OSTEP
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, tr ylocks, and ordering across locks)?

Review Semaphores

CV's vs. Semaphores

CV rules of thumb:

- Keep state in addition to CV's
- Always do wait/signal with lock held
- Whenever you acquire a lock, recheck state

How do semaphores eliminate these needs?

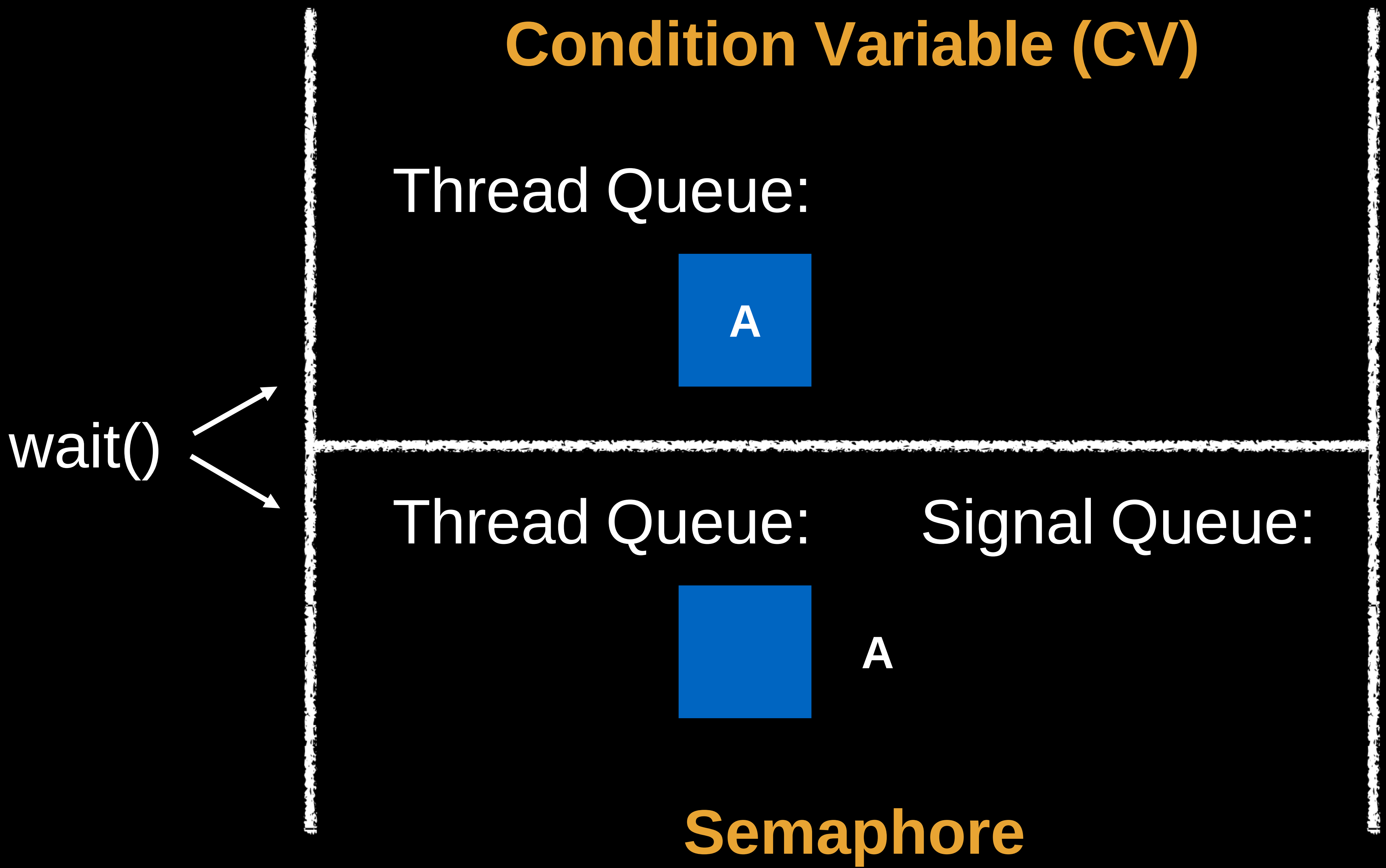
Condition Variable (CV)

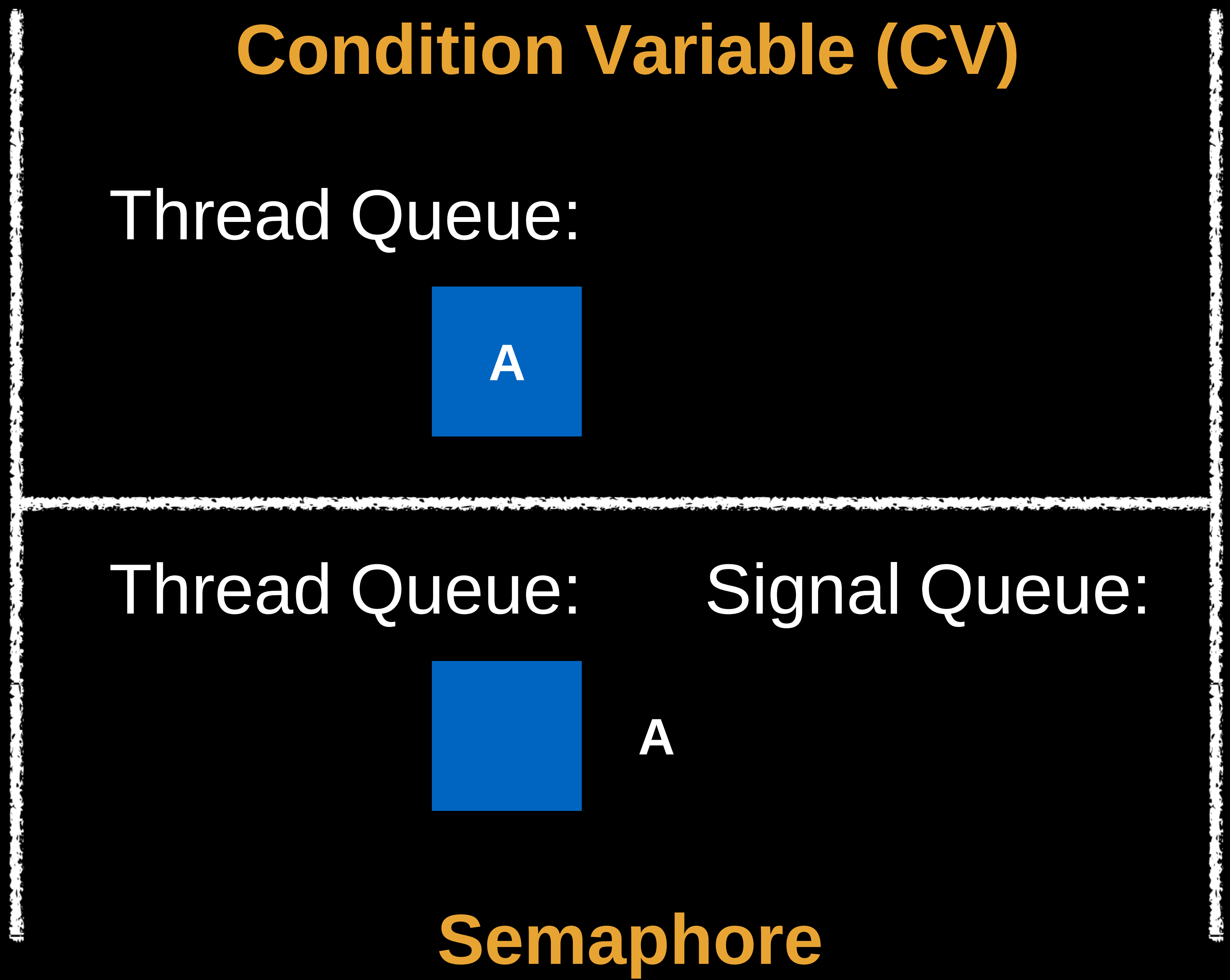
Thread Queue:

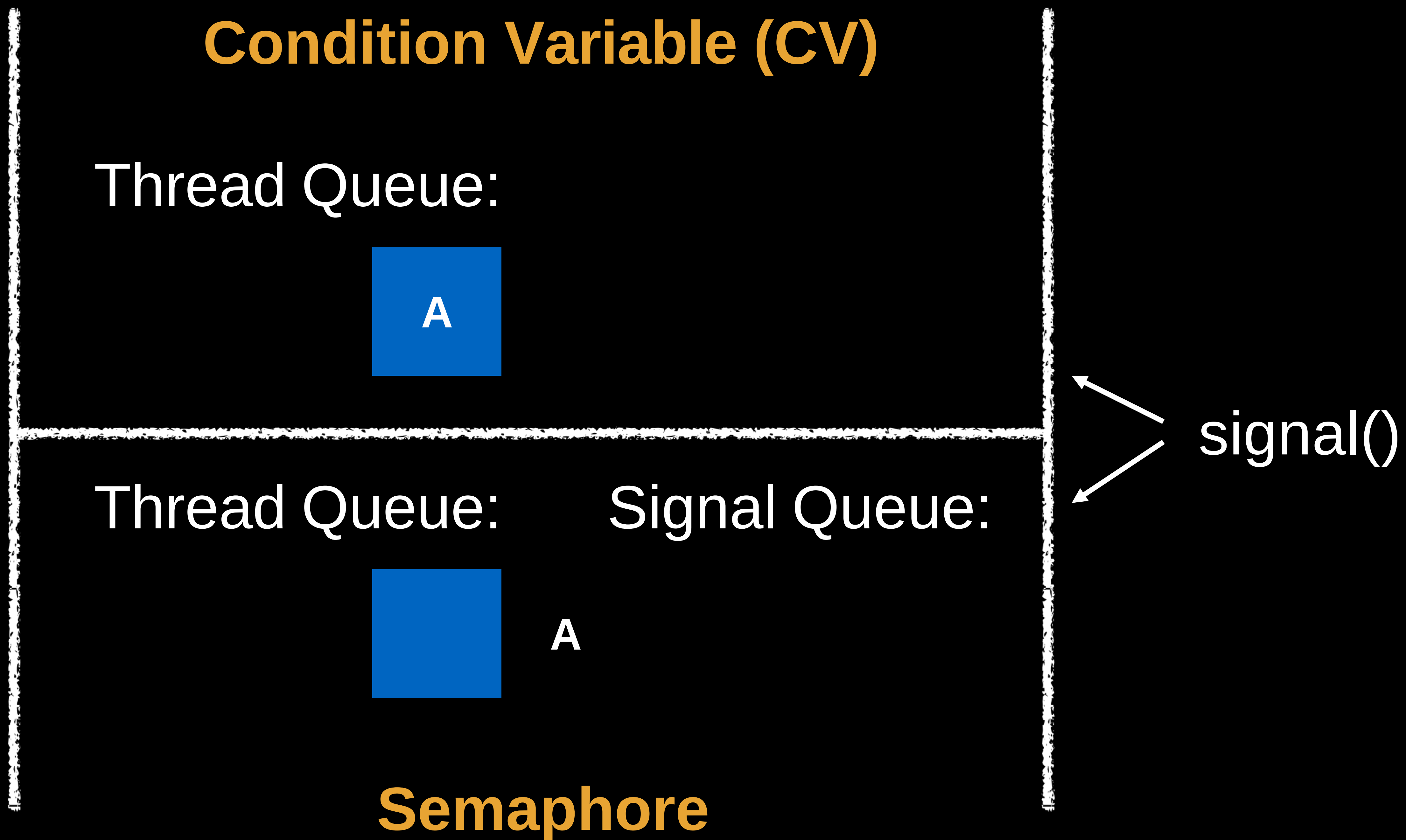
Thread Queue:

Signal Queue:

Semaphore


Condition Variable (CV)
Thread Queue:
A
wait()
Thread Queue:
Signal Queue:
A
Semaphore


Condition Variable (CV)
Thread Queue:
A
Thread Queue:
Signal Queue:
A
Semaphore


Condition Variable (CV)
Thread Queue:
A
signal()
Thread Queue:
Signal Queue:
A
Semaphore

Condition Variable (CV)

Thread Queue:

signal()

Thread Queue: Signal Queue:

Semaphore

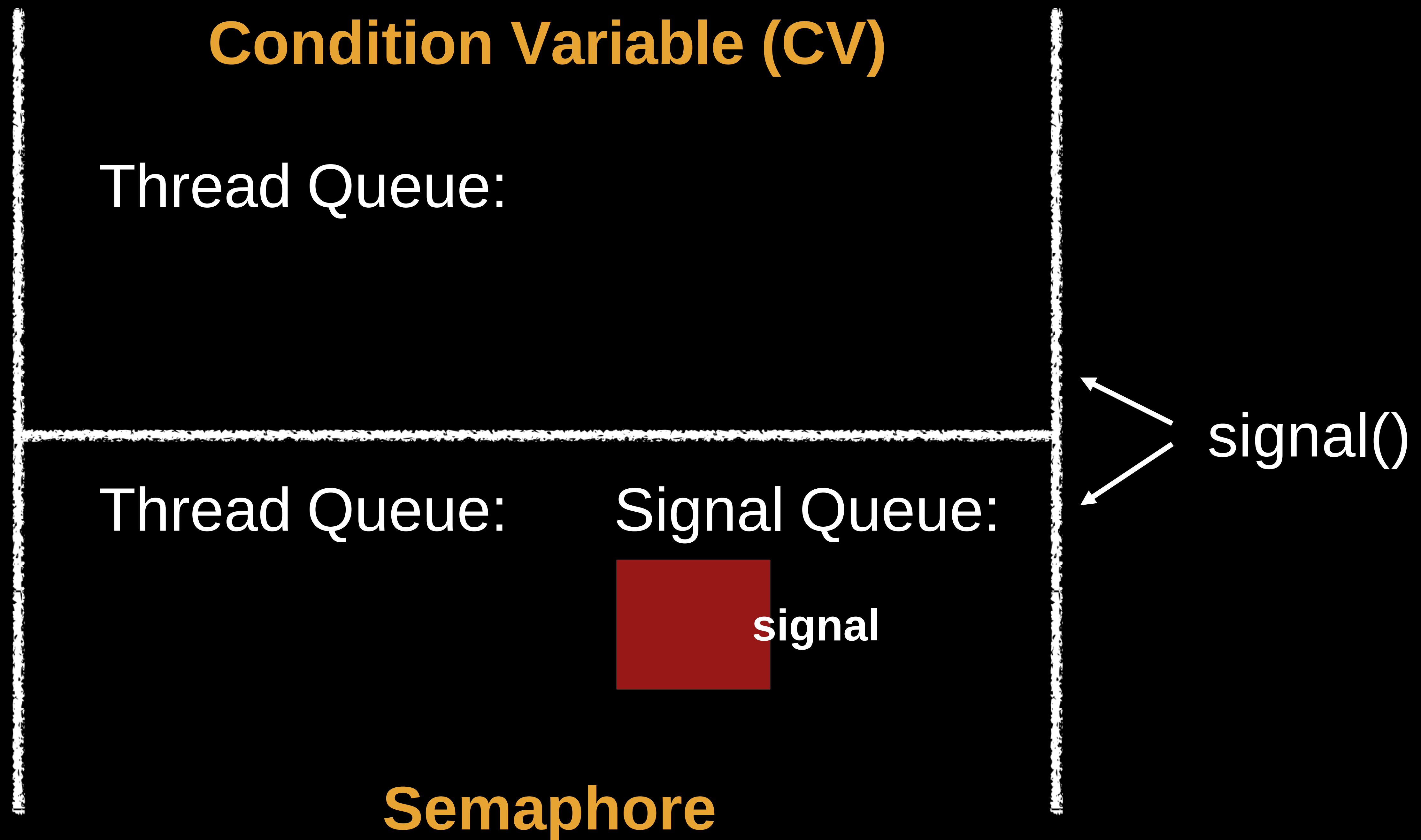
Condition Variable (CV)

Thread Queue:

Thread Queue:

Signal Queue:

Semaphore


Condition Variable (CV)
Thread Queue:
signal()
Thread Queue:
Signal Queue:
signal
Semaphore

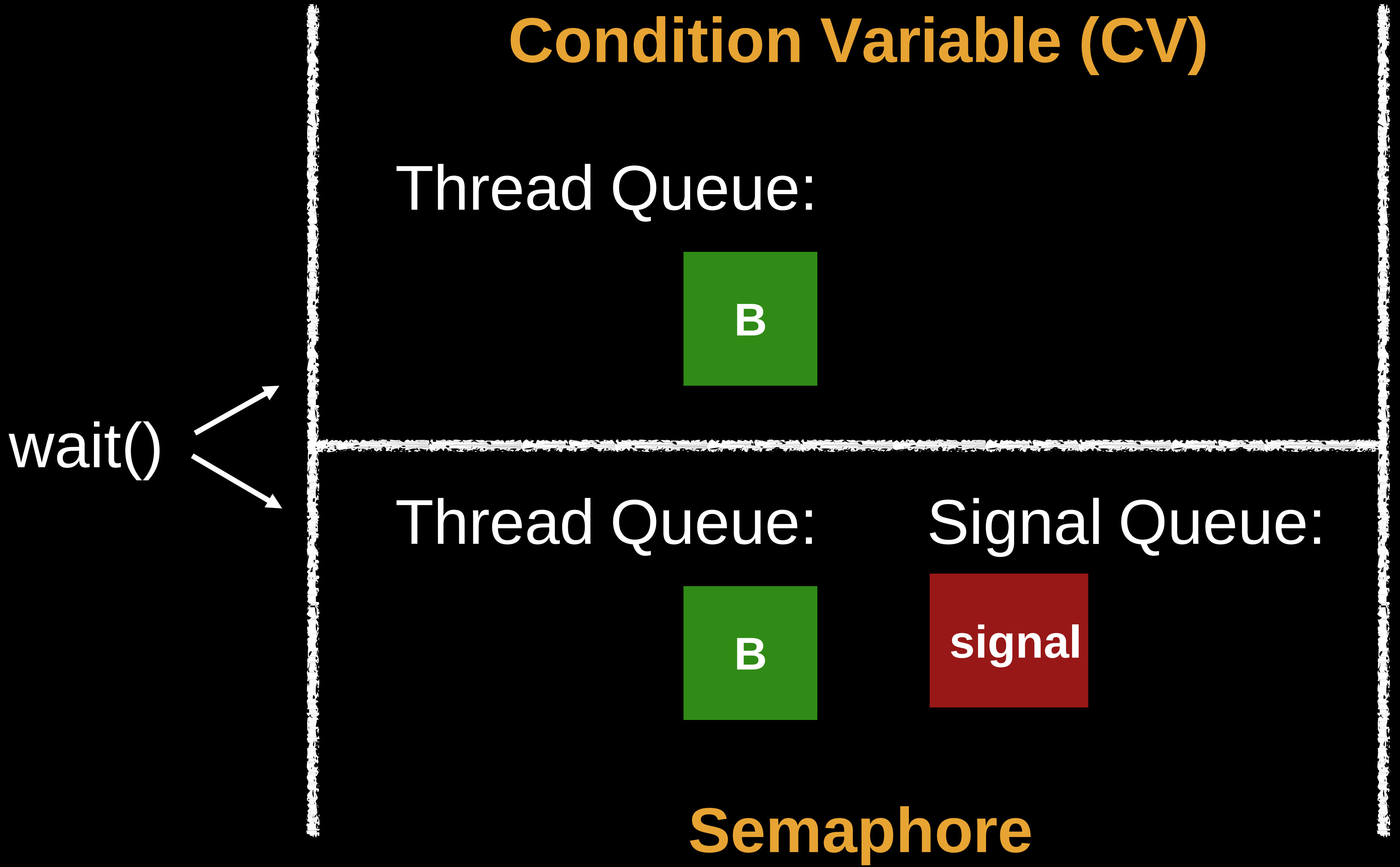
Condition Variable (CV)

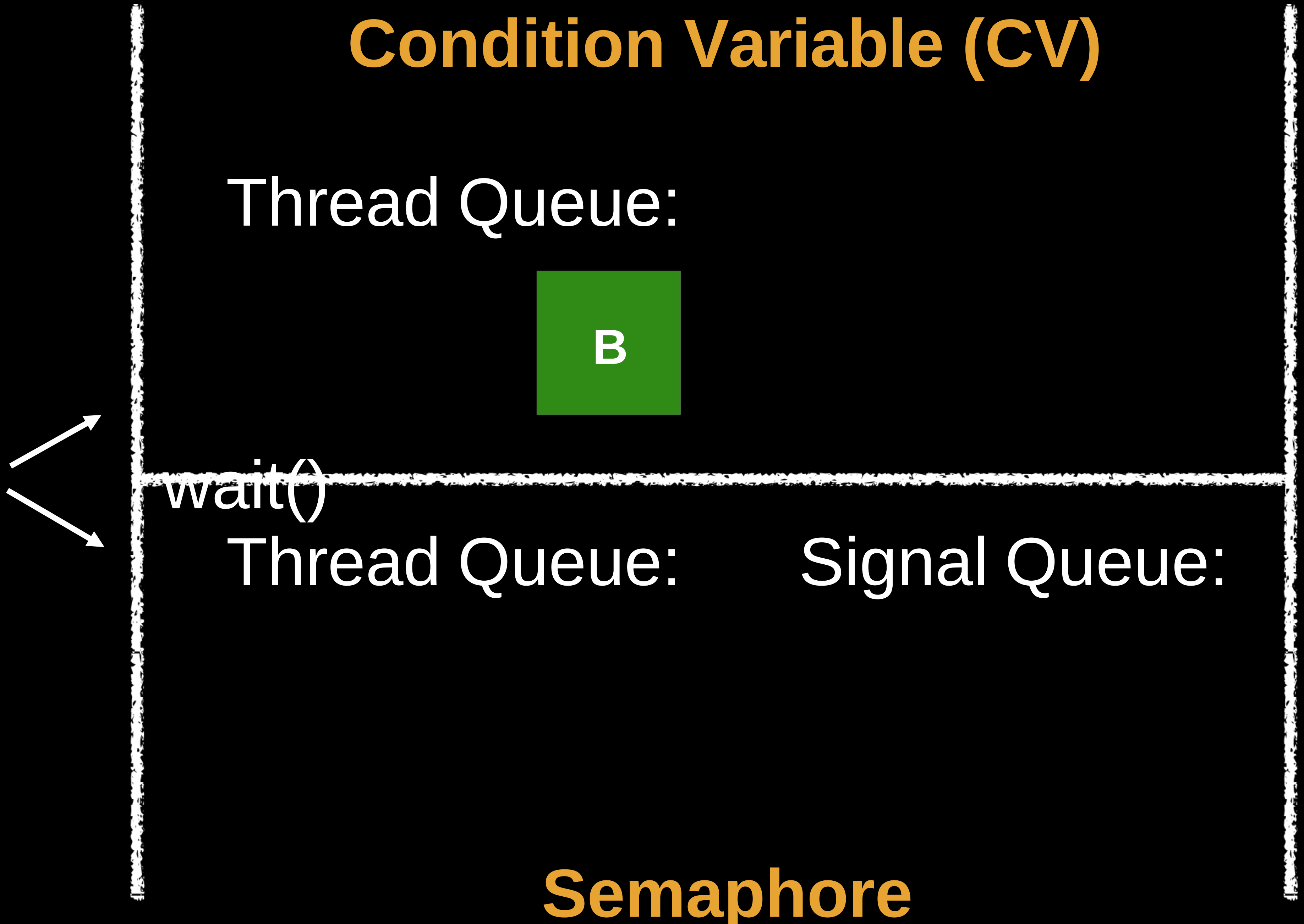
Thread Queue:

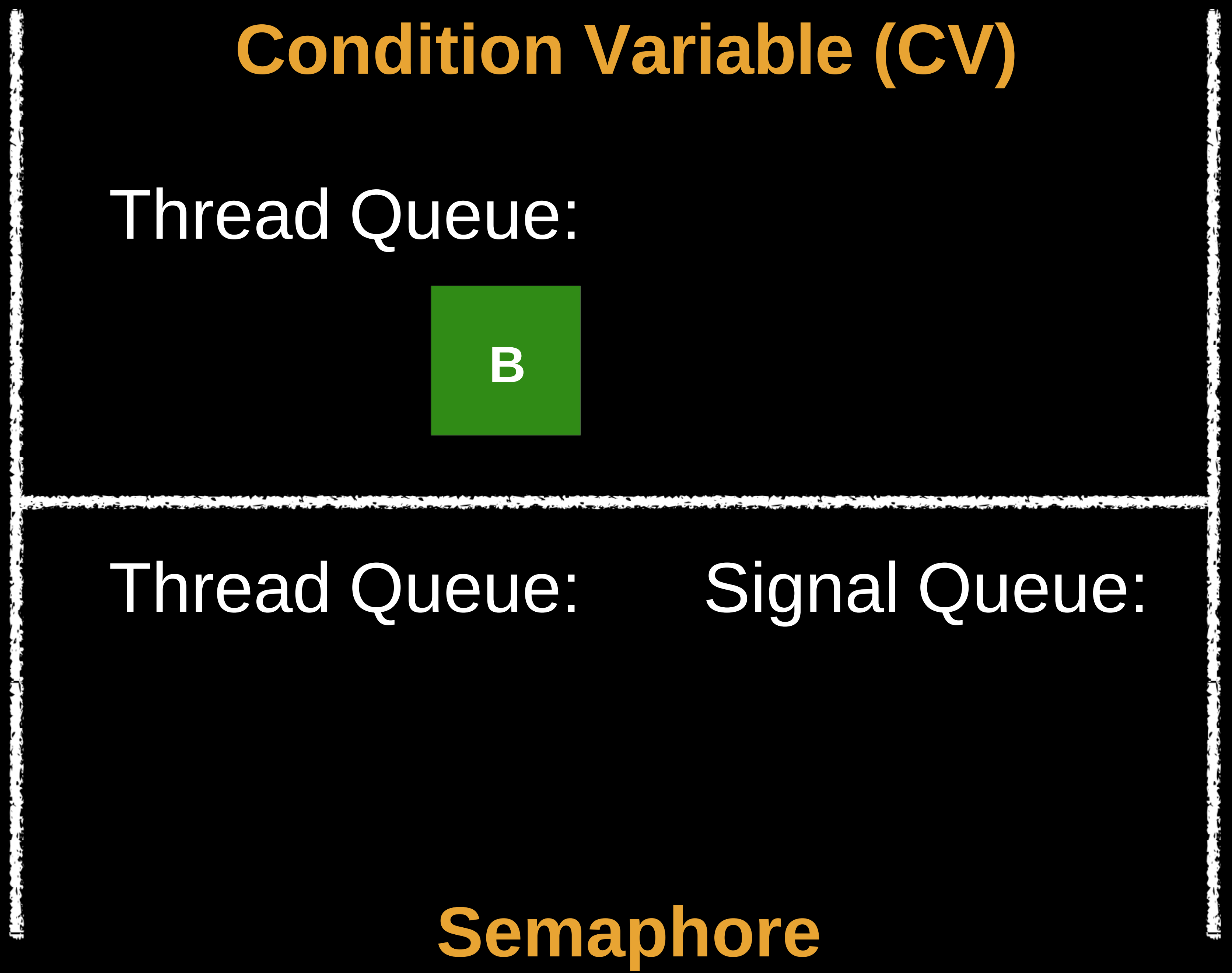
Thread Queue:

Signal Queue:

Semaphore


Condition Variable (CV)
Thread Queue:
B
wait()
Thread Queue:
Signal Queue:
B
signal
Semaphore


Condition Variable (CV)
Thread Queue:
B
wait()
Thread Queue:
Signal Queue:
Semaphore


Condition Variable (CV)
Thread Queue:
B
Thread Queue:
Signal Queue:
Semaphore

Condition Variable (CV)

Thread Queue:

B

may wait forever
(if not careful)

Thread Queue:

Signal Queue:

Semaphore

Condition Variable (CV)

Thread Queue:

B

may wait forever
(if not careful)

Thread ~~Signal Queue:~~

just use counter

Semaphore

Join w/ CV

```
int done      = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
    printf("child\n");
    Mutex_lock(&m);
    done = 1;
    cond_signal(&c);
    Mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    pthread_t       c;
    printf("parent:   begin\n");
    Pthread_create(c, NULL, child, NULL);
    Mutex_lock(&m);
    while(done == 0)
    Cond_wait(&c, &m);
    Mutex_unlock(&m);
    printf("parent: end\n");
}
```

Join w/ CV

```
int done = 0;
mutex_t m = MUTEX_INIT;
  cond_t c = COND_INIT;
```

extra state and mutex

```
void *child(void *arg) {
     printf("child\n");
      Mutex_lock(&m);
        done = 1;
    cond_signal(&c);
    Mutex_unlock(&m);
}
```

locks around state/signal

```
int main(int argc, char *argv[]) {
    pthread_t          c;
    printf("parent:   begin\n");
    Pthread_create(c, NULL, child, NULL);
     Mutex_lock(&m);
     while(done == 0)
         Cond_wait(&c, &m);
    Mutex_unlock(&m);
    printf("parent: end\n");
              }
```

while loop for checking state

Join w/ CV

```
int done      = 0;
mutex_t m = MUTEX_INIT;
cond_t c = COND_INIT;
void *child(void *arg) {
     printf("child\n");
      Mutex_lock(&m);
       done = 1;
     cond_signal(&c);
    Mutex_unlock(&m);
            }

int main(int argc, char *argv[]) {
    pthread_t          c;
    printf("parent:   begin\n");
    Pthread_create(c, NULL, child, NULL);
        Mutex_lock(&m);
        while(done == 0)
        Cond_wait(&c, &m);
      Mutex_unlock(&m);
    printf("parent: end\n");
             }
```

Join w/ Semaphore

```
        sem_t s;
void *child(void *arg) {
    printf("child\n");
      sem_post(&s);
          }

int main(int argc, char *argv[]) {
    sem_init(&s, 0);
     pthread_t c;
    printf("parent: begin\n");
    Pthread_create(c, NULL, child, NULL);
      sem_wait(&s);
    printf("parent: end\n");
             }
```

Semaphore Uses

For the following init's, what might the use be?

(a) sem_init(&s, **0**);

(b) sem_init(&s, **1**);

(c) sem_init(&s, **N**);

Producer/Consumer

How many semaphores do we need?

Producer/Consumer

How many semaphores do we need?

```
Sem_init(&empty  max); // max are empty
,
Sem_init(&full, 0);    // 0 are full
Sem_init(&mutex  1);   // mutex
,
```

Producer/Consumer

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Sem_wait(&empty);
    Sem_wait(&mutex);
      do_fill(i);
    Sem_post(&mutex);
    Sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
      Sem_wait(&full);
      Sem_wait(&mutex);
       tmp = do_get();
      Sem_post(&mutex);
      Sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

Producer/Consumer

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Sem_wait(&empty);
    Sem_wait(&mutex);
       do_fill(i);
    Sem_post(&mutex);
     Sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
  while (1) {
      Sem_wait(&full);
     Sem_wait(&mutex);
      tmp = do_get();
     Sem_post(&mutex);
     Sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

Mutual Exclusion

Producer/Consumer

```
void *producer(void *arg) {
  for (int i = 0; i < loops; i++) {
    Sem_wait(&empty);
    Sem_wait(&mutex);
       do_fill(i);
    Sem_post(&mutex);
     Sem_post(&full);
  }
}
```

```
void *consumer(void *arg) {
     while (1) {
      Sem_wait(&full);
      Sem_wait(&mutex);
       tmp = do_get();
      Sem_post(&mutex);
      Sem_post(&empty);
    printf("%d\n", tmp);
  }
}
```

Signaling

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 hard ware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The soft ware interlock could fail due to a **race condition**."

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

Getting concurrency right can sometimes save lives!

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

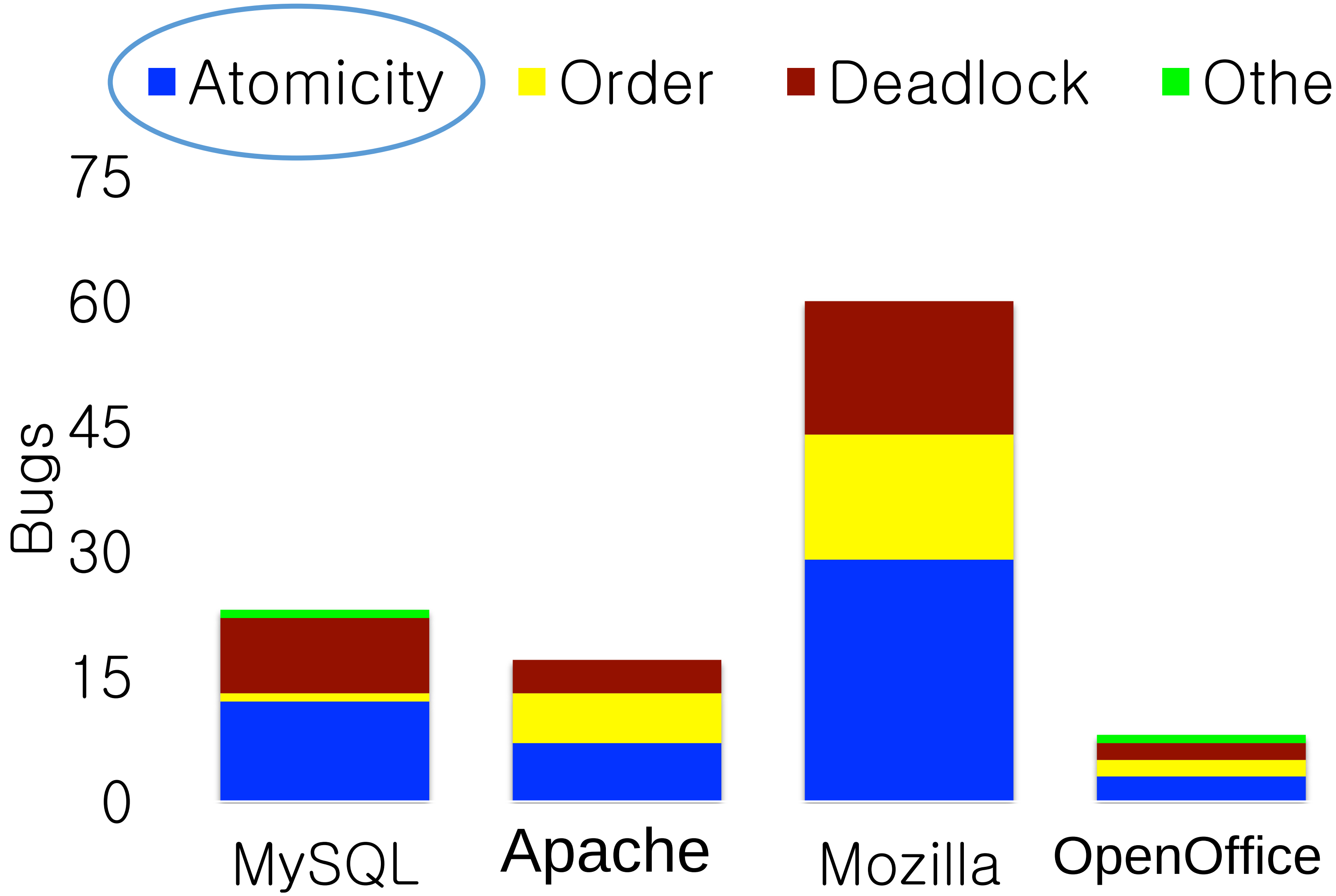
What Types Of Bugs Exist?

- Focus on four major open-source applications
  - MySQL, Apache, Mozilla, OpenOffice.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

Bugs In Modern Applications

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 co ncurrency bugs.

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

Atomicity: MySQL

The desired **serializability** among multiple memory accesses is *violated*.

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 set (writing to thd->proc_info) should be atomic

Fix Atomicity Bugs with Locks

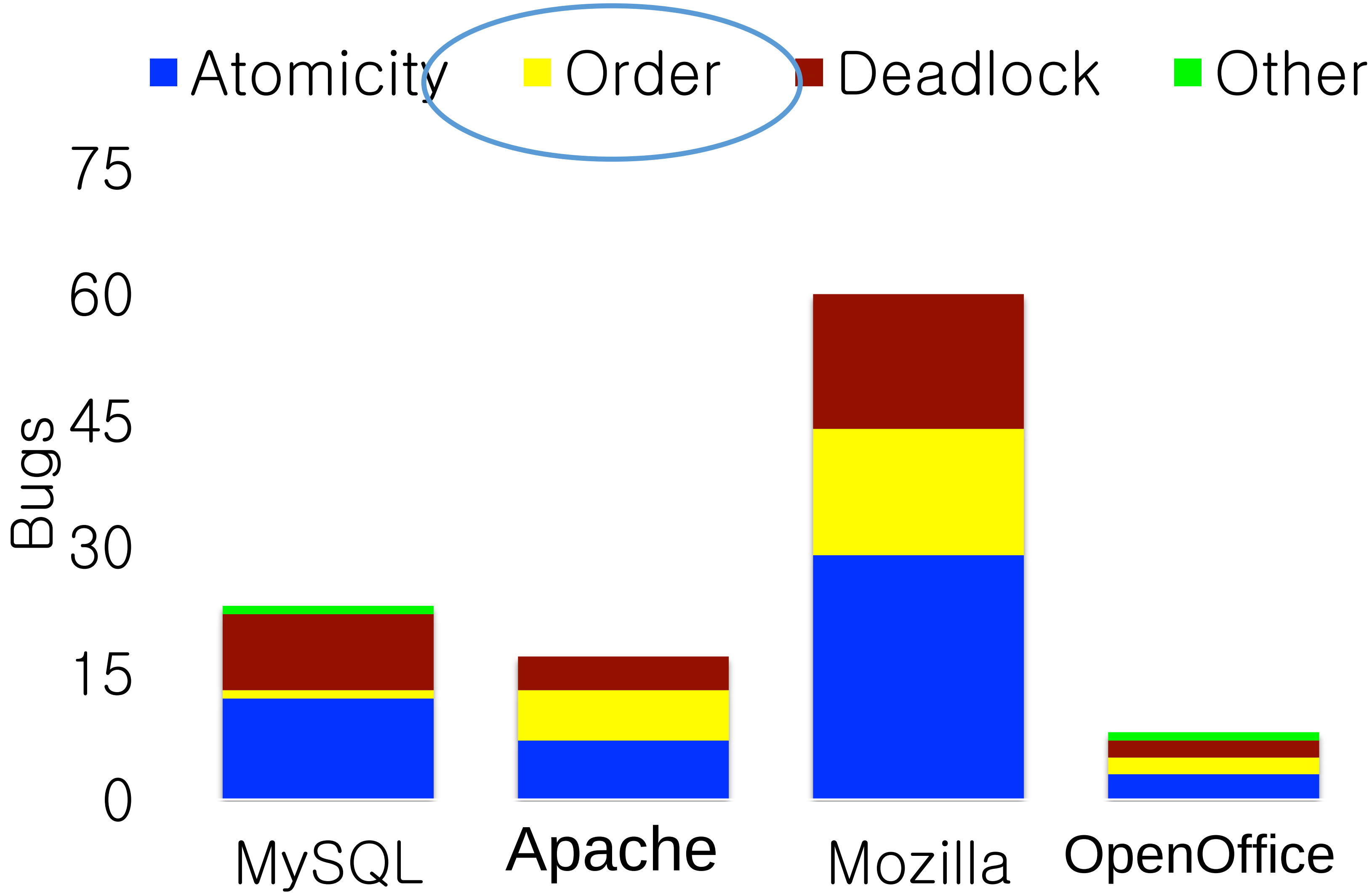
Thread 1:

```
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);
```

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 co ncurrency bugs.

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

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);

    …
}
```

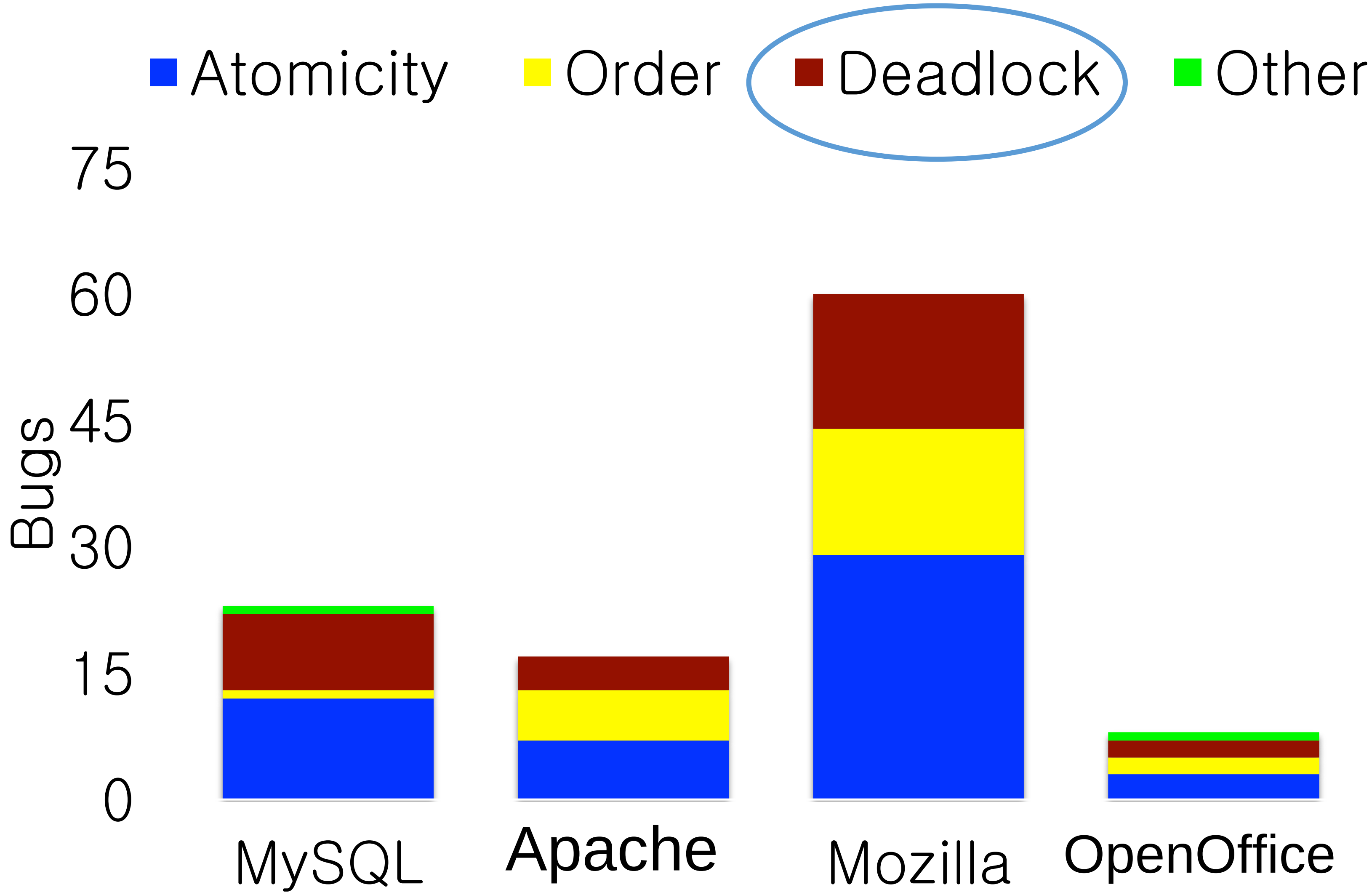
Thread 2:

```
void mMain(…) {
  …

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

  mState = mThread->State;
  …
}
```

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 co ncurrency bugs.

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

Deadlock

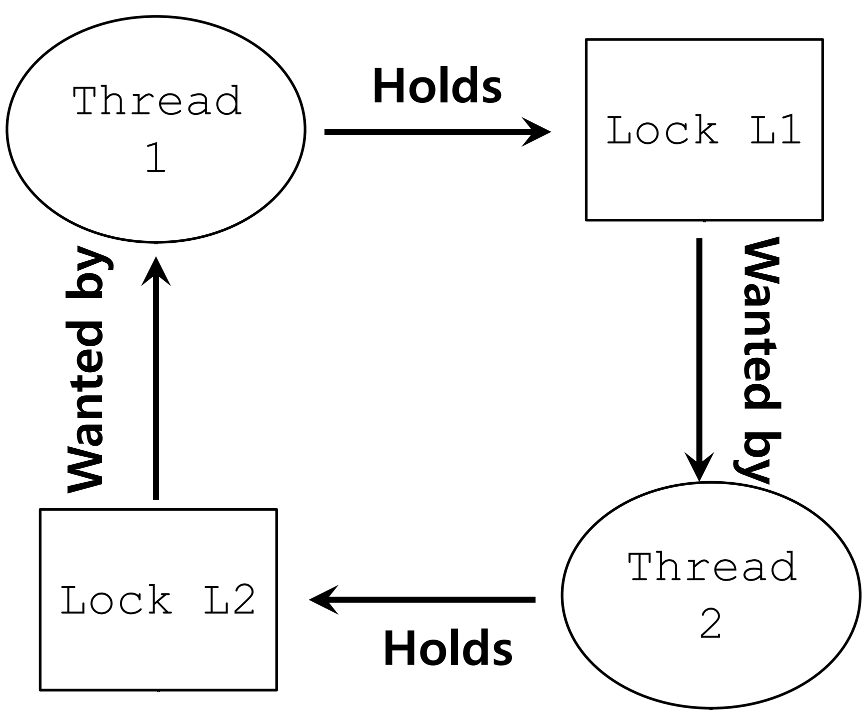
Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

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

Deadlock Bugs

```
Thread 1:    Thread 2:
lock(L1);    lock(L2);
lock(L2);    lock(L1);
```

- The presence of a cycle
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`.
  - `Thread2` that holds lock `L2` is waiting for `L1` to be release.



Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, complex dependencies arise between components.

- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way.
    - Such modularity *does not mesh* well with locking.

Why Do Deadlocks Occur? (Cont.)

- **Example**: Java Vector class and the method `AddAll()`

```
Vector v1,v2;
v1.AddAll(v2);
```

- **Locks** for both the vector being added to (`v1`) and the parameter (`v2`) *need to be acquired*.
  - The routine acquires said locks in some arbitrary order (v1 then v2).
  - If some other thread `calls v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock.

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur.

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

  - If any of these four conditions are not met, **deadlock cannot occur**.

Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies.

- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks at once, atomically.

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

- This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.

- **Problem**:
  - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
  - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.

- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

Prevention – No Preemption (Cont.)

- livelock
  - Both systems are running through the code sequence *over and over again*.
  - Progress is not being made.

  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

```
int CompareAndSwap(int *address, int expected, int new){
  if(*address == expected){
      *address = new;
      return 1; // success
  }
  return 0;
}
```

Prevention – Mutual Exclusion (Cont.)

- We now wanted to atomically increment a value by a certain amount:

```
void AtomicIncrement(int *value, int amount){
    do{
            int old = *value;
    }while( CompareAndSwap(value, old, old+amount)==0);
}
```

- Repeatedly tries to update the value to *the new amount* and uses the `compare-and-swap` to do so.

- **No lock** is acquired
- **No deadlock** can arise
- **livelock** is still a possibility.

Prevention – Mutual Exclusion (Cont.)

- **More complex example**: list insertion

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value    = value ;
    n->next     = head;
    head  = n;
}
```

- If called by multiple threads at the "*same time*", this code has a race condition.

Prevention – Mutual Exclusion (Cont.)

- **Solution**:
  - Surrounding this code with a lock acquire and release.

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    lock(listlock); // begin critical section
    n->next      = head;
    head    = n;
    unlock(listlock) ;  //end critical section
}
```

  - wait-free manner using the `compare-and-swap` instruction

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
           n->next = head;
    } while (CompareAndSwap(&head, n->next, n));
}
```

Deadlock Avoidance via Scheduling

- In some scenarios deadlock avoidance is preferable.

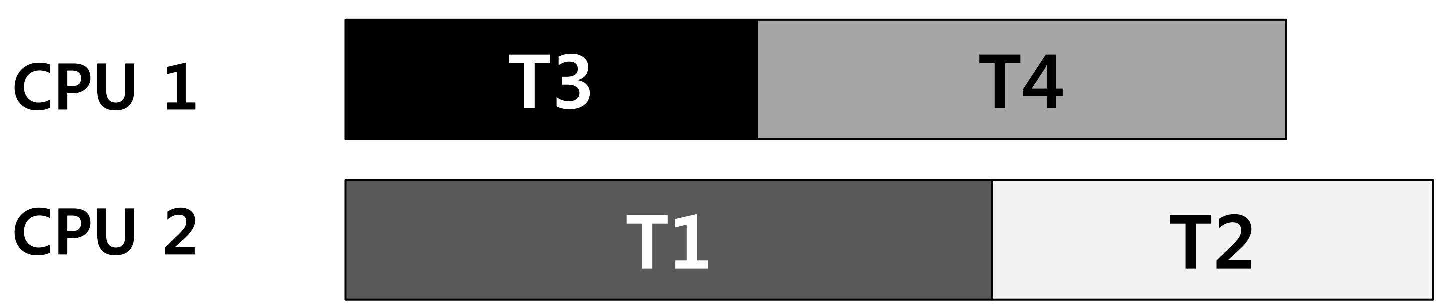
  - **Global knowledge** is required:
    - Which locks various threads might grab during their execution.
    - Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
  - Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

  - A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.

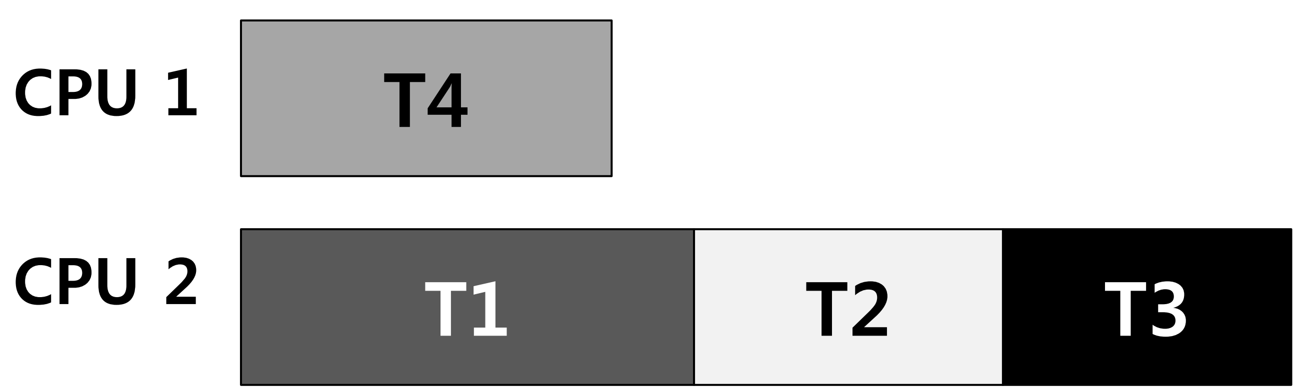


Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

  - A possible schedule that guarantees that *no deadlock* could ever occur.

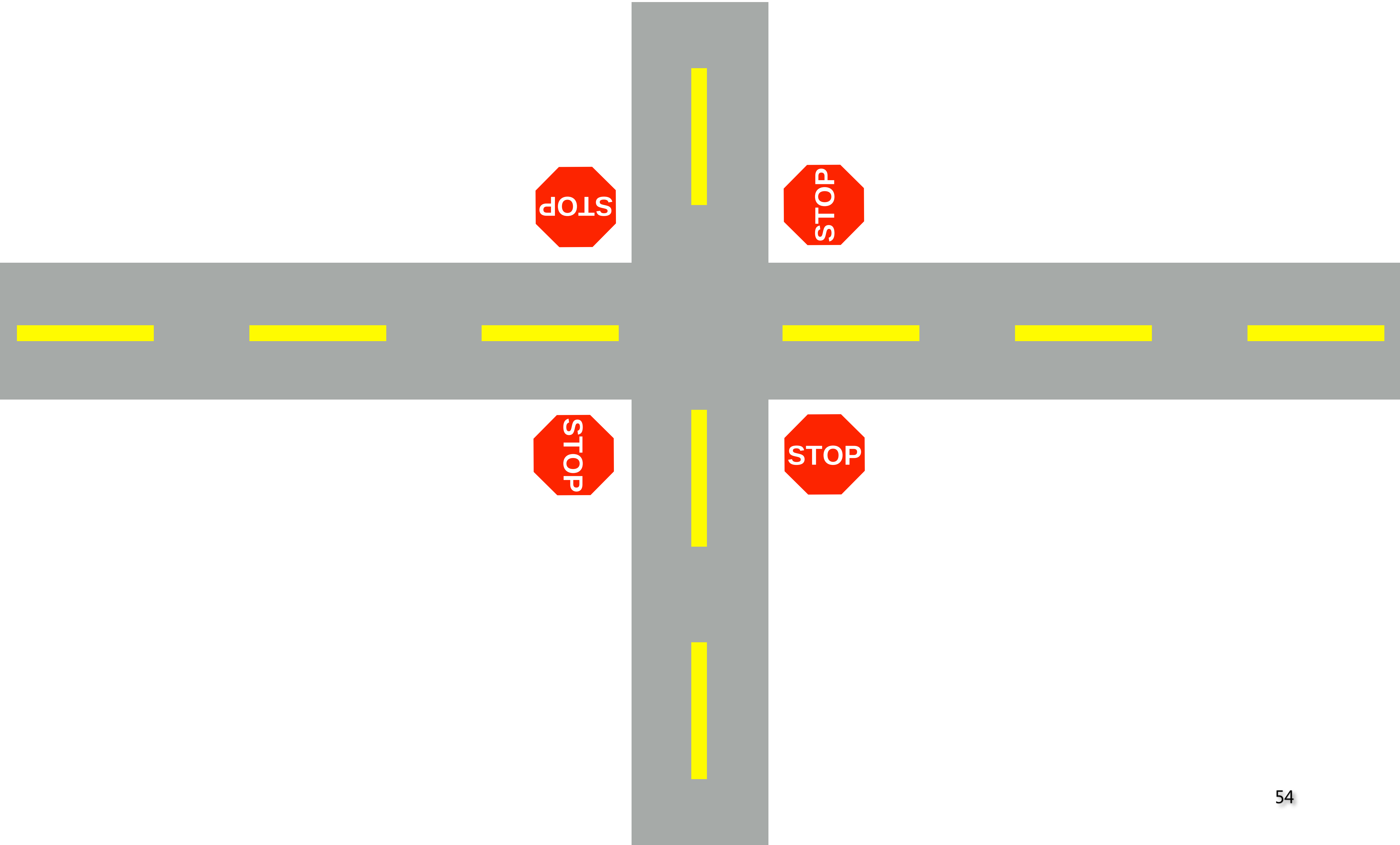


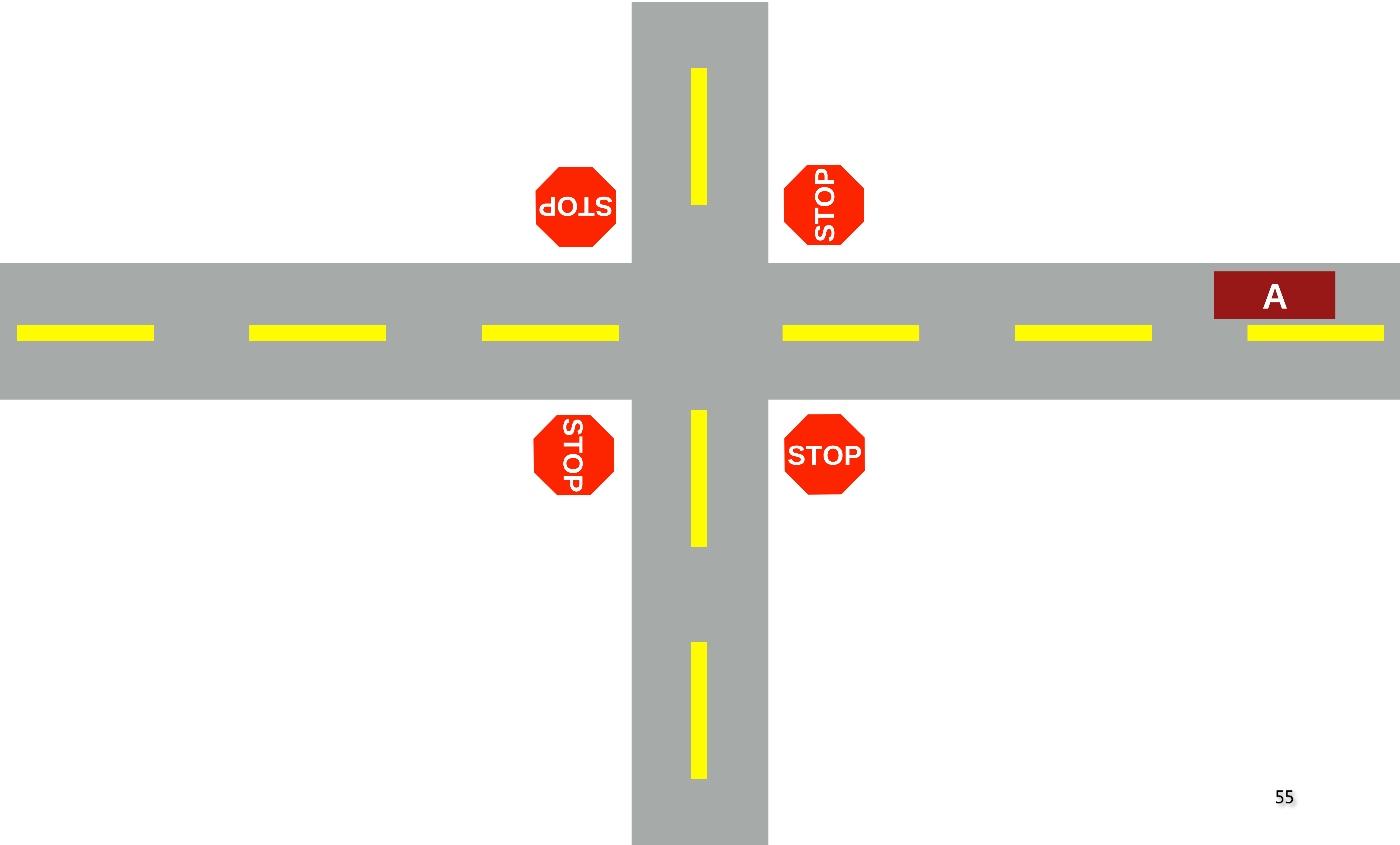
    - The total time to complete the jobs is lengthened considerably.

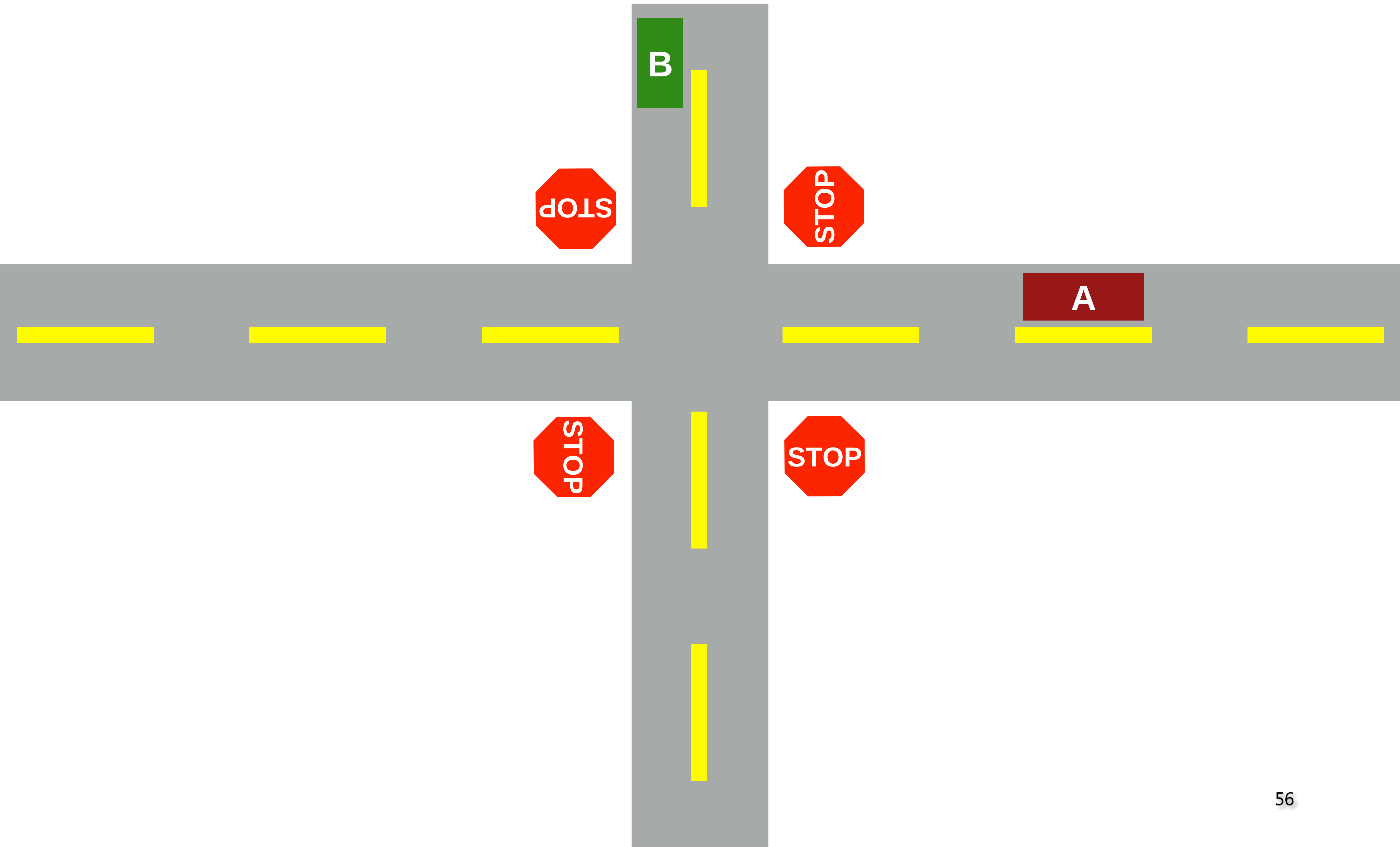
Detect and Recover

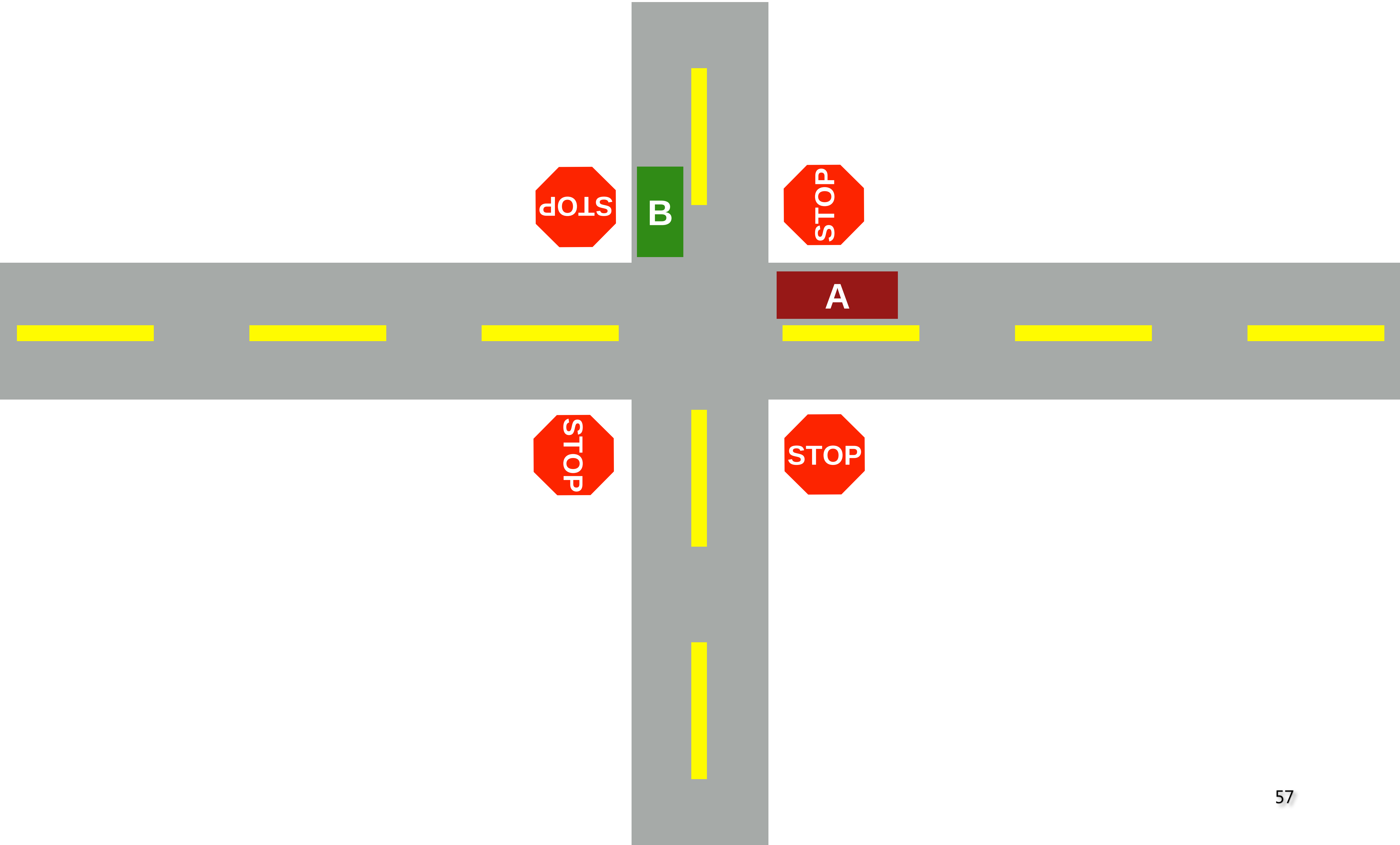
- Allow deadlock to occasionally occur and then *take some action*.
  - **Example**: if an OS froze, you would reboot it.

- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for cycles.
  - In deadlock, the system **need to be restarted**.

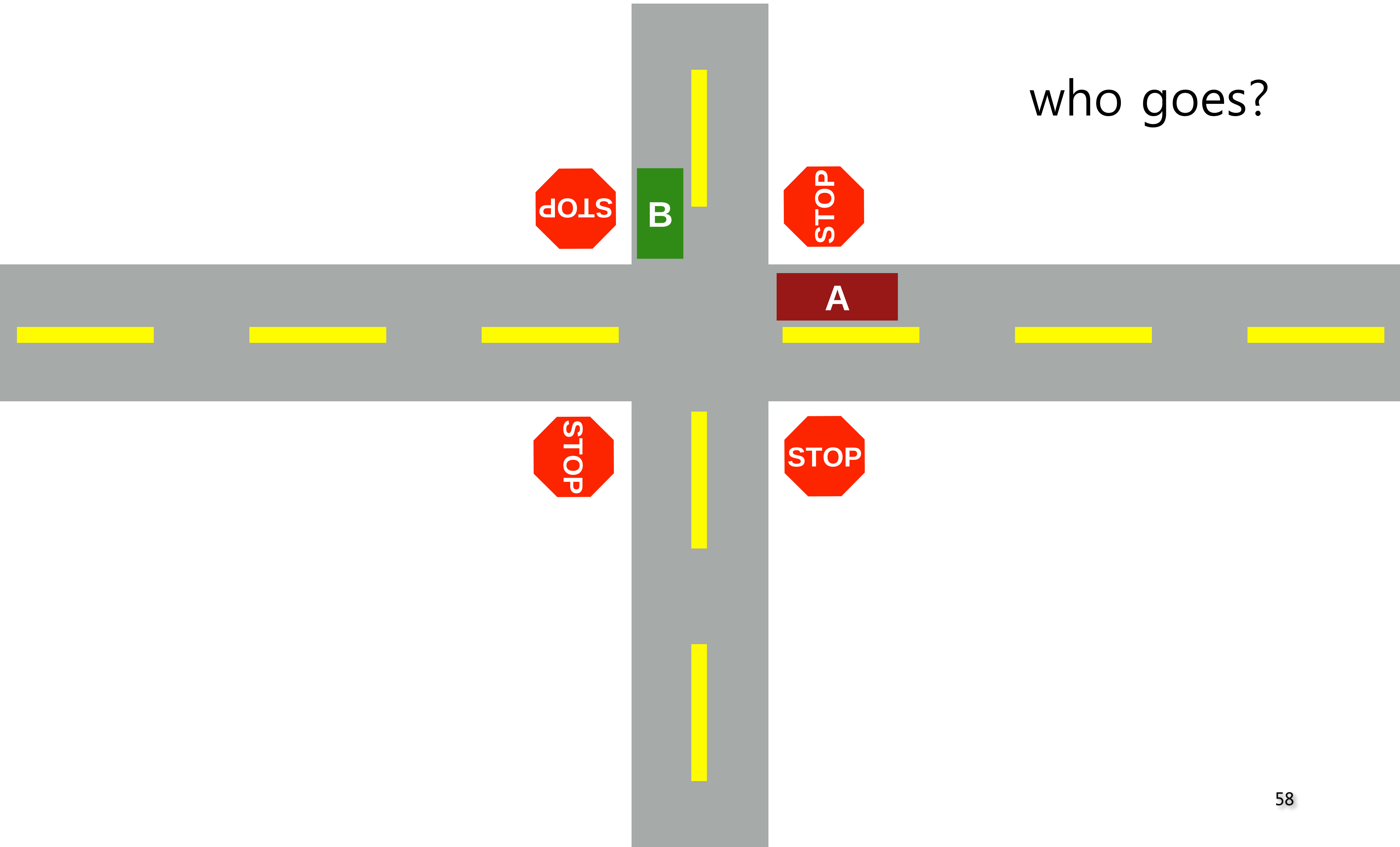

STOP
STOP
STOP
STOP

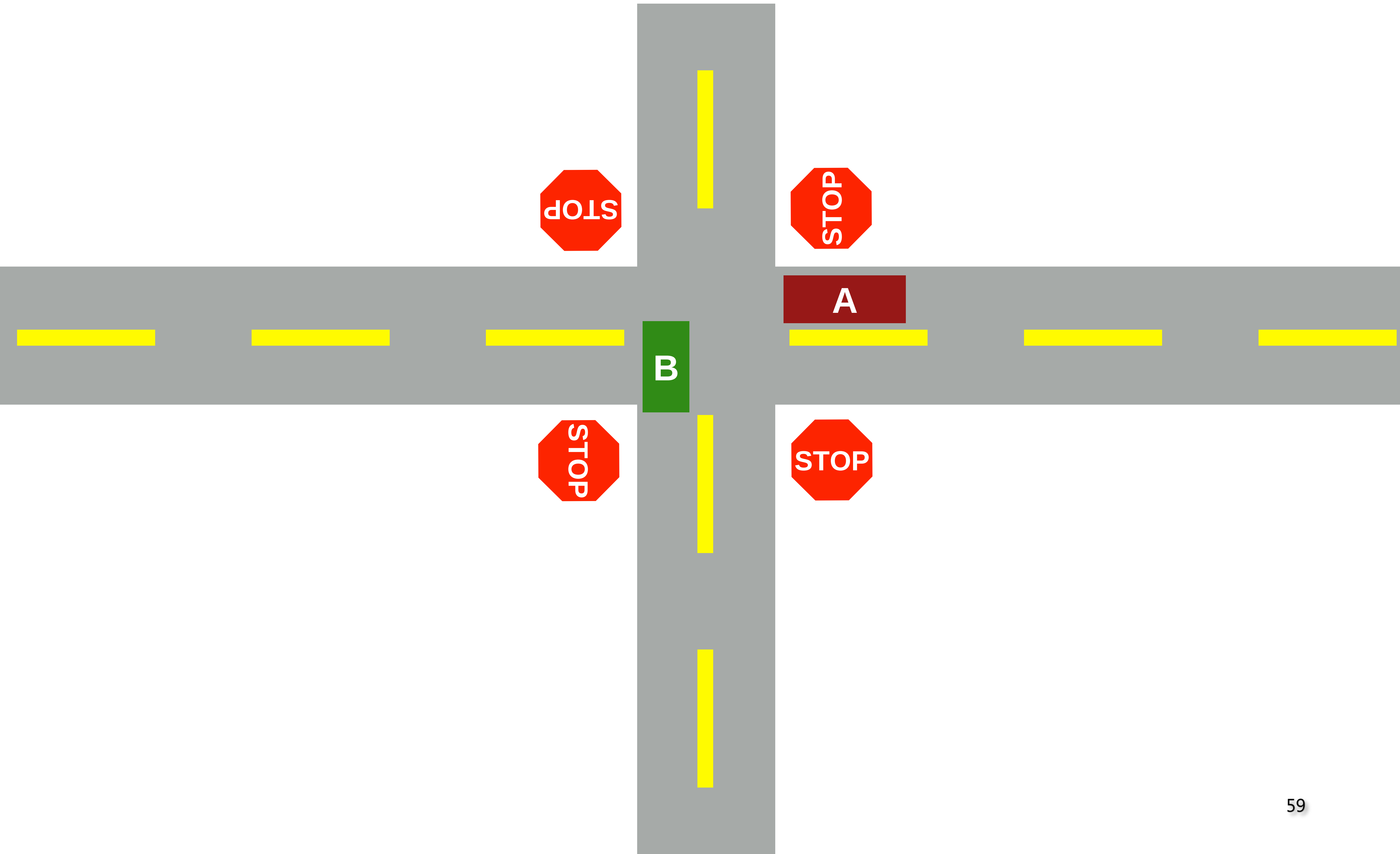

STOP
STOP
A
STOP
STOP

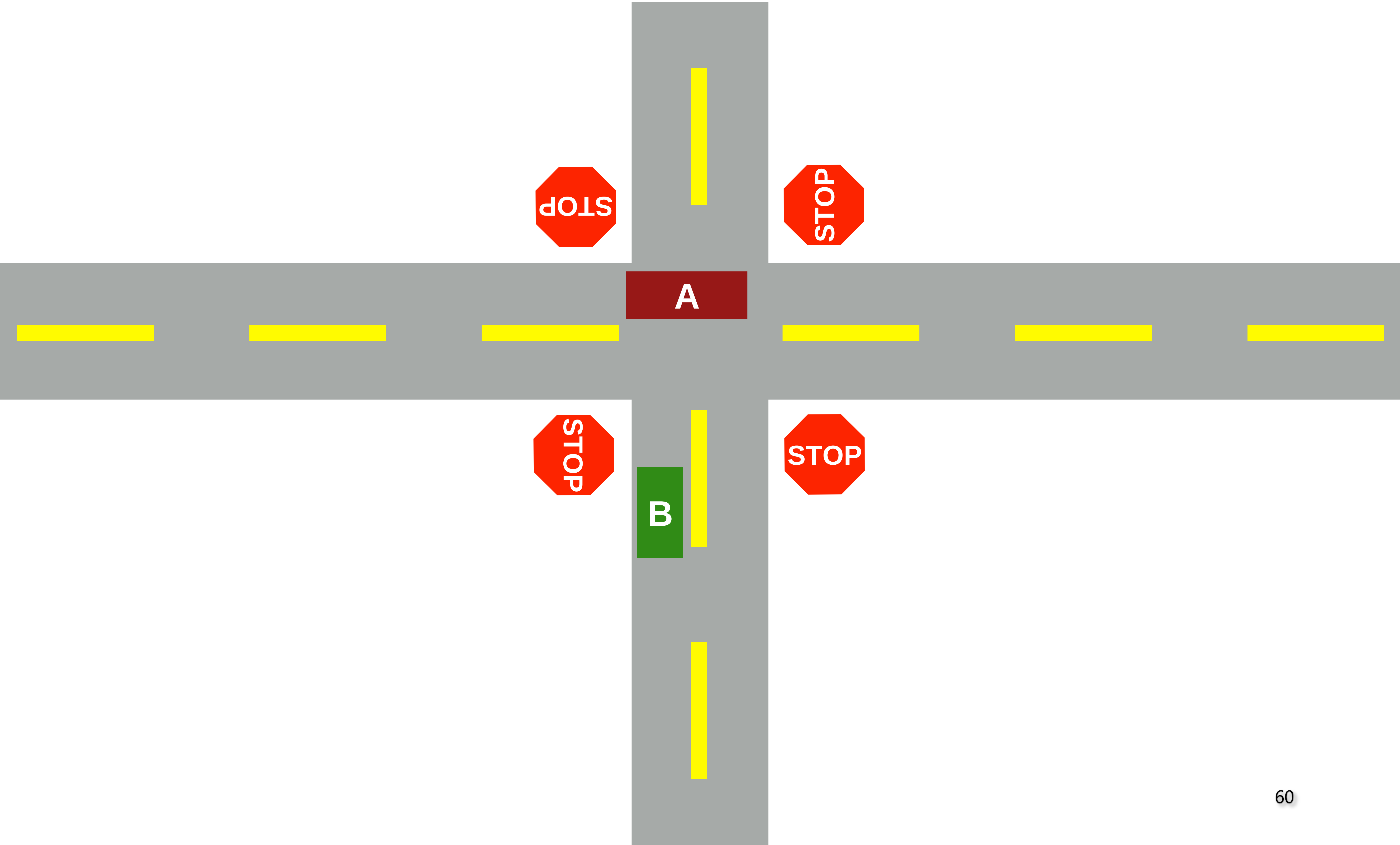

B
STOP
STOP
A
STOP
STOP

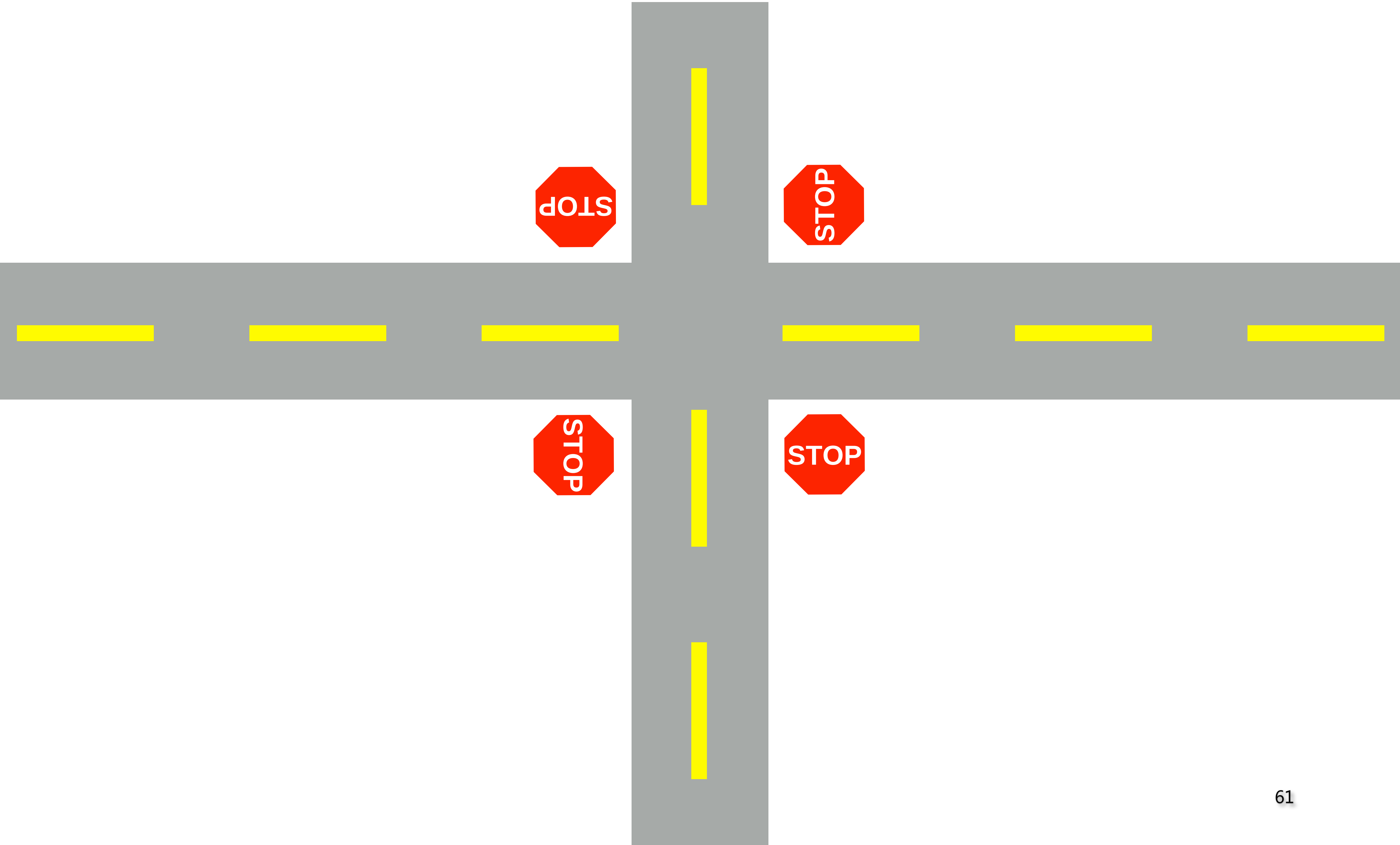

STOP
B
STOP
A
STOP
STOP

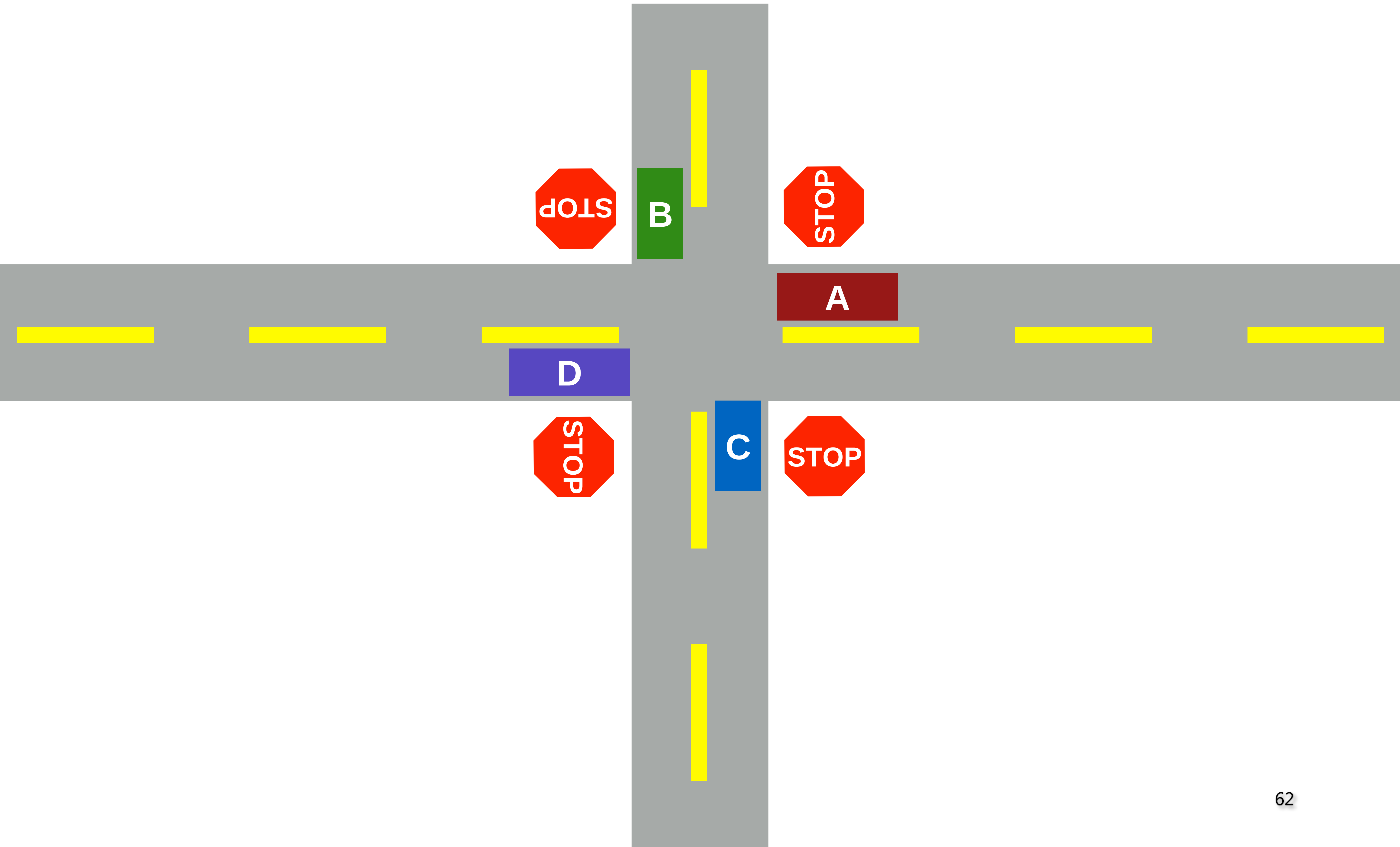
who goes?



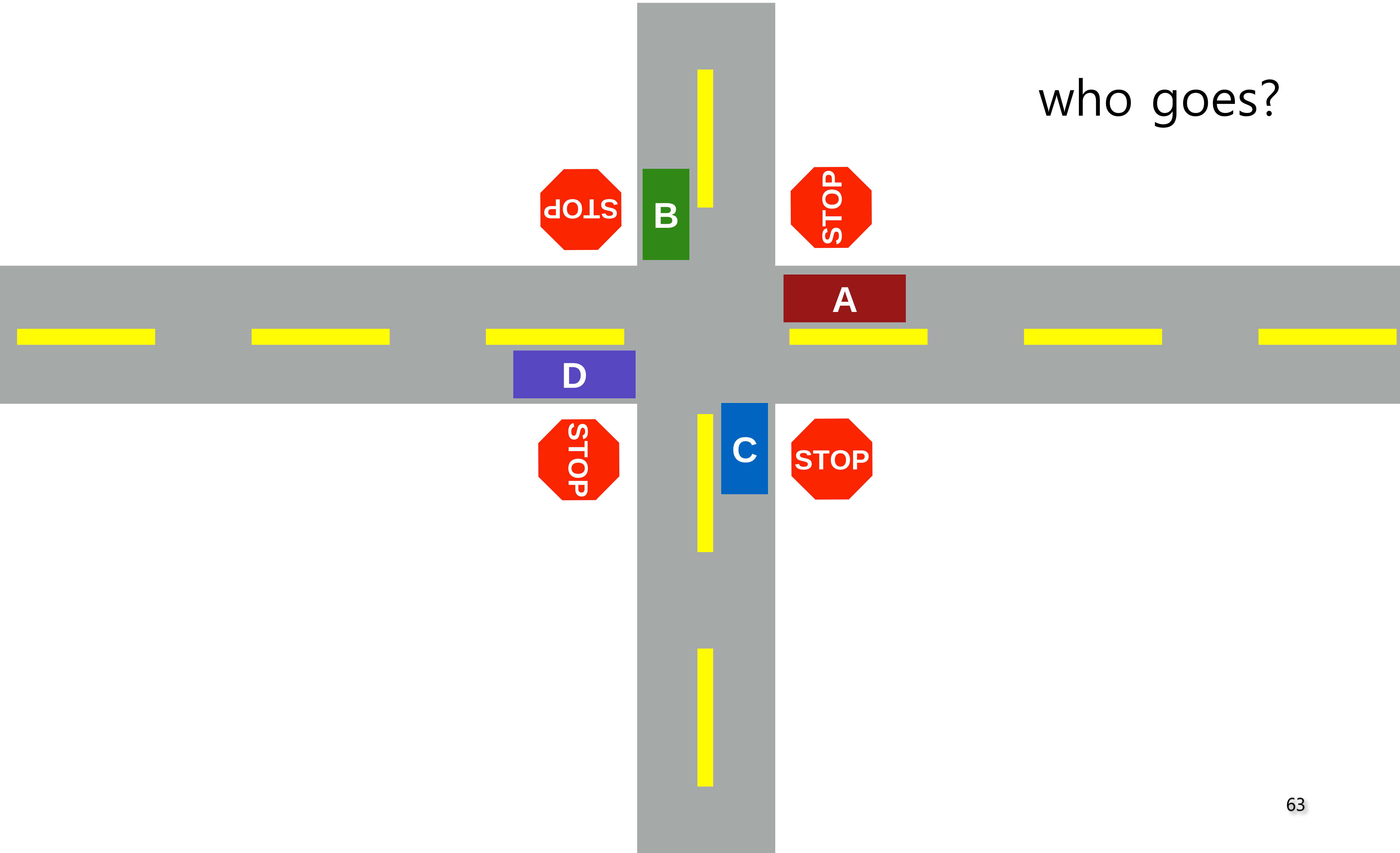

STOP
STOP
A
B
STOP
STOP

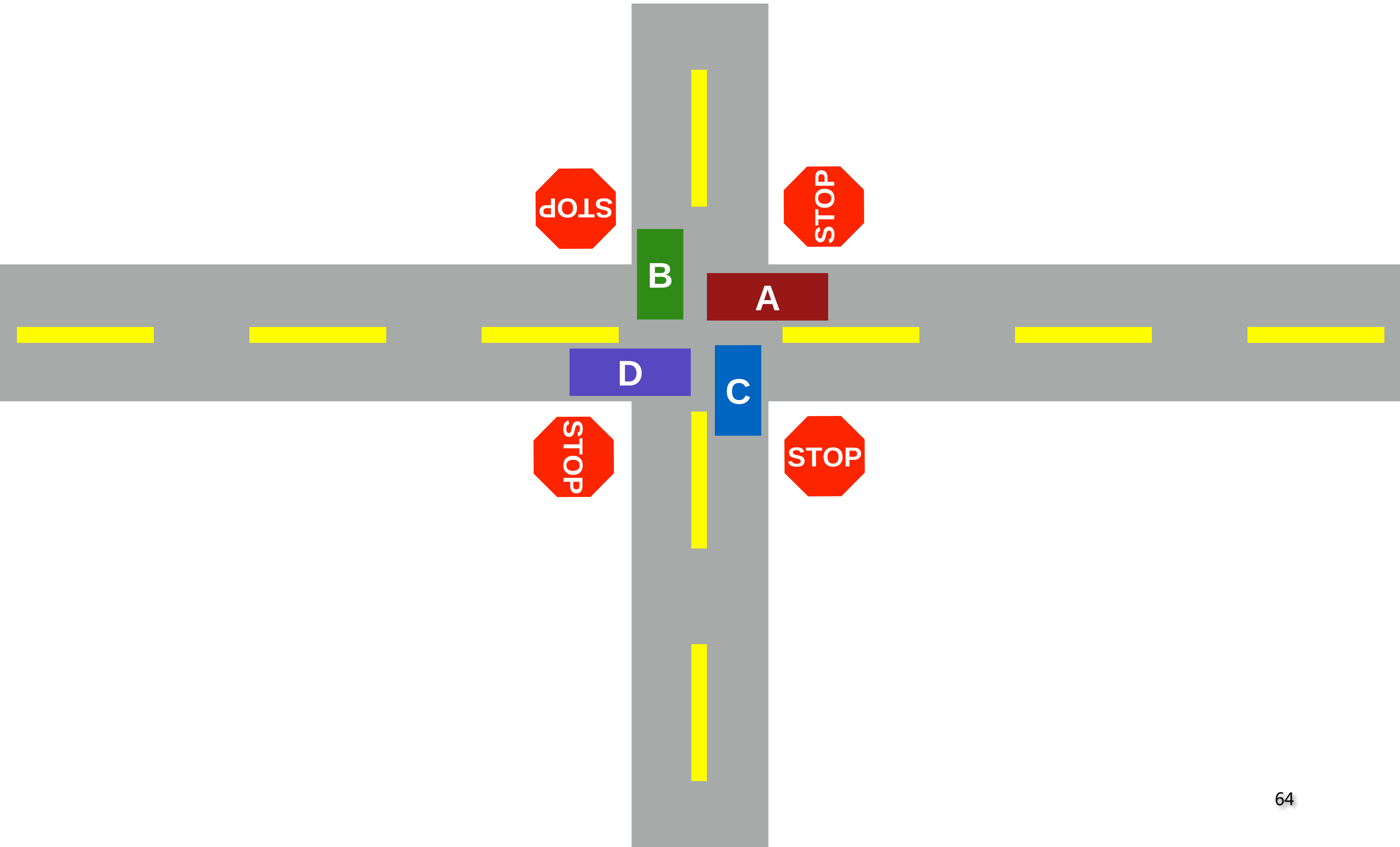

STOP
STOP
A
STOP
STOP
B

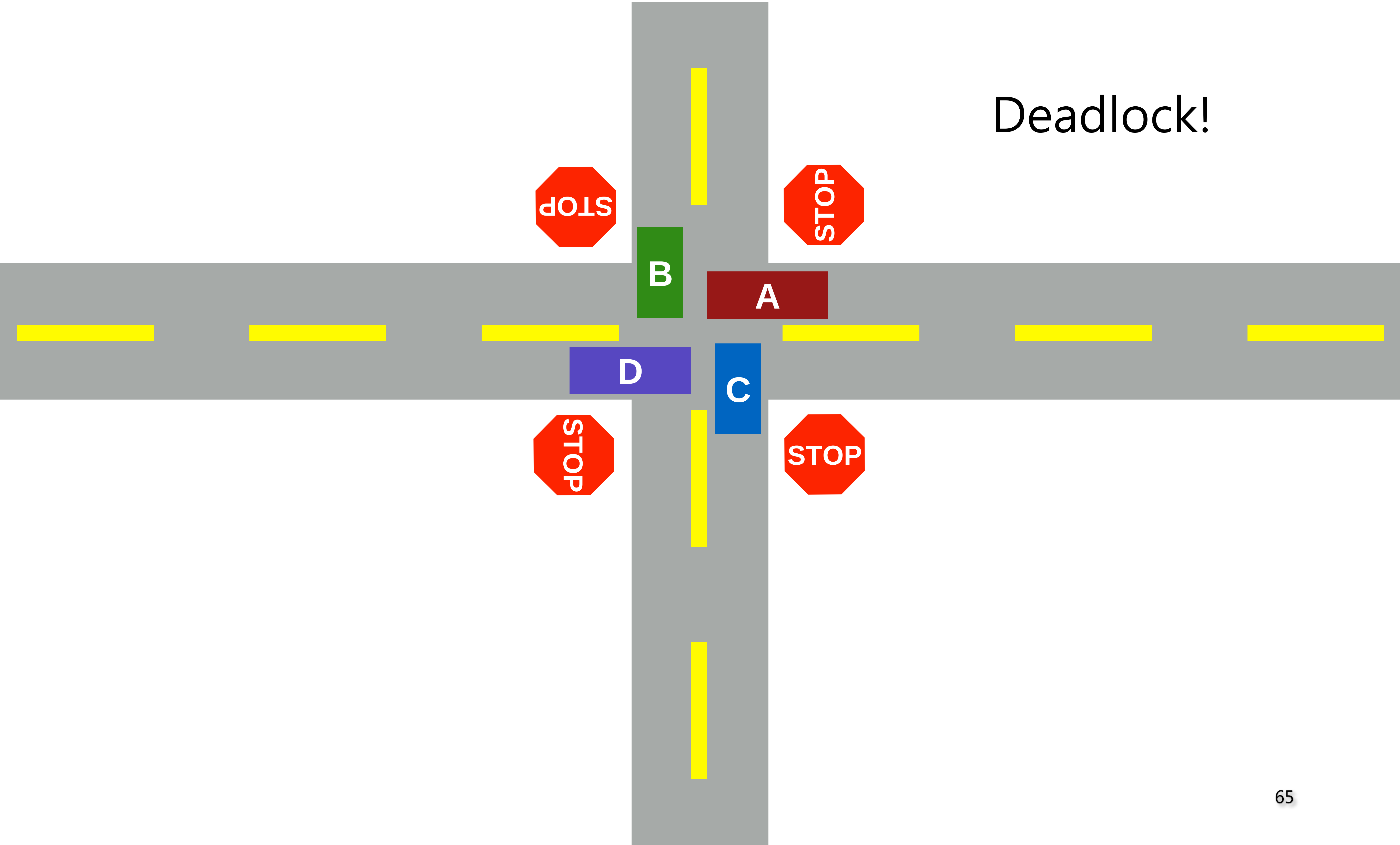

STOP
STOP
STOP
STOP


STOP
B
STOP
A
D
STOP
C
STOP

who goes?




STOP
STOP
B
A
D
C
STOP
STOP



Deadlock!

Boring Code Example

Thread 1 [RUNNING]:

lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:

lock(&B);
lock(&A)

Boring Code Example

Thread 1 [RUNNING]:

lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:

lock(&B);
lock(&A)

Boring Code Example

Thread 1 [RUNNABLE]:

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

Thread 2 [RUNNING]:

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

Boring Code Example

Thread 1 [RUNNABLE]:

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

Thread 2 [RUNNING]:

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

Boring Code Example

Thread 1 [RUNNING]:

lock(&A);
lock(&B)

Thread 2 [RUNNABLE]:

lock(&B);
lock(&A)

Boring Code Example

Thread 1 [SLEEPING]:

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

Thread 2 [RUNNABLE]:

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

Boring Code Example

Thread 1 [SLEEPING]:

lock(&A);
lock(&B)

Thread 2 [RUNNING]:

lock(&B);
lock(&A)

Boring Code Example

Thread 1 [SLEEPING]:

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

Thread 2 [SLEEPING]:

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

Boring Code Example

Thread 1 [SLEEPING]:

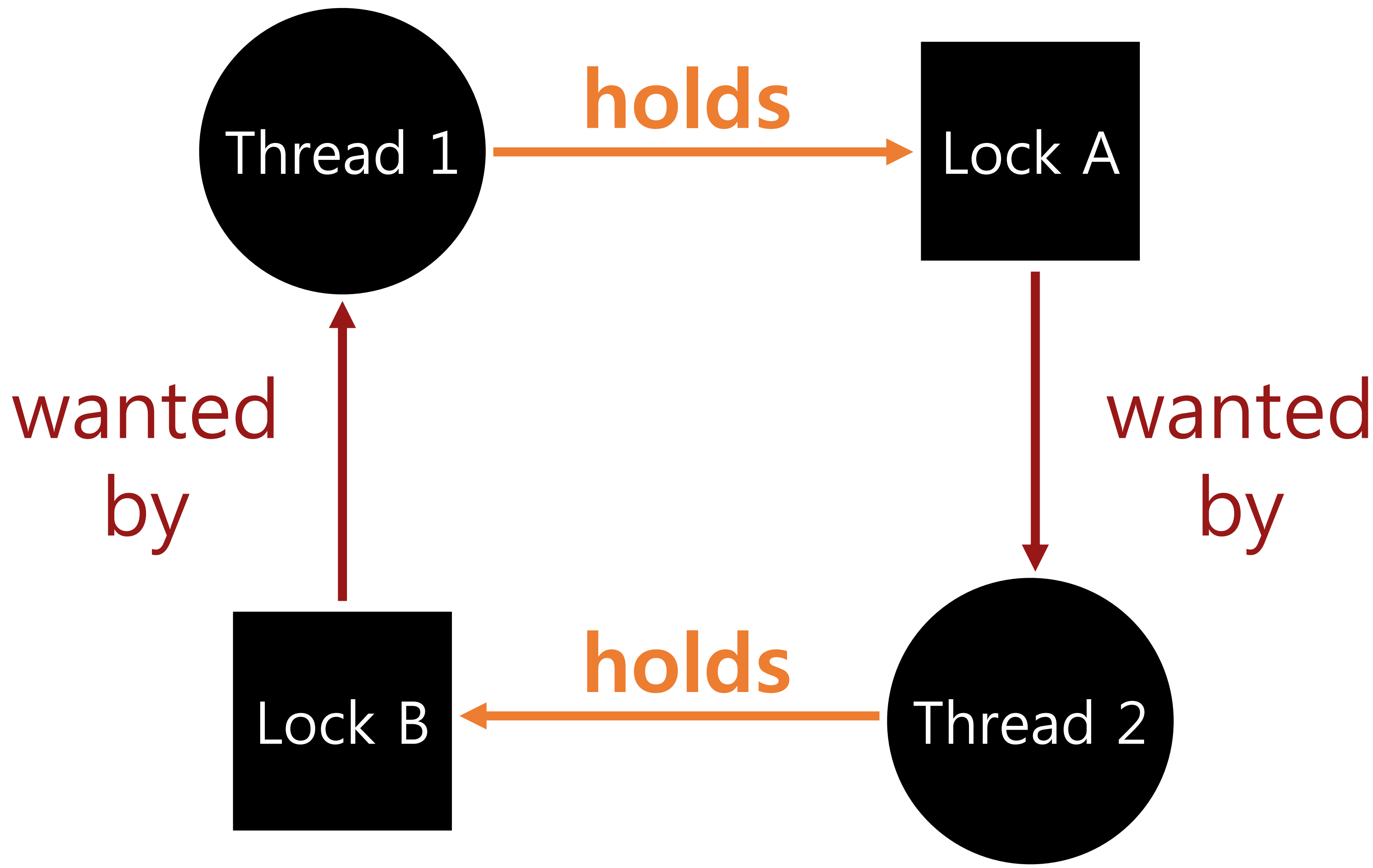
lock(&A);
lock(&B)

Thread 2 [SLEEPING]:

lock(&B);
lock(&A)

Deadlock!

Circular Dependency



Fix Deadlocked Code

Thread 1:

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

Thread 2:

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

How would you fix this code?

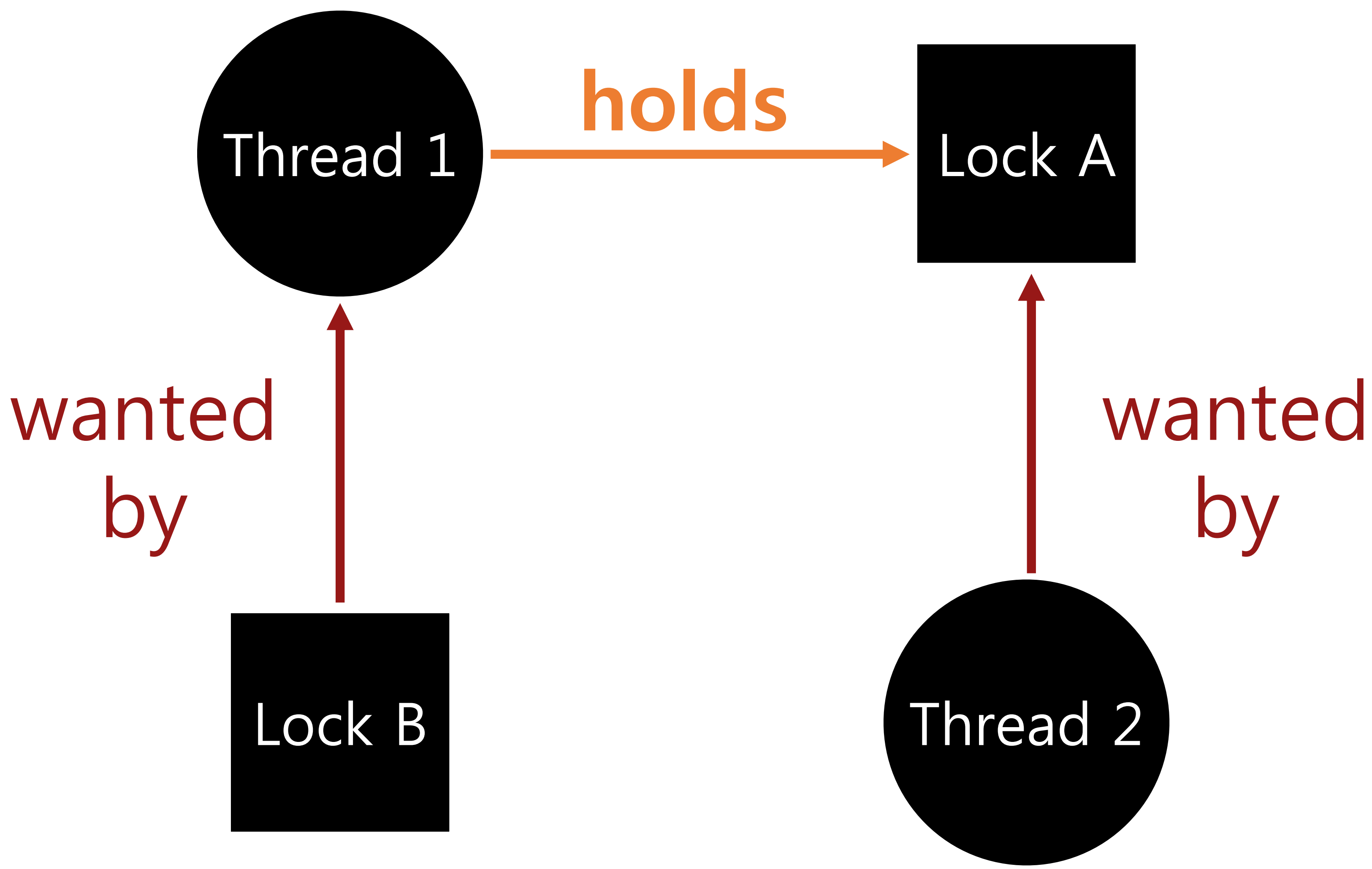
Thread 1

```
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);
}
```

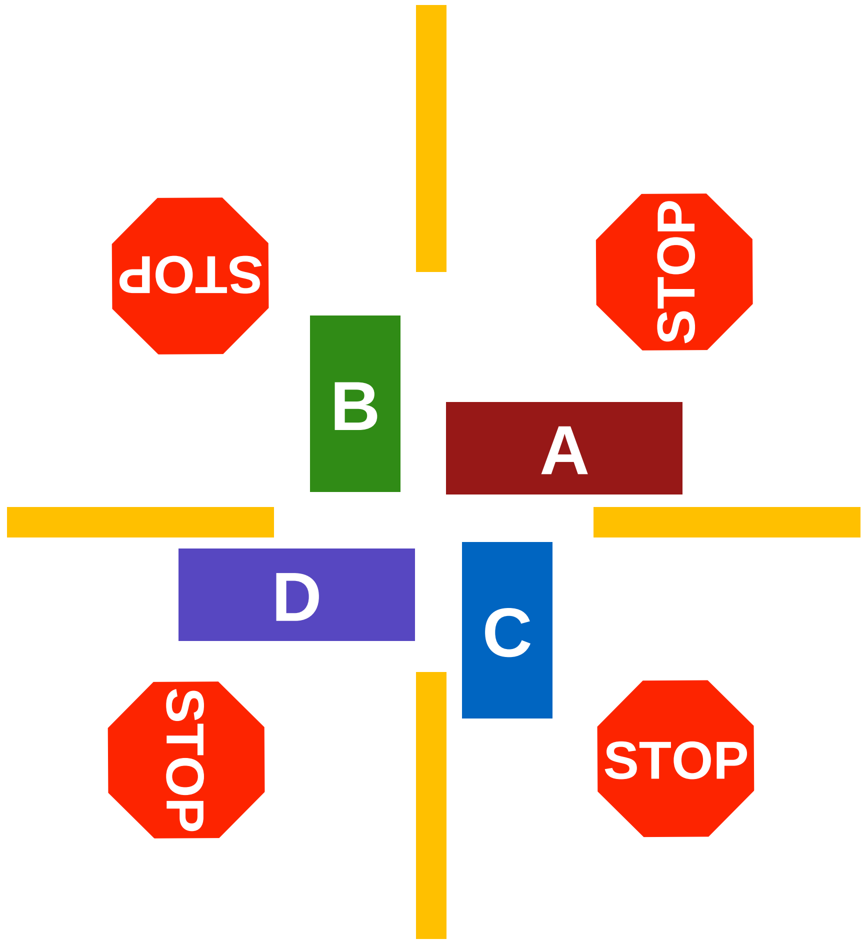
Any other problems?

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

Def:

Threads claim exclusive control of resources that t hey 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

Def:

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 h ave 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

Def:

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

Def:

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

Eliminating Circular Wait

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

 * 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
 */
```

Linux lockdep Module

Idea:
- track order in which locks are acquired
- give warning if circular

Extremely useful for debugging!

Example Output

==
[INFO: possible circular locking dependency detected]
3.1.0rc4test00131g9e79e3e #2

insmod/1357 is trying to acquire lock:
(lockC){+.+...}, at: [<ffffffffa000d438>] pick_test+0x2a2/0x892
[lockdep_test]

but task is already holding lock:
(lockB){+.+...}, at: [<ffffffffa000d42c>] pick_test+0x296/0x892
[lockdep_test]

Source: http://www.linuxplumbersconf.org/2011/ocw/sessions/153

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