOP


STOP
B
STOP
A
D
STOP
C
STOP



who goes?


STOP
STOP
B
A
D
C
STOP
STOP


Deadlock!
STOP
STOP
B
A
D
C
STOP
STOP

Code Example

Thread 1:

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

Thread 2:

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

Can deadlock happen with these two threads?

Code Example

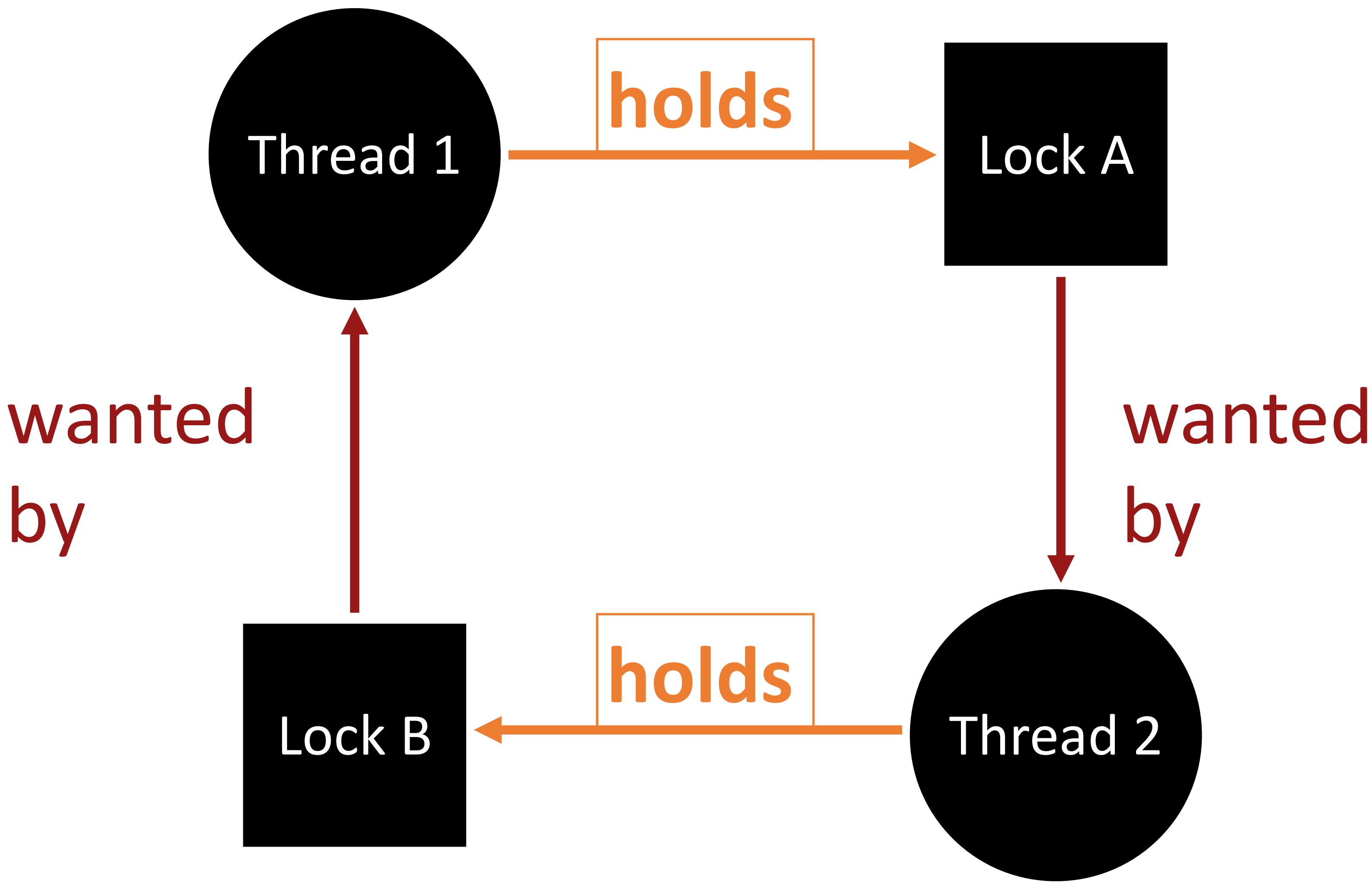
Thread 1:

```
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);
```

How would you fix this code?

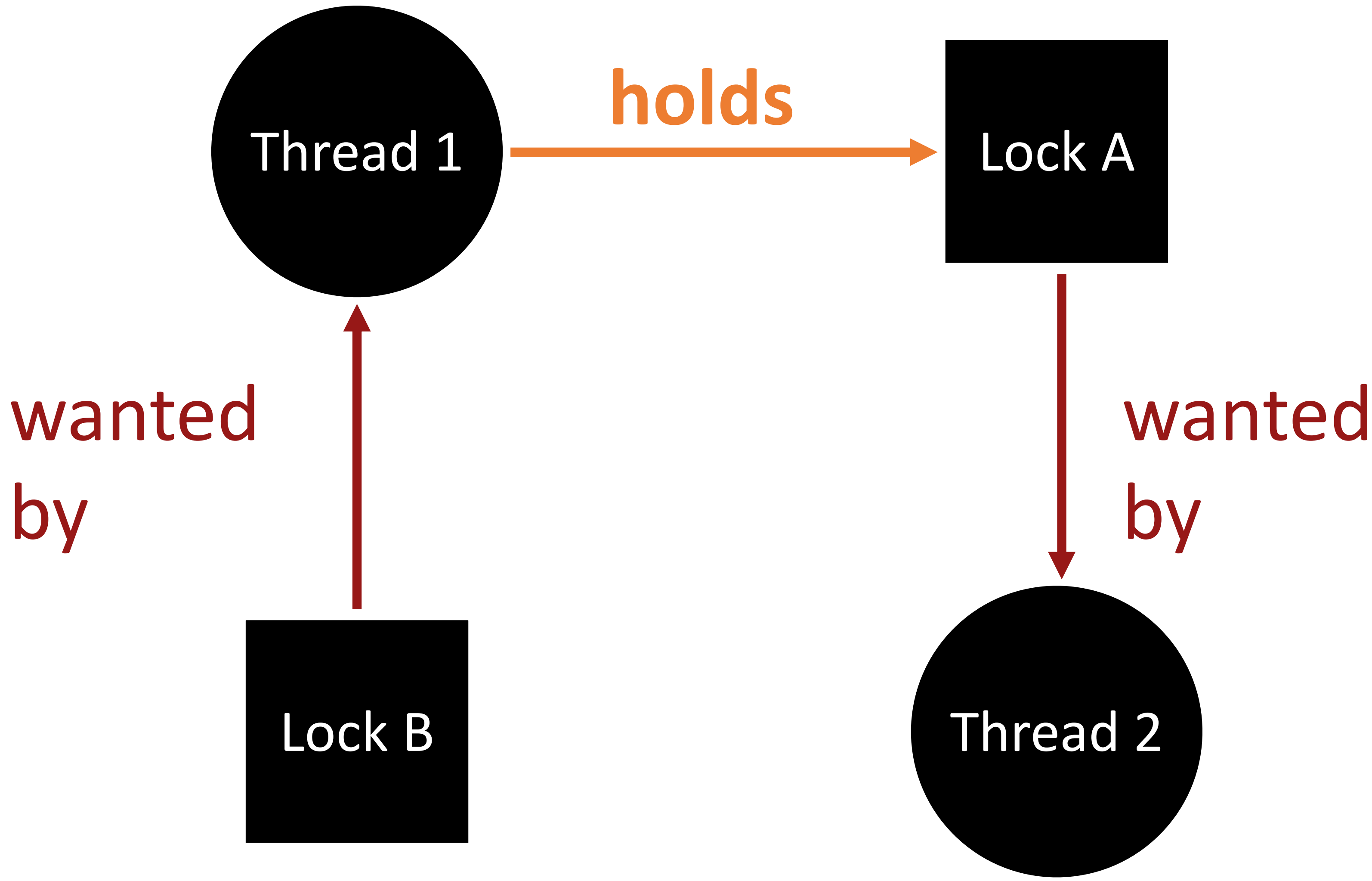
Thread 1

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

Thread 2

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

Non-circular Dependency (fine)



What’s Wrong?

```
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);
}
```

Encapsulation

Modularity can make it harder to see deadlocks

Thread 1:

```
rv = set_intersection(setA,
                      setB);
```

Thread 2:

```
rv = set_intersection(setB,
                      setA);
```

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?

Code assumes m1 != m2 (not same lock)

Deadlock Theory

Deadlocks can only happen with these four conditions:

- **mutual exclusion**
- hold-and-wait
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition

Mutual Exclusion

Definition:

Threads claim ***exclusive control of resources*** *that they require (e.g., thread grabs a lock)*

Wait-Free Algorithms

Strategy: Eliminate locks!

Try to replace locks with atomic primitive:

```
int CompAndSwap(int *addr, int expected, int new);
```

Returns 0: fail, 1: success

```
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);
}
```

?? → old

Wait-Free Algorithms: Linked List Insert

Strategy: Eliminate locks!

```
int CompAndSwap(int *addr, int expected, int new);
```

Returns 0: fail, 1: success

```
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));
}
```

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- **hold-and-wait**
- no preemption
- circular wait

Eliminate deadlock by eliminating any one condition

Hold-and-Wait

Definition:

*Threads **hold** resources allocated to them (e.g., locks they have already acquired) **while waiting for additional resources** (e.g., locks they wish to acquire).*

Eliminate Hold-and-Wait

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, like this:

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

// Critical section code

unlock(…);
```

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

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- **no preemption**
- circular wait

Eliminate deadlock by eliminating any one condition

No preemption

Definition:

Resources *(e.g., locks)* ***cannot be forcibly removed*** *from threads that are holding them.*

Support Preemption

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

Deadlock Theory

Deadlocks can only happen with these four conditions:

- mutual exclusion
- hold-and-wait
- no preemption
- **circular wait**

Eliminate deadlock by eliminating any one condition

Circular Wait

Definition:

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

Eliminating Circular Waiting

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 Linux

In linux-3.2.51/include/linux/fs.h

```
/* inode->i_mutex nesting subclasses for the lock
 * validator:
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 * The locking order between these classes is
 * parent -> child -> normal -> xattr -> quota
 */
```

Summary

- When in doubt about correctness, better to limit concurrency (i.e., add unneccessary 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