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 **Therac-25** was a computer-controlled radiation therapy machine produced by Atomic Energy of Canada Limited (AECL) in 1982
- It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation.
- Because of concurrent programming errors (also known as race conditions), it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.
- Become a standard case study in health informatics, software engineering, and computer ethics.

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

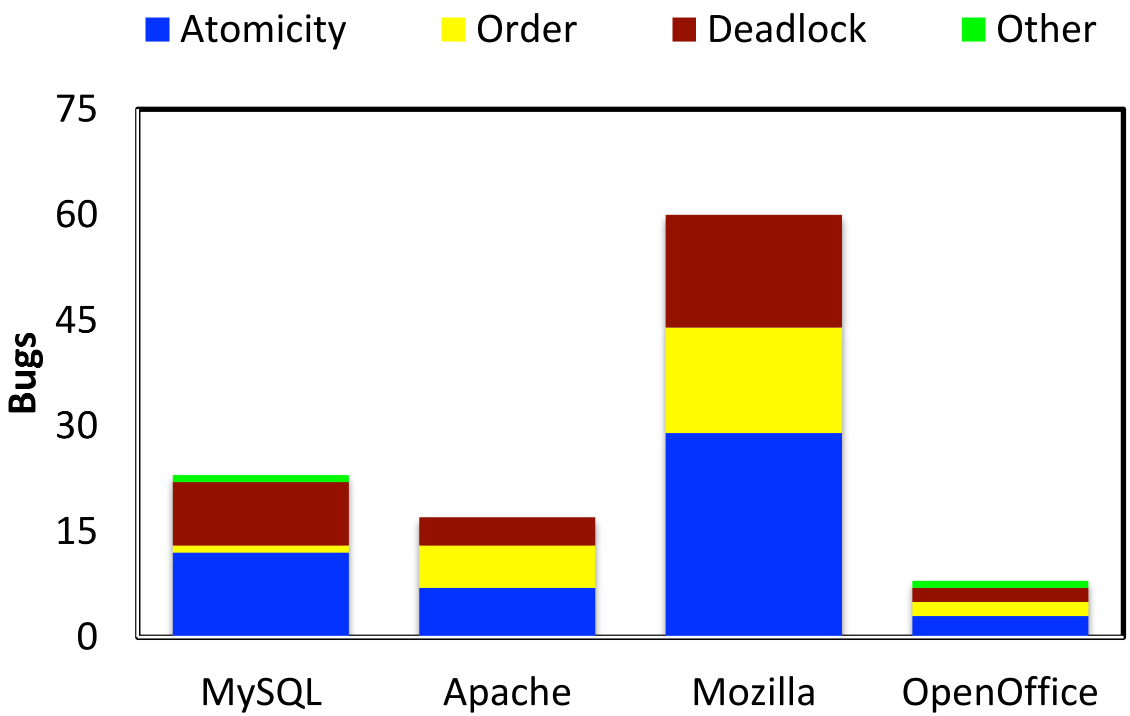
Concurrency in Medicine: Therac-25 (1980's)

"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 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;
```

What’s wrong?

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

Fix Atomicity Bugs with Locks

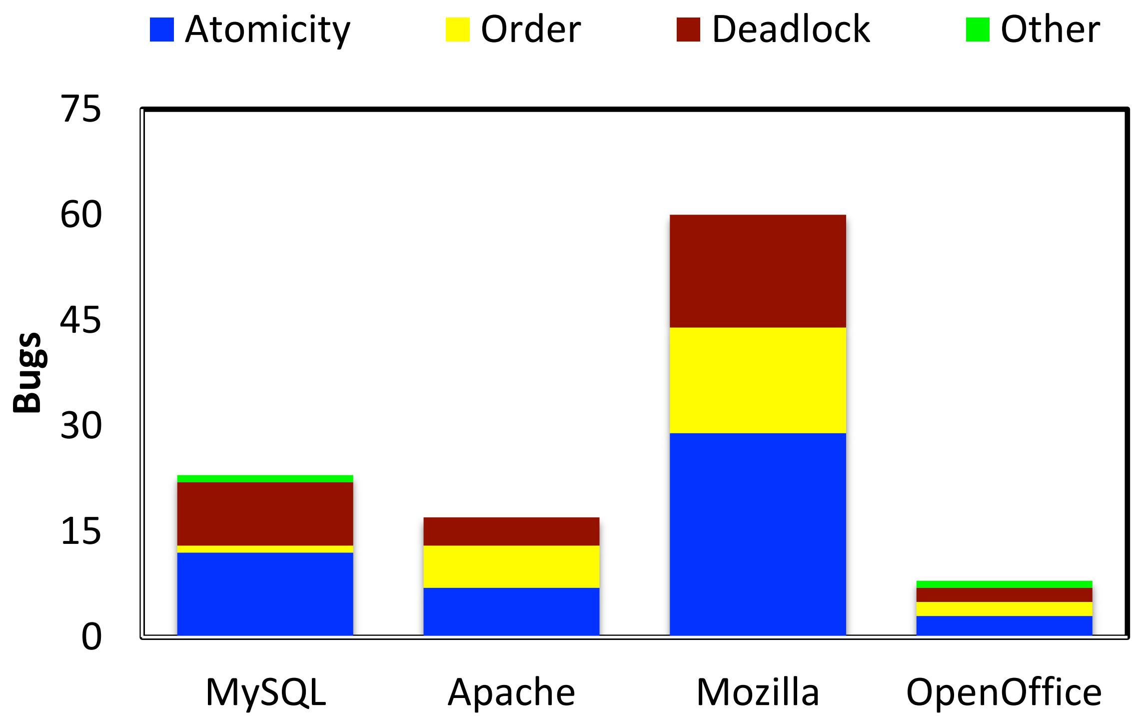
Thread 1:

```
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 concurrency 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, ...);

   Mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   Mutex_unlock(&mtLock);

               ...
}
```

Thread 2:

```
void mMain(...) {
  ...

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

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

One Worry: Races

- A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

race.c

One Worry: Races

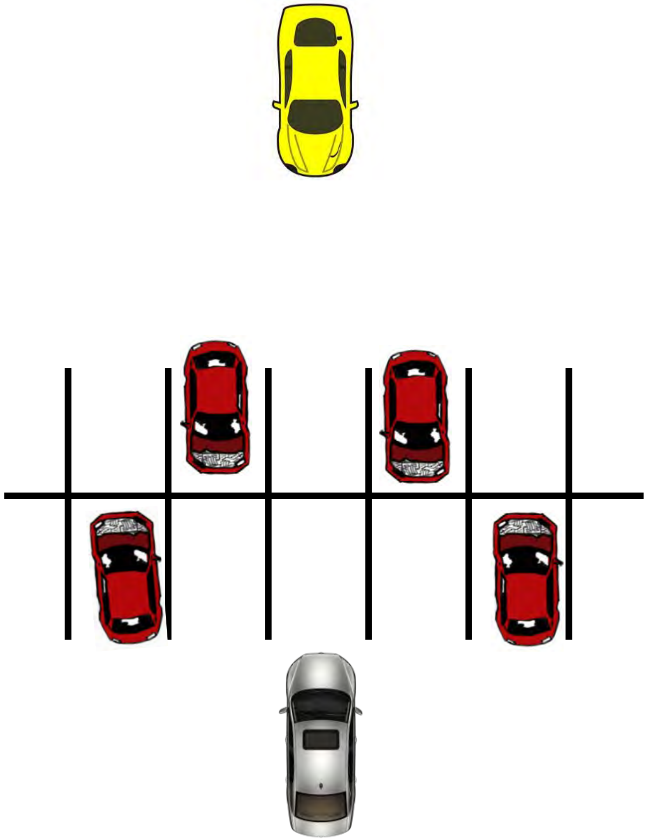
- A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

race.c

Data Race



Race Elimination

- Make sure don't have unintended sharing of state

```
/* a threaded program without the race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = Malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

norace.c

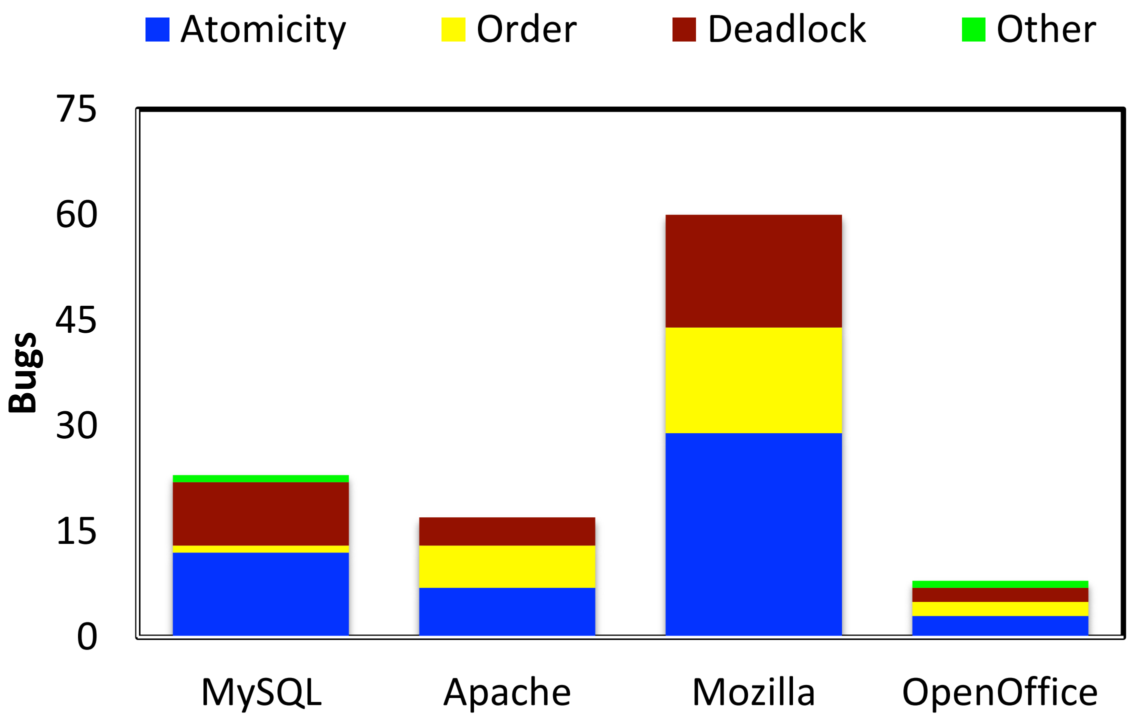
Race Elimination

- Is this correct?

```
/* a threaded program without the race */
int main(int argc, char** argv) {
    pthread_t tid[N];
    uint64_t i;
    for (i = 0; i < N; i++) {
        Pthread_create(&tid[i], NULL, thread, (void *)i);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    uint64_t myid = (uint64_t)vargp;
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

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

Deadlock

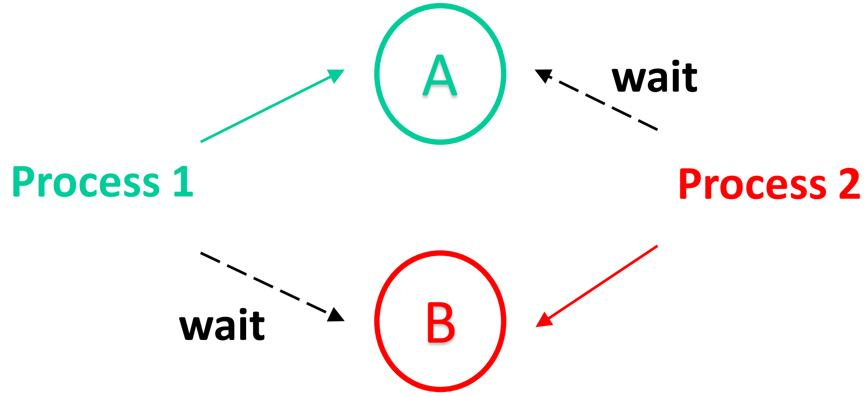
Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

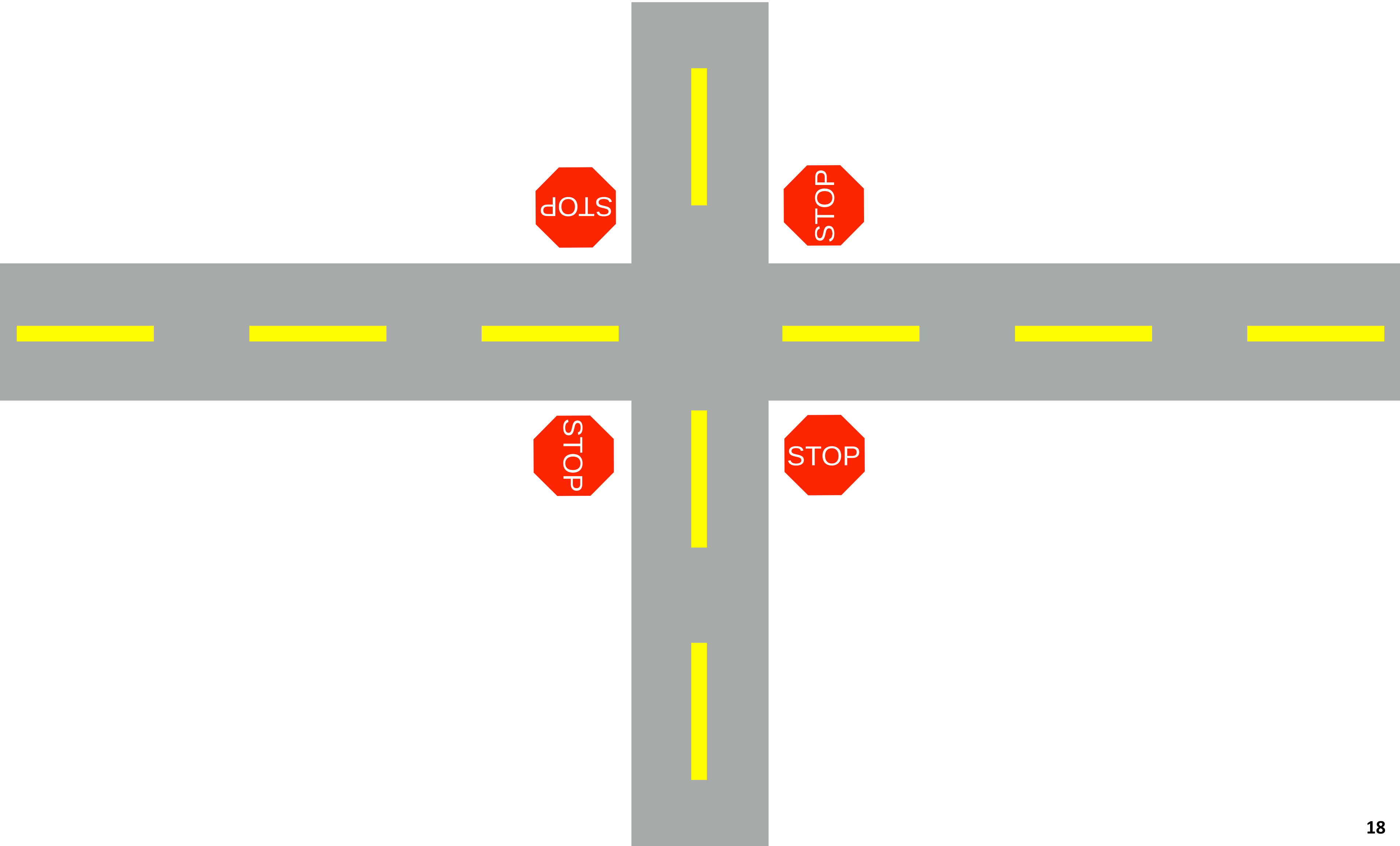
“Cooler" name: the deadly embrace (Dijkstra)

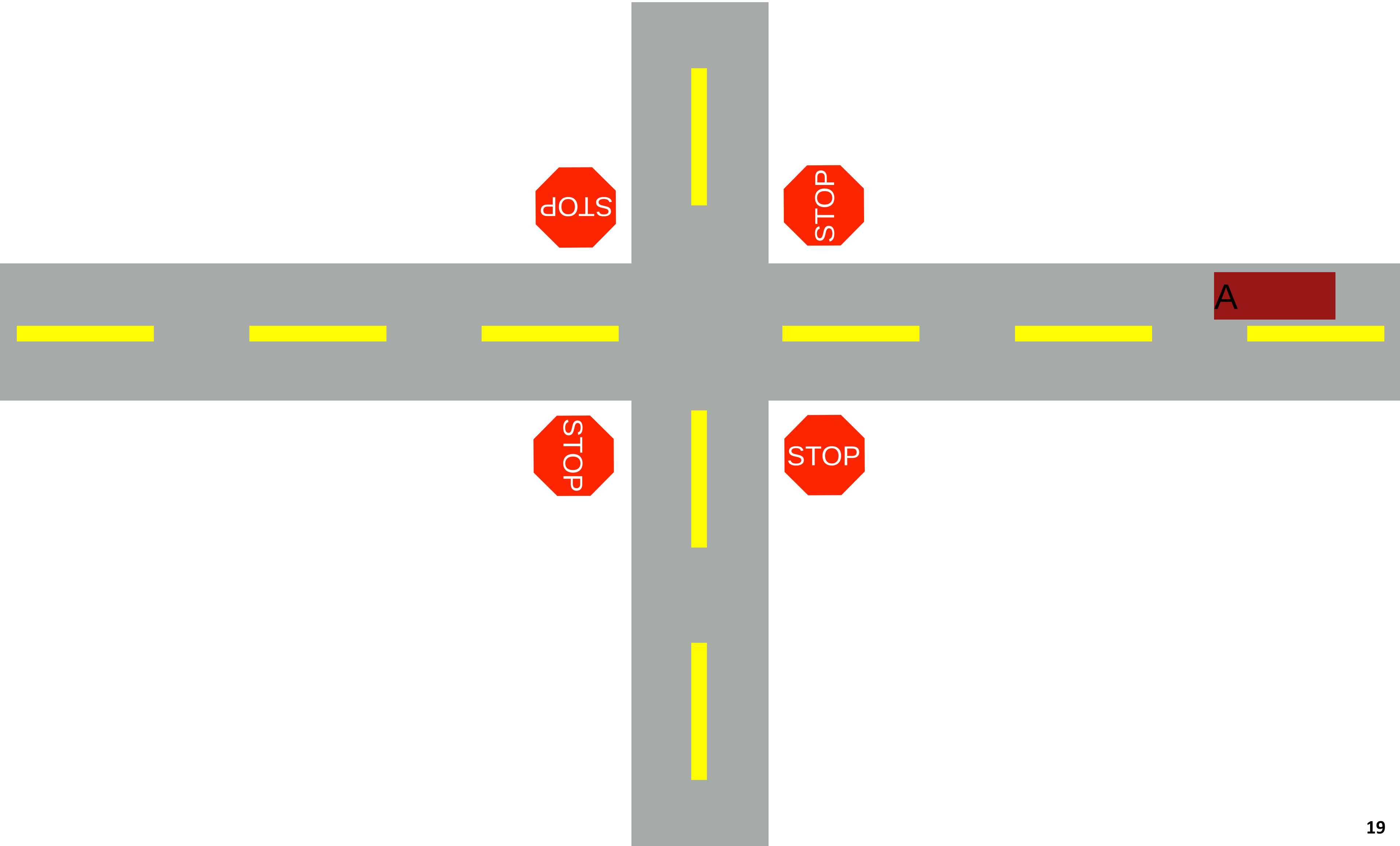
Deadlock

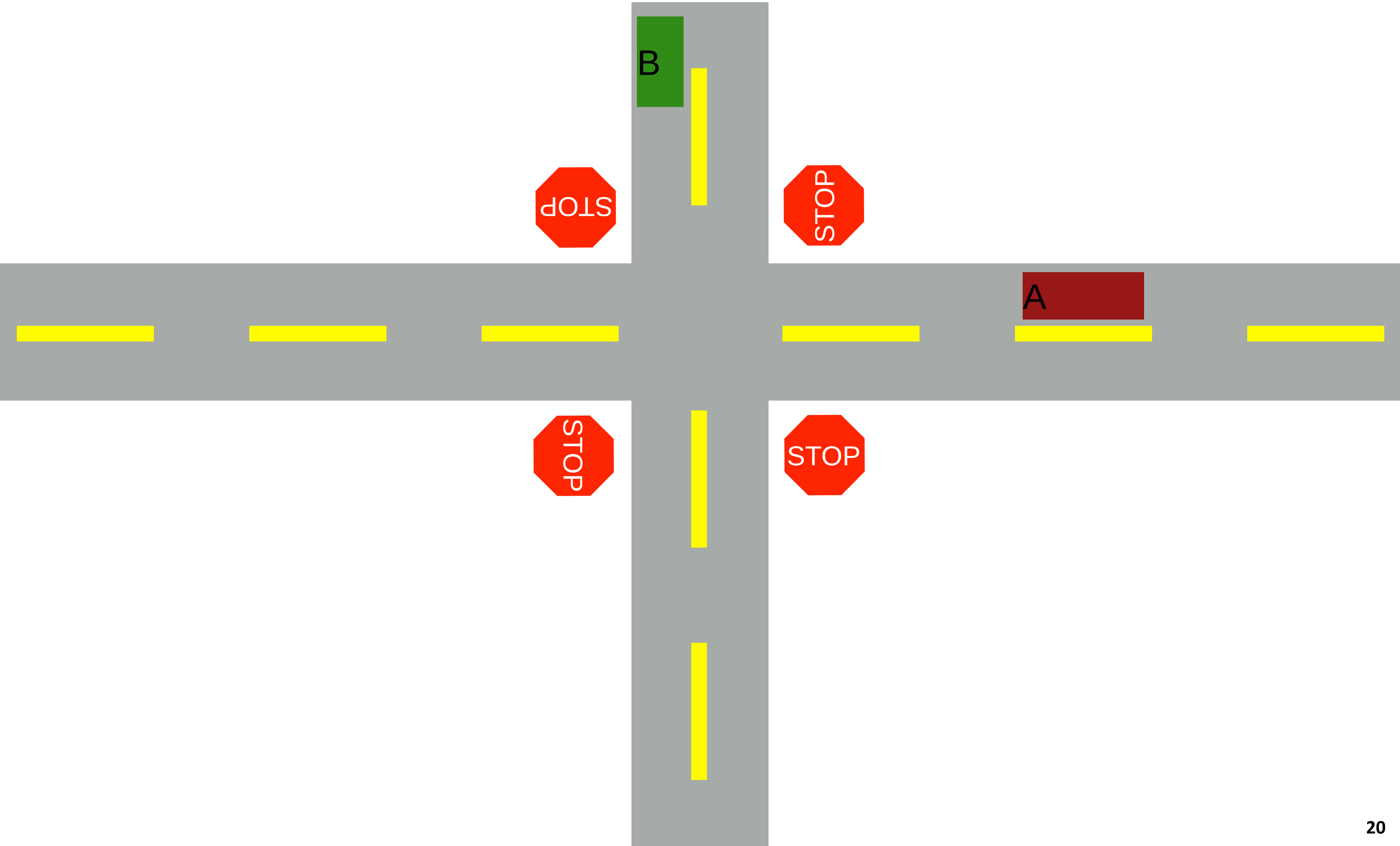
- **Another Def: A process is *deadlocked* if it is waiting for a condition that will never be true.**

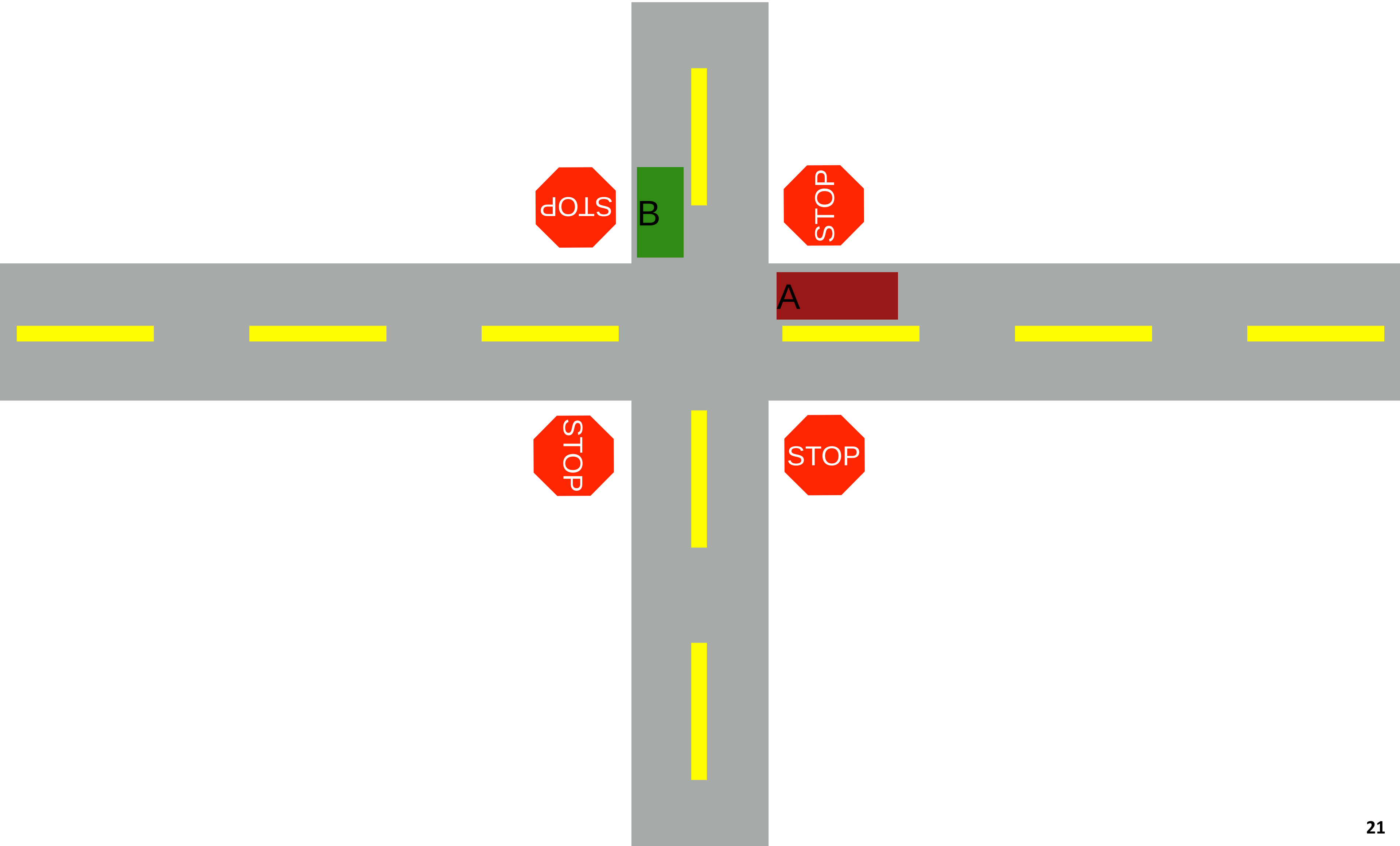
- **Typical Scenario**
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!



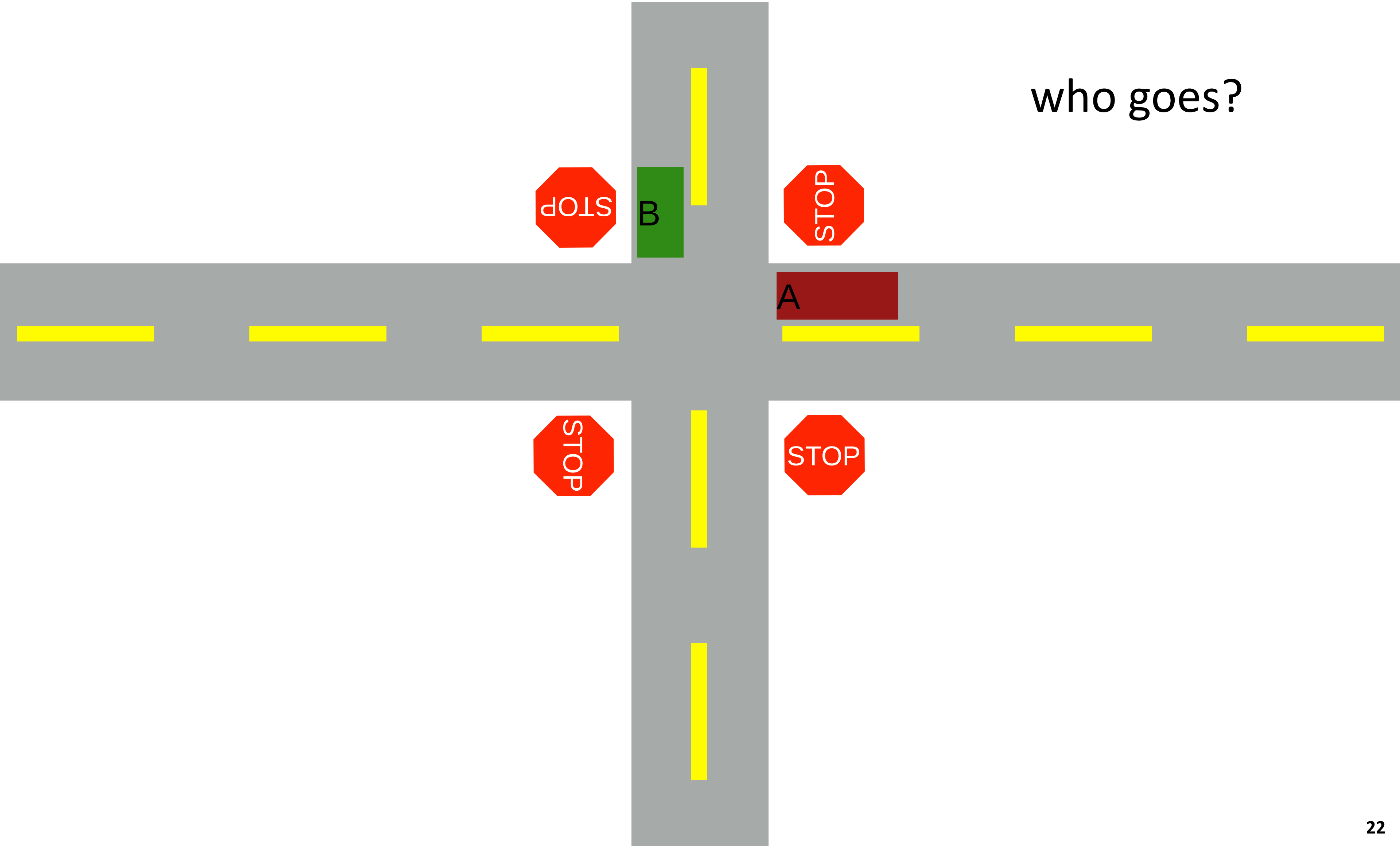

STOP
STOP
STOP
STOP

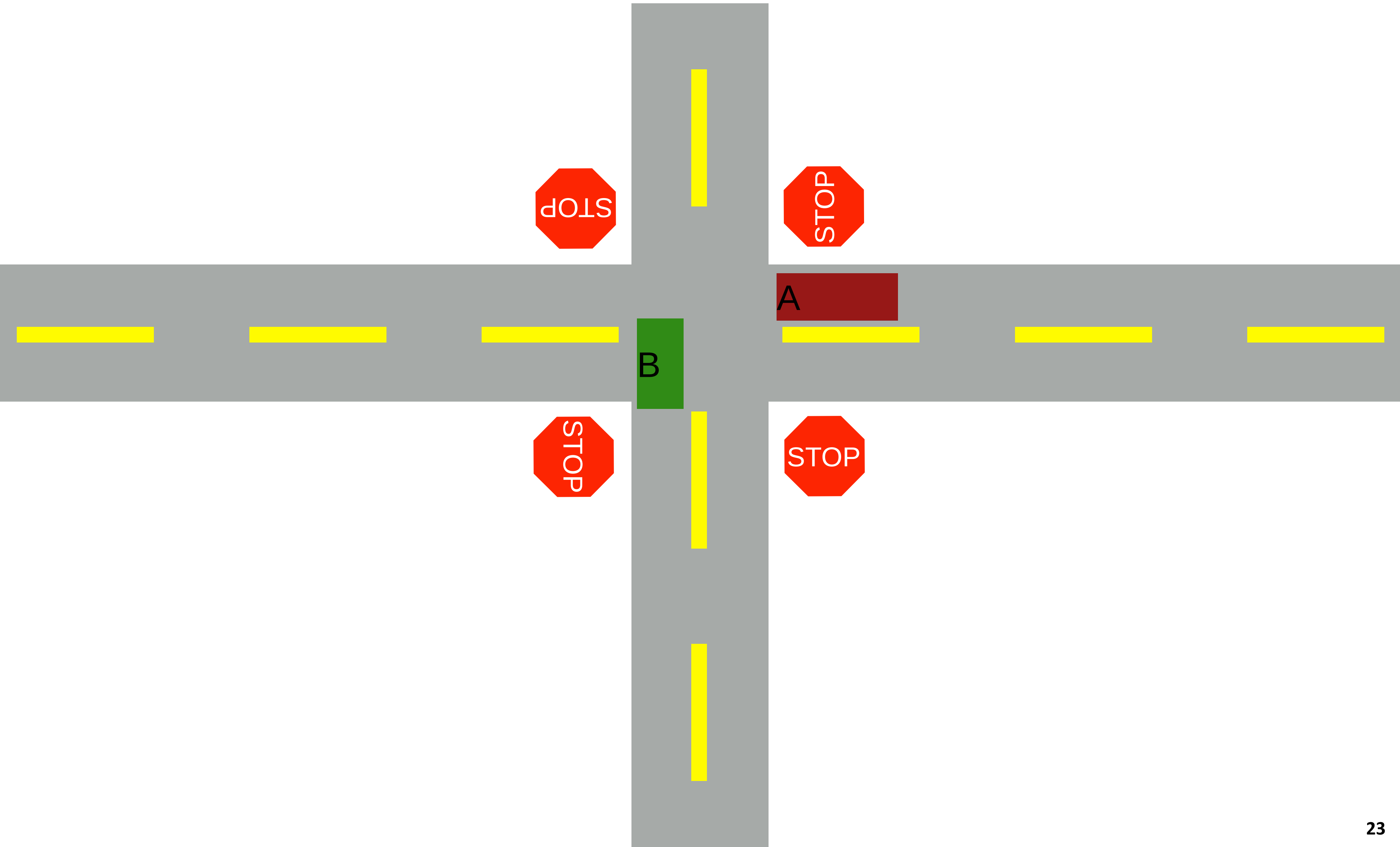

STOP
STOP
STOP
STOP
A

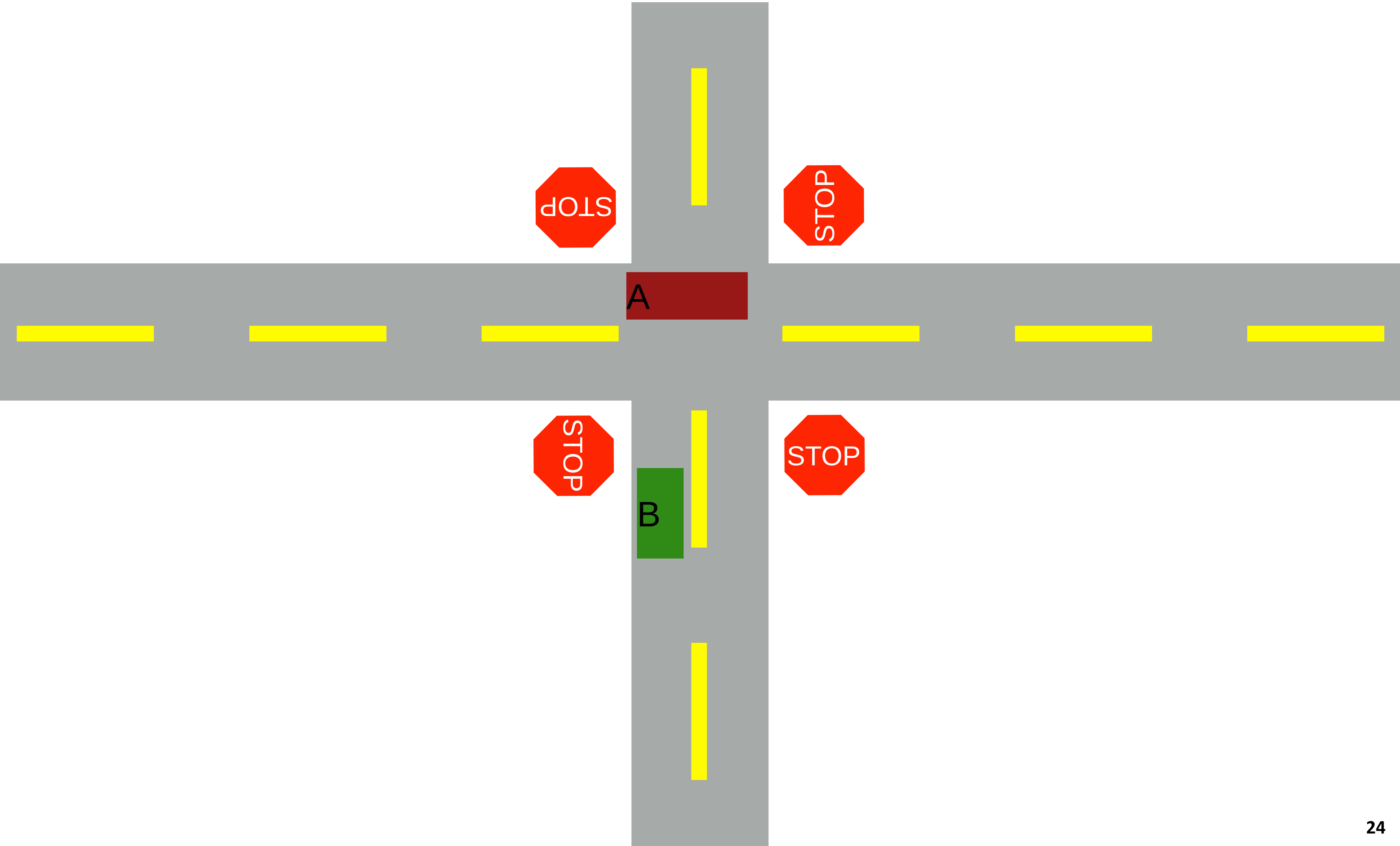

B
STOP
STOP
A
STOP
STOP

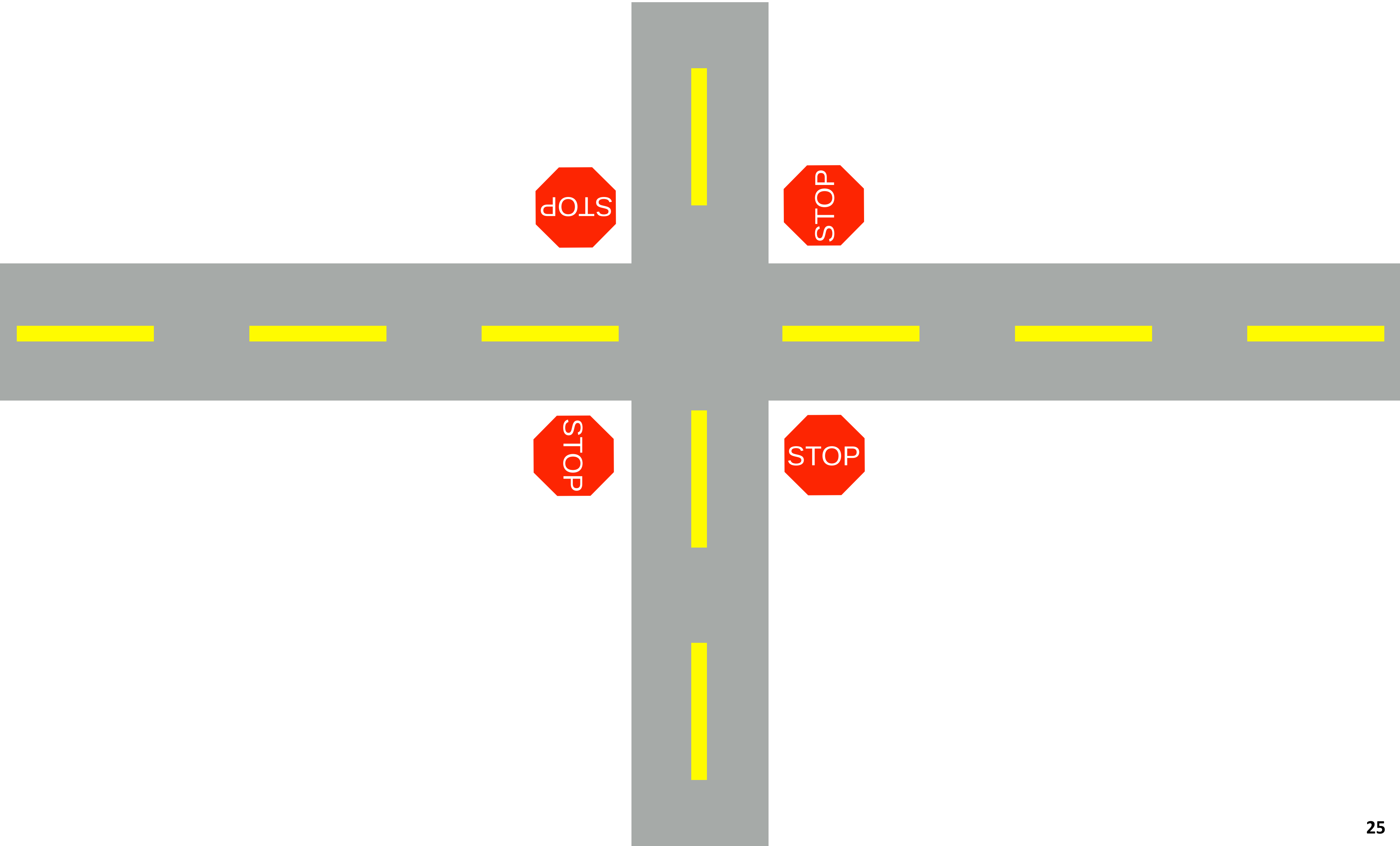

STOP
B
STOP
A
STOP
STOP

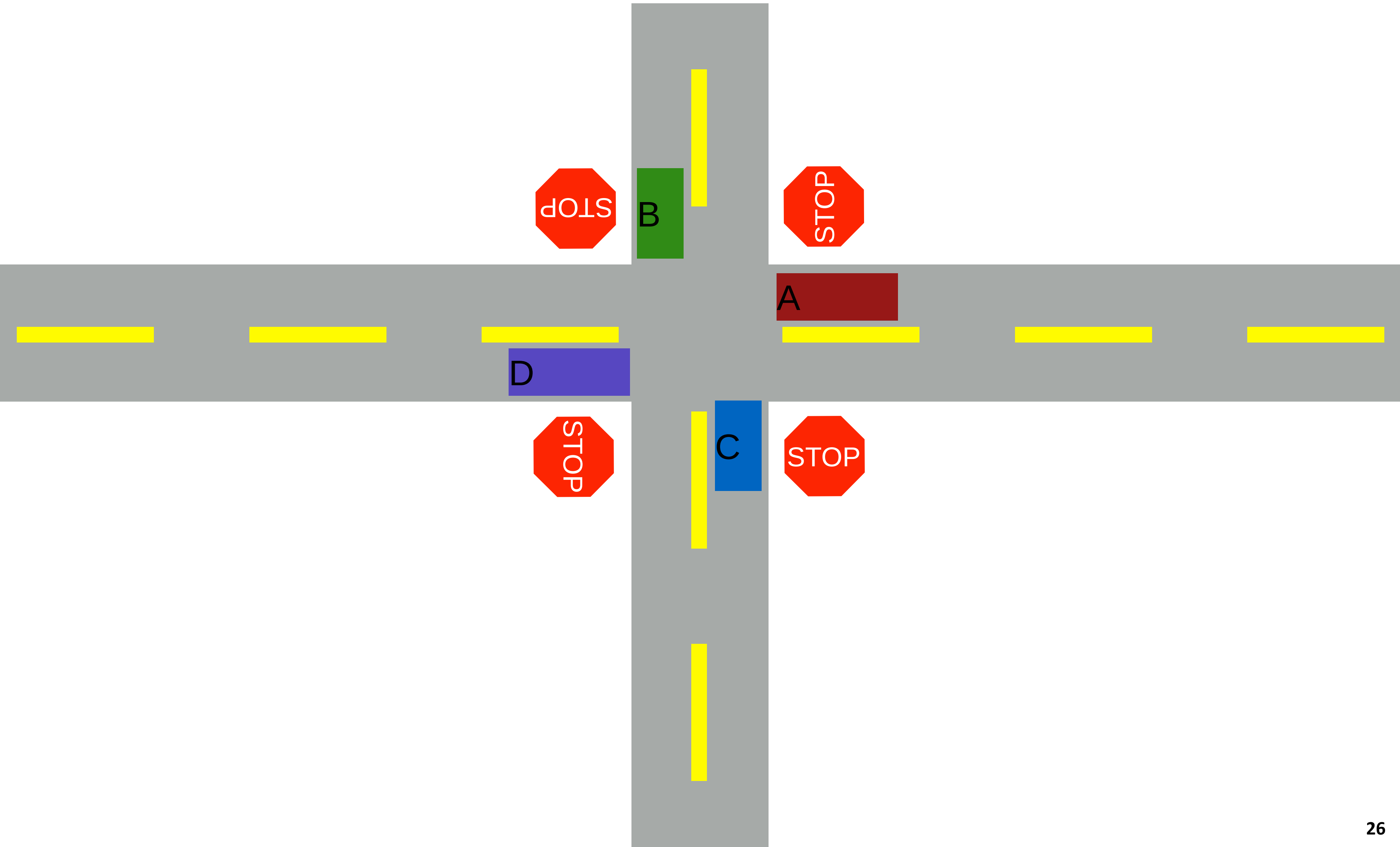
who goes?



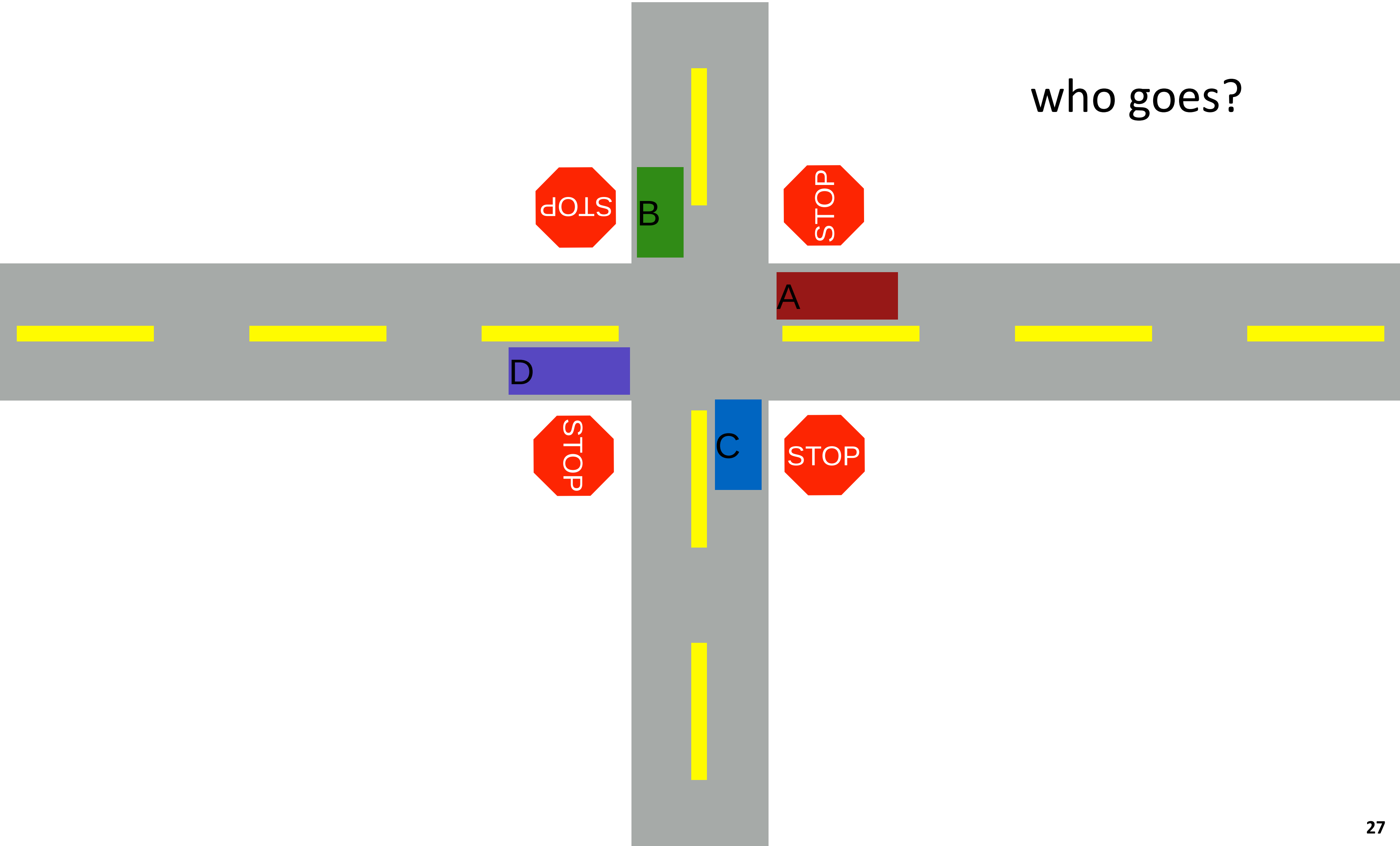

STOP
STOP
A
B
STOP
STOP


STOP
STOP
A
STOP
B
STOP

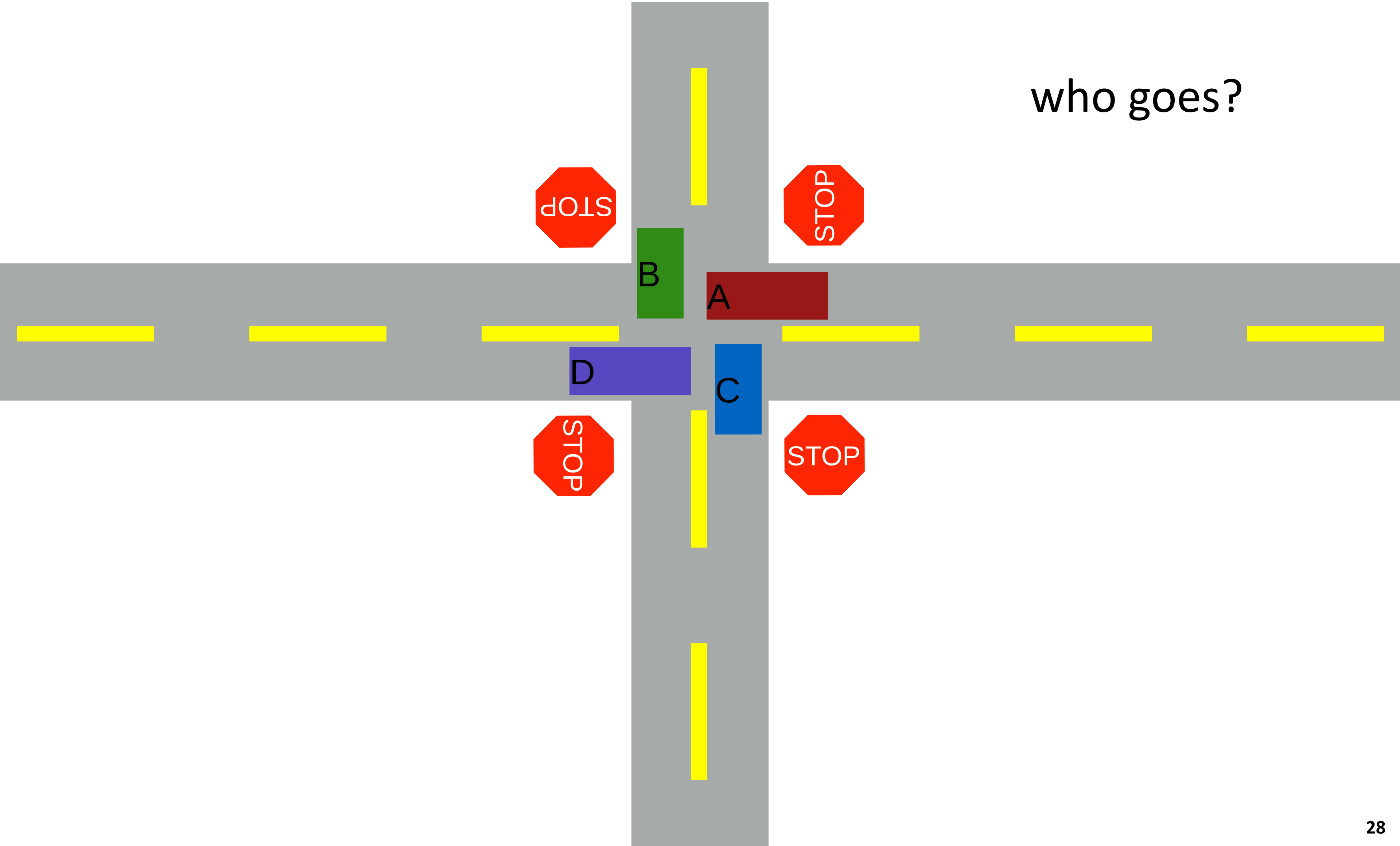

STOP
STOP
STOP
STOP

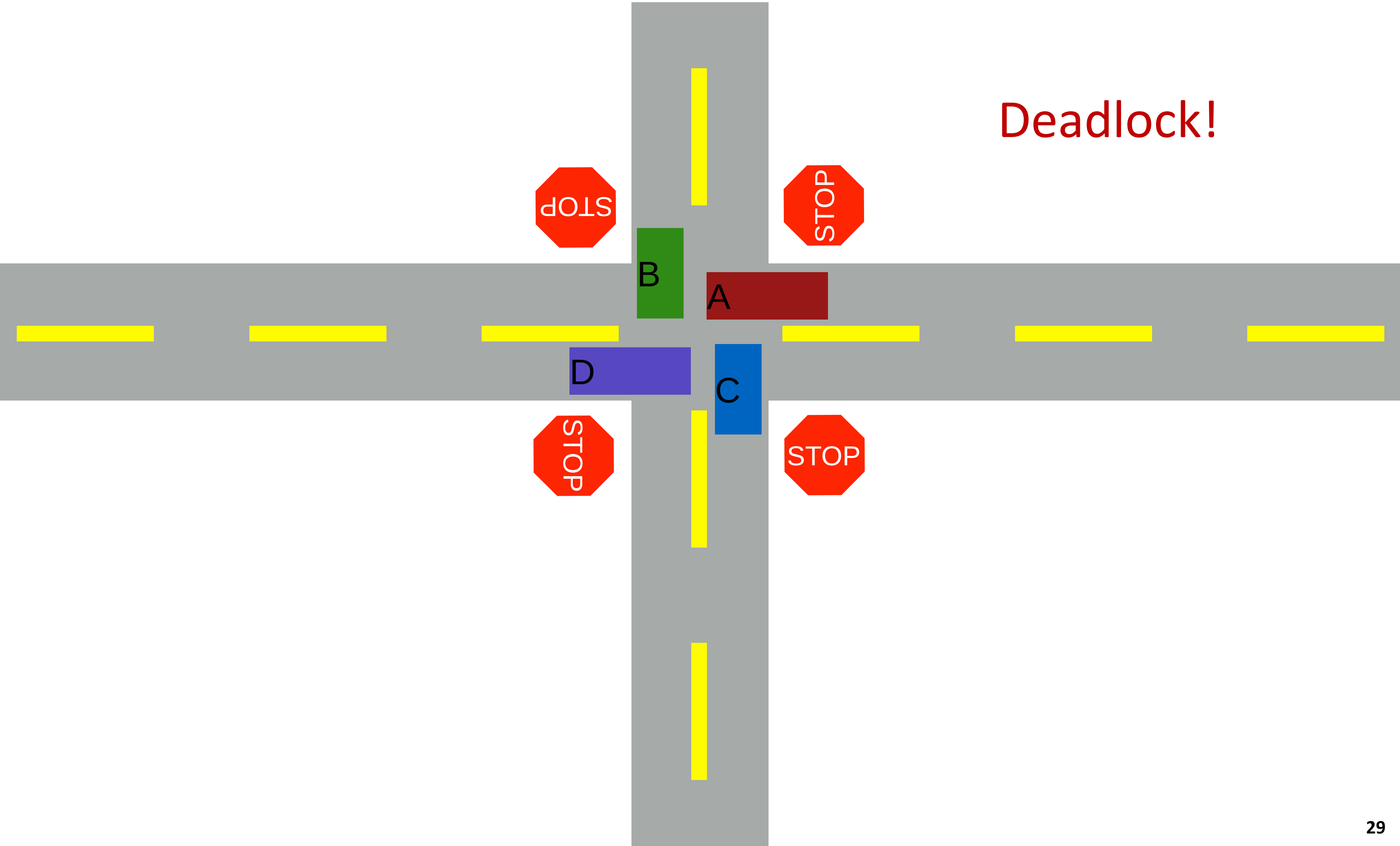

STOP
B
STOP
A
D
C
STOP
STOP

who goes?



who goes?





Deadlock!

Code Example

Thread 1:

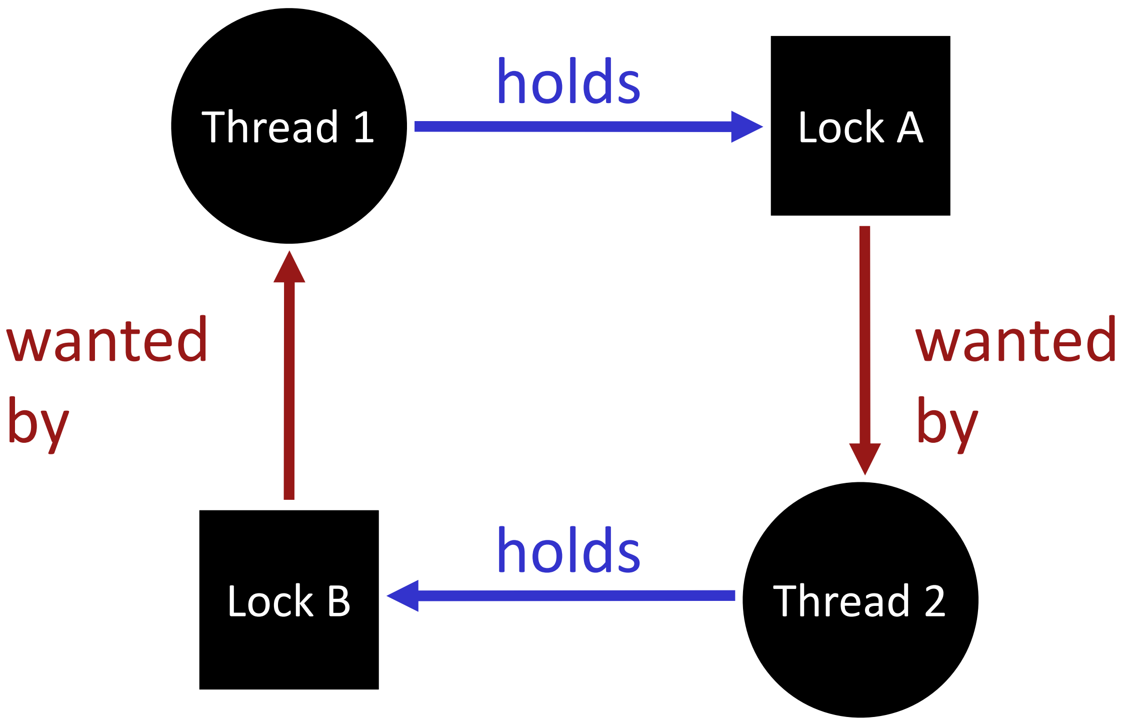
```
lock(&A);
lock(&B);
```

Thread 2:

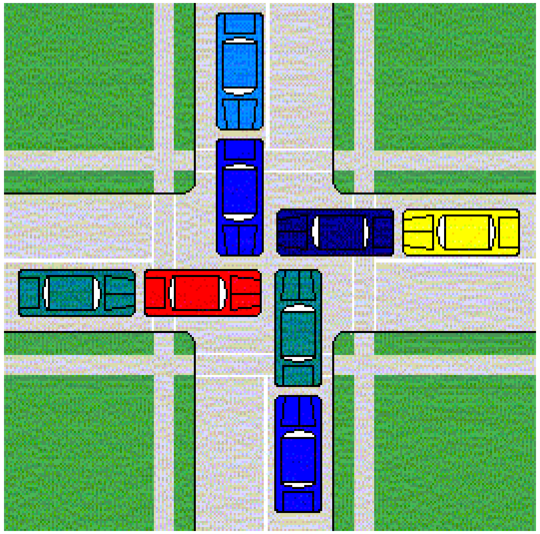
```
lock(&B);
lock(&A);
```

Can deadlock happen with these two threads?

Circular Dependency



Deadlock



Fix Deadlocked Code

Thread 1:

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

Thread 2:

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

How would you fix this code?

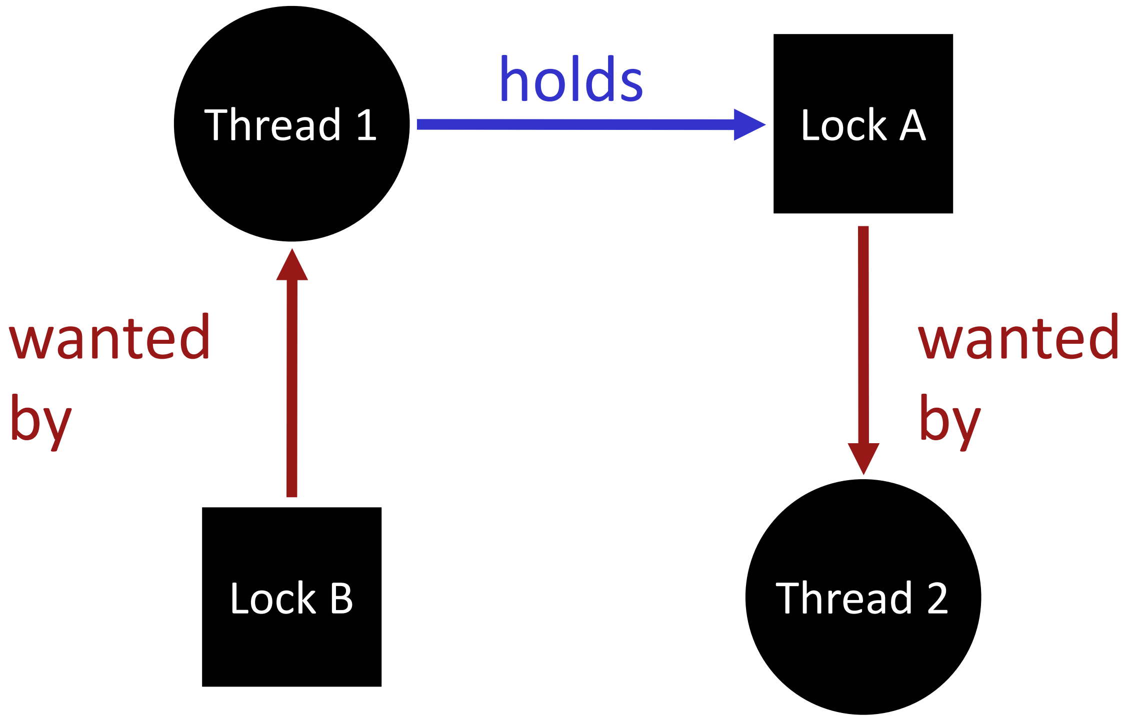
Thread 1

```
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?

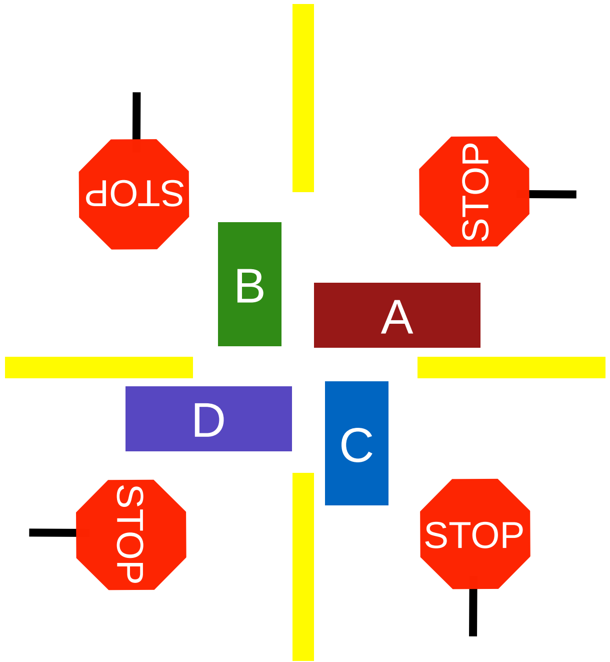
```
if (m1 > m2) { // Compare some kind of encoding
        // grab locks in high-to-low address order
        pthread_mutex_lock(m1);
        pthread_mutex_lock(m2);
} else {
        pthread_mutex_lock(m2);
        pthread_mutex_lock(m1);
}
```

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 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!**
- **Try to replace locks with atomic primitive:**
- **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 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

- **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

  How to address?

Support Preemption

- **Classic solution: Exponential back-off -- wait some time**

```
        i = 1;
top:
        lock(A);
        if (trylock(B) == -1) {
                unlock(A);
                sleep(i);
                i *= 2;
                goto top;
        }
        …
```

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 requested 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**

Releasing Lock in Other Order

thread 0

```
A.lock();
B.lock();

B.unlock();
A.unlock();
```

thread 1

```
A.lock();
B.lock();

A.unlock();
B.unlock();
```

Releasing Lock in Other Order

thread 0

```
A.lock();
B.lock();
Foo();
B.unlock();
Bar();
A.unlock();
```

thread 1

```
A.lock();
B.lock();
Foo();
A.unlock();
Bar();
B.unlock();
```

Any problem?

Releasing Lock in Other Order

thread 0

```
A.lock();
B.lock();
Foo();
B.unlock();
Bar();
A.unlock();
```

thread 1

```
A.lock();
B.lock();
Foo();
A.unlock();
Bar();
B.unlock();
```

If **Bar()** attempts to reacquire A, you've effectively broken your lock ordering. You're holding B and then trying to get A. Now it **can** deadlock.

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
 */
```

Banker's Algorithm

- **Banker's algorithm** is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.
- When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

Banker’s Algorithm

- For the Banker's algorithm to work, it needs to know three things:
  - How much of each resource each process could possibly request ("MAX")
  - How much of each resource each process is currently holding ("ALLOCATED")
  - How much of each resource the system currently has available ("AVAILABLE")
- Resources (memory, semaphores, interface) may be allocated to a process only if the amount of resources requested is less than or equal to the amount available; otherwise, the process waits until resources are available.

Banker’s Algorithm

- The Banker's algorithm derives its name from the fact that this algorithm could be used in a banking system to ensure that the bank does not run out of resources, because the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers.
- By using the Banker's algorithm, the bank ensures that when customers request money the bank never leaves a safe state. If the customer's request does not cause the bank to leave a safe state, the cash will be allocated, otherwise the customer must wait until some other customer deposits enough.

Banker's Algorithm

Total system resources:

A	B	C	D
6	5	7	6

Available system res:

A	B	C	D
3	1	1	2

Needed resources:
max - allocated

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Maximum allocated resources for processes:

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Banker's Algorithm – Safe State

Available system res:

A	B	C	D
3	1	1	2

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Needed resources:
(max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Assumption: the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward.

Safe state: a state is considered safe if it is possible for all processes to finish executing (terminate).

Safe state?

Banker's Algorithm – P1-P2-P3

Available system res:

A	B	C	D
3	1	1	2

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Needed resources:
(max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Available system res after P1:

A	B	C	D
4	3	3	3

After P1 resources for processes:

	A	B	C	D
P1				
P2	1	0	3	3
P3	1	2	1	0

Safe!

Banker's Algorithm – P1-P2-P3

Available system res after P1:

A	B	C	D
4	3	3	3

After P1

resources for processes:

	A	B	C	D
P1				
P2	1	0	3	3
P3	1	2	1	0

Available system res after P2:

A	B	C	D
5	3	6	6

After P2

resources for processes:

	A	B	C	D
P1				
P2				
P3	1	2	1	0

Needed resources:

(max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Safe!

Banker's Algorithm – Unsafe State

Available system res:

A	B	C	D
3	1	1	2

Current allocated resources for processes:

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0

Needed resources: (max – allocated)

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

Assumption: the system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward.

Safe state: a state is considered safe if it is possible for all processes to finish executing (terminate).

Unsafe state?

Summary

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