CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec. 14: Concurrency Bugs, Deadlocks

Adapted from content originally created by: Prof. Yeongjin Jang

Administrivia

- Lab 3 due date: **May 20, 2022** [**Friday**] at **11:59 PM**!

- Quiz 3 on **May 24, 2022** [**Tuesday**] at **8:30 AM**!
    - Available until **May 25, 2022** [**Wednesday**], **11:59 PM**

- Watch all **Tutorials** and go through the slides/textbook

Quiz 3 Coverage

- JOS Lab 3 (User/Kernel, System Call and Interrupt Handling)
- JOS Lab 4 (Preemptive Multitasking & Copy-on-Write Fork)
- Lecture 11: Multithreading and Synchronization
- Lecture 12: Locks
- Lecture 13: Locks 2
- Lecture 14: Concurrency Bugs and Deadlocks
- Sample Quiz
  - https://sibin.github.io/teaching/cs444-osu-operating-systems/spring_2022/l/quiz_3.sample.pdf
  - https://sibin.github.io/teaching/cs444-osu-operating-systems/spring_2022/l/quiz_3.sample.answer.pdf

Recap: lock-example

- Repo: https://gitlab.unexploitable.systems/root/lock-example
- 5 Lock implementations
  - Naïve lock [**bad_lock, inconsistent**]
  - xchg lock [test-and-set, **slow**]
  - cmpxchg lock [**a fake test and test-and-set, still slow**]
  - Software test and hardware test-and-set [fast!]
  - Hardware test-and-set with exponential backoff [**faster!**]
- Performance checks
  - Total execution time
  - L1-dcache-load-misses
  - Compare with pthread_mutex

Lock	**Cache Misses [approx.]**	**Time [ms]**
xchg	17 million	944
cmpxchng	19 million	1124
tts	14 million	500
backoff	230 thousand	197
pthread_mutex	1.6 million	458

pthread_mutex | implementation

```
if (LLL_MUTEX_TRYLOCK (mutex) != 0)
  {
    int cnt = 0;
    int max_cnt = MIN (max_adaptive_count (),
                       mutex->__data.__spins * 2 + 10);
    do
      {
        if (cnt++ >= max_cnt)
          {
            LLL_MUTEX_LOCK (mutex);
            break;
          }
        atomic_spin_nop ();
      }
    while (LLL_MUTEX_TRYLOCK (mutex) != 0);
```

test [is lock variable **not** ‘0’?]

exponential backoff setup

Spins * 2 + 10
Default count is 100

test-and-set [use `xchg` for locking]

```
#define atomic_spin_nop() __asm ("pause")
```

exponential backoff [wait until we reach max count]

Can **acquire** lock if lock variable ‘0’

Locks are **Slow!**

```
void
xchg_lock(volatile uint32_t *lock) {
    while(xchg(lock, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

Run **while()** loops internally

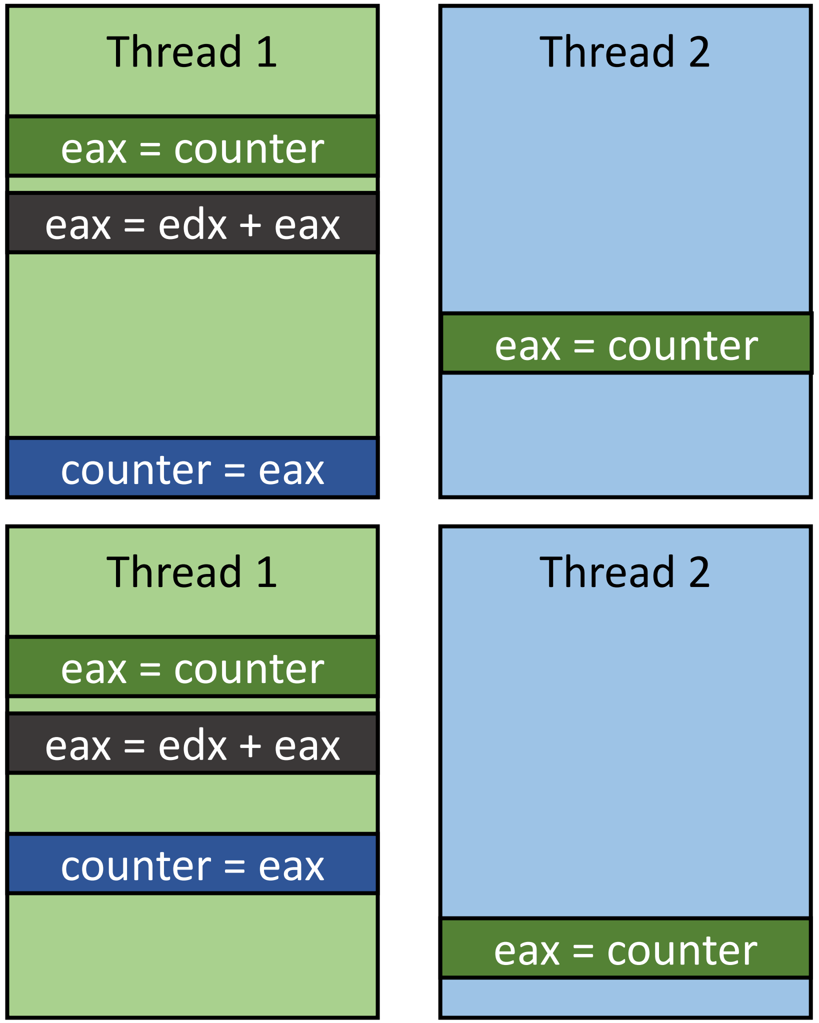
Can block other threads

Need to **carefully** determine **when** and **where** to use locks

When Do We Need to Use a Lock?

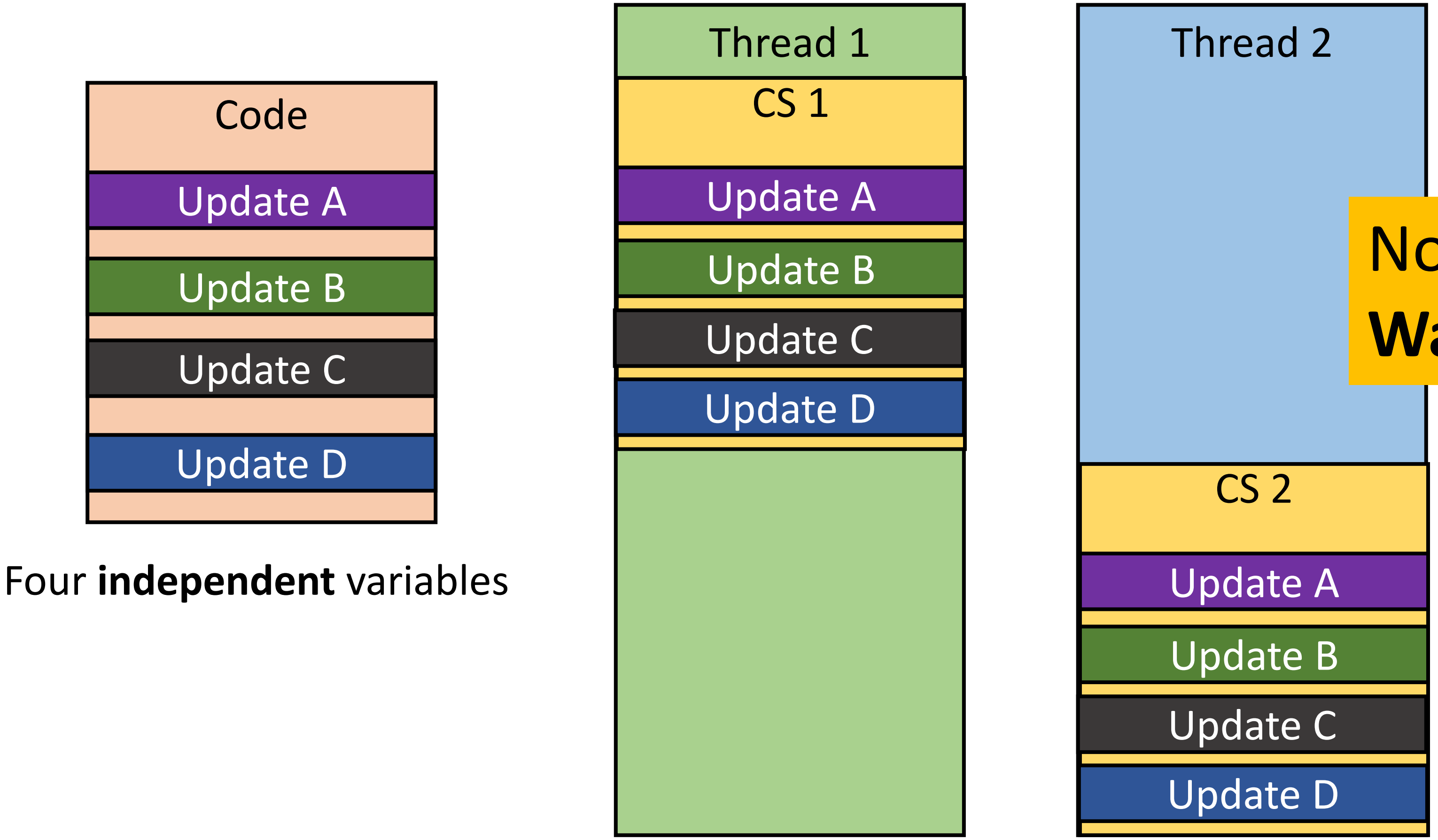
- Write must finish **before** the next load
- **Many** reads/writes
  - Especially **many writers**!

- **One writer** and many readers
  - Not always if there is only one writer
  - If **write-read order is not important**
  - Having no lock is fine



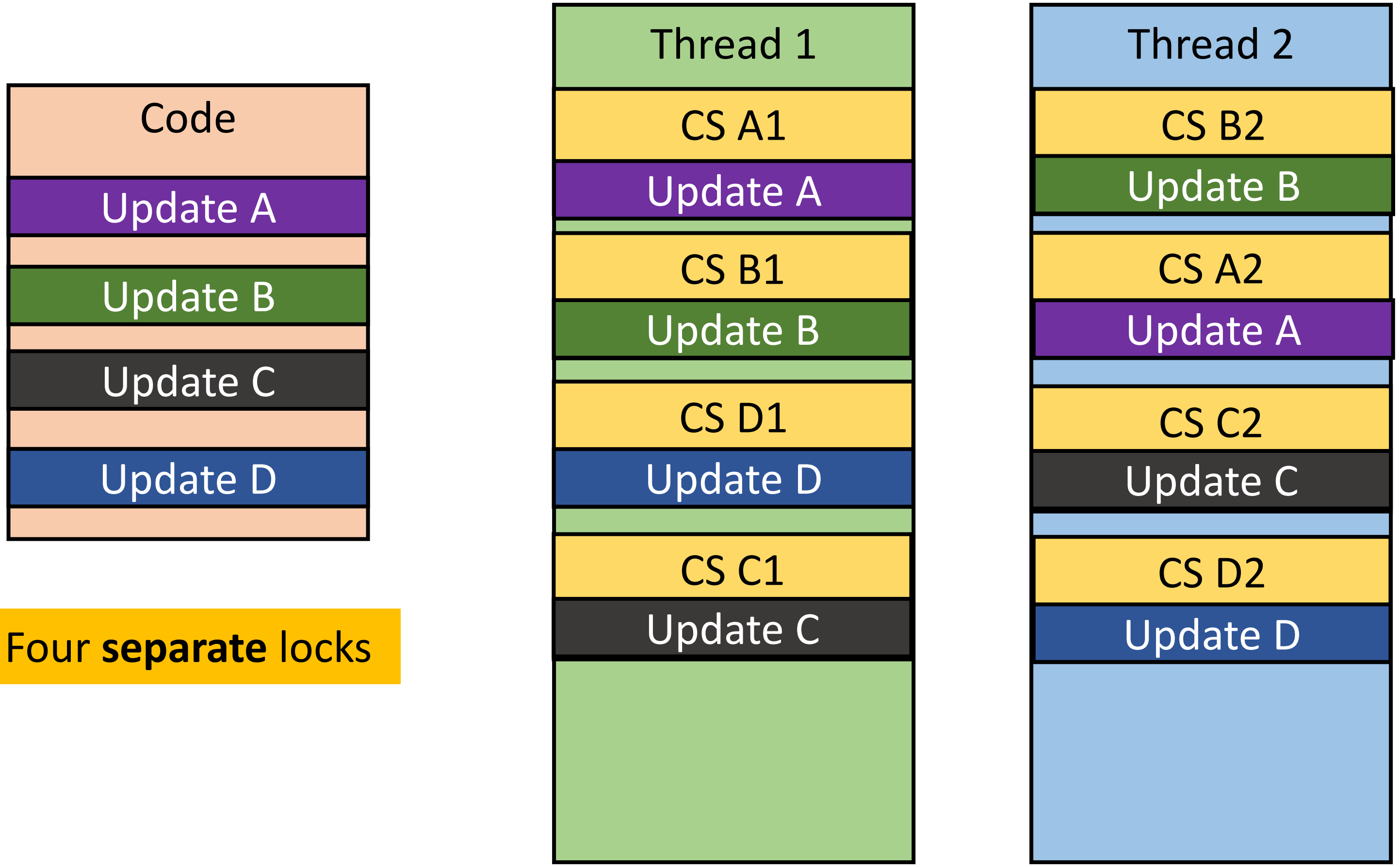
Where Do We Need to Put a Lock?

- What if a **critical section is too big**?



No real data race!
Wasted Time!

Small Critical Sections



Four **separate** locks

Fast! Developer should be careful about splitting critical sections

General Practice

- Use lock **only if required**
  - Determine cases when you do not need a lock
    - Atomic read
    - Only one writer
- Use **small critical sections**
  - Critical section prohibits concurrent execution
  - Determine where do we share a variable
  - Wrap only the code that updates the shared variable
- **Looks simple but often gets really complex!**

Concurrency **Bugs**

- Atomicity
- Ordering
- Deadlocks

Concurrency Bugs 1

reader

writer

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

Concurrency Bugs 1

```
Thread 1::
if (thd->proc_info) {
   ...
   fputs(thd->proc_info, ...);
   ...
}

Thread 2::
thd->proc_info = NULL;
```

time of **check** (line `if (thd->proc_info) {`)

time of **use** (line `fputs(thd->proc_info, ...);`)

Concurrency Bugs 1

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

time of **check**

time of **use**

write!

“time of check to time of use” [TOCTTOU] bug!

Concurrency Bugs 1

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

`thd->proc_info` is **not** `NULL`

`thd->proc_info` is `NULL`!

set `thd->proc_info` to `NULL`!

TOCTTOU Solution? **Lock!**

```
Thread 1::
if (thd->proc_info) {
    ...
    fputs (thd->proc_info, ...);
    ...
}

Thread 2::
thd->proc_info = NULL;
```

TOCTTOU Solution? **Lock!**

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
```

```
Thread 1::
if (thd->proc_info) {
      ...
      fputs (thd->proc_info, ...);
      ...
}

Thread 2::
thd->proc_info = NULL;
```

TOCTTOU Solution? **Lock!**

```
pthread_mutex_t jeff = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
if (thd->proc_info) {
      ...
      fputs (thd->proc_info, ...);
      ...
}

Thread 2::
thd->proc_info = NULL;
```

TOCTTOU Solution? **Lock!**

```
pthread_mutex_t jeff = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock (&jeff) ;
if (thd->proc_info) {
      ...
      fputs (thd->proc_info, ...);
      ...
}
pthread_mutex_unlock (&jeff);

Thread 2::
thd->proc_info = NULL;
```

TOCTTOU still exists!

write!

no lock on the write!

TOCTTOU Solution? **Lock!**

```
pthread_mutex_t jeff = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock (&jeff) ;
if (thd->proc_info) {
        ...
        fputs (thd->proc_info, ...);
        ...
}
pthread_mutex_unlock (&jeff);
```

```
Thread 2::
pthread_mutex_lock (&jeff) ;
thd->proc_info = NULL;
pthread_mutex_unlock (&jeff);
```

Mutual Exclusion

TOCTTOU resolved!

Concurrency Bugs

- No bugs in single thread execution
- **Bugs show up in multithreaded execution**
  - Multiple cores, etc.

- Three types of concurrency bugs
  1. Atomicity
  2. Ordering
  3. Deadlock

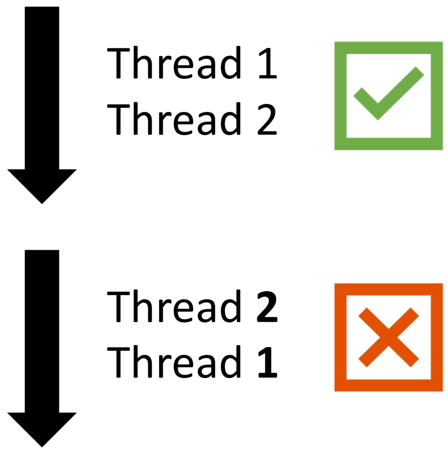

ordering

Ordering Example | Mozilla Code

```
Thread 1::
void init() {
    ...
    mThread = PR_CreateThread(mMain, ...);
    ...
}

Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

not initialized

order of execution:



Solution? Locks and **Conditional Variables**

- Thread scheduling order shouldn't matter
- **Conditional variables**
  - **waits on actions** from **other threads**

```
Thread 1::
void init() {
    ...
   mThread = PR_CreateThread(mMain, ...);
    ...
}

Thread 2::
void mMain(...) {
     ...
     mState = mThread->State;
     ...
}
```

run **first**

wait until `init()` completes

Conditional Wait

- `pthread_cond_signal(&lock_variable);`
- **signal all waiting threads that the condition has been met**

- `pthread_cond_wait(&lock_variable);`
- **wait until signal is received**

Conditional Wait | Usage

```
Thread 1::
void init() {
        ...
        mThread = PR_CreateThread(mMain, ...);
        ...
}

Thread 2::
void mMain(...){
        ...
        mState = mThread->state;
        ...
}
```

Conditional Wait | Usage

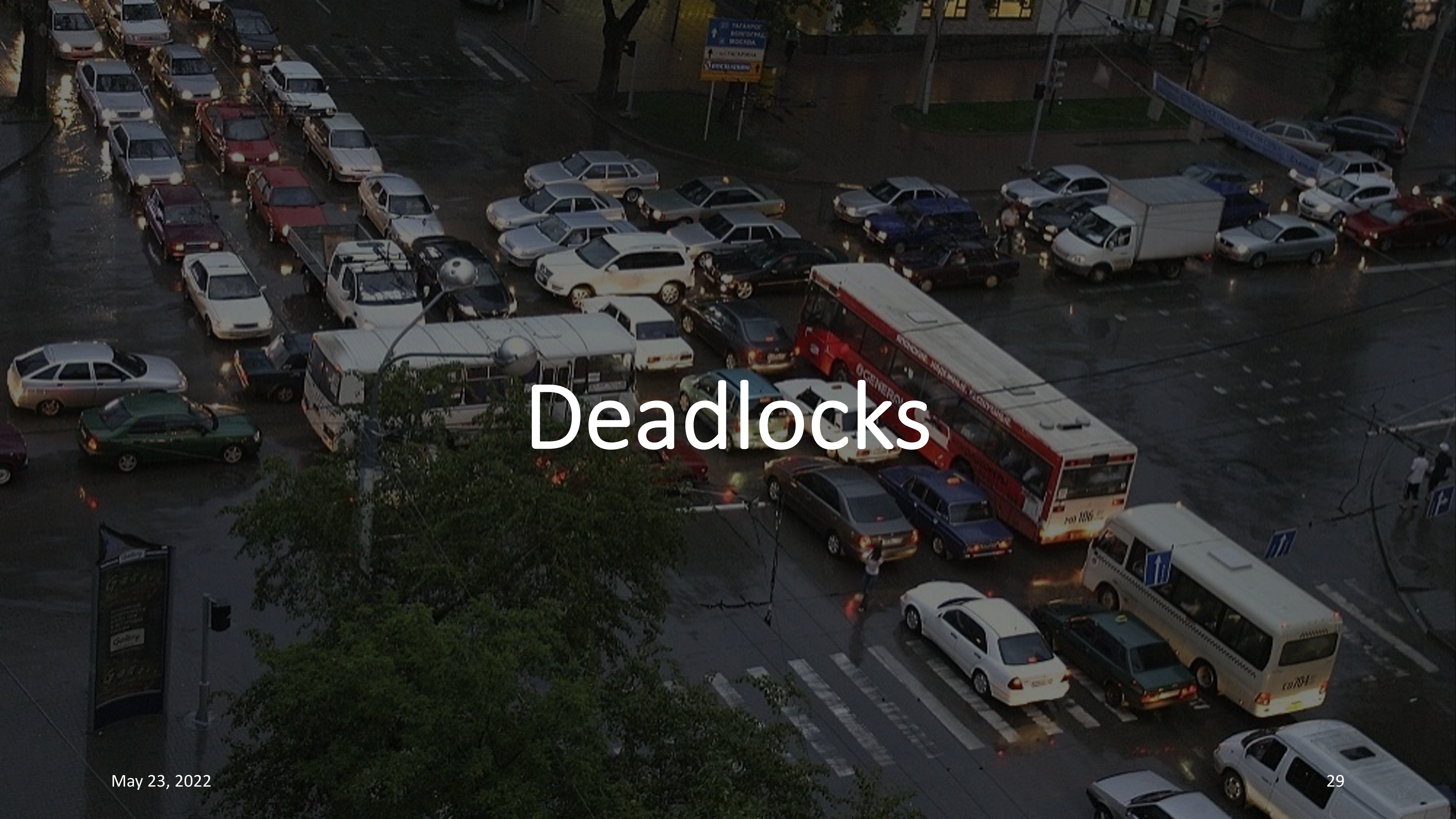
```
Thread 1::
void init() {
        ...
        mThread = PR_CreateThread(mMain, ...);
        //signal that thread has been created
        pthread_cond_signal(&mtCond);
        ...
}

Thread 2::
void mMain(...){
        ...
        pthread_cond_wait(&mtCond);
        mState = mThread->state;
        ...
}
```

incorrect!

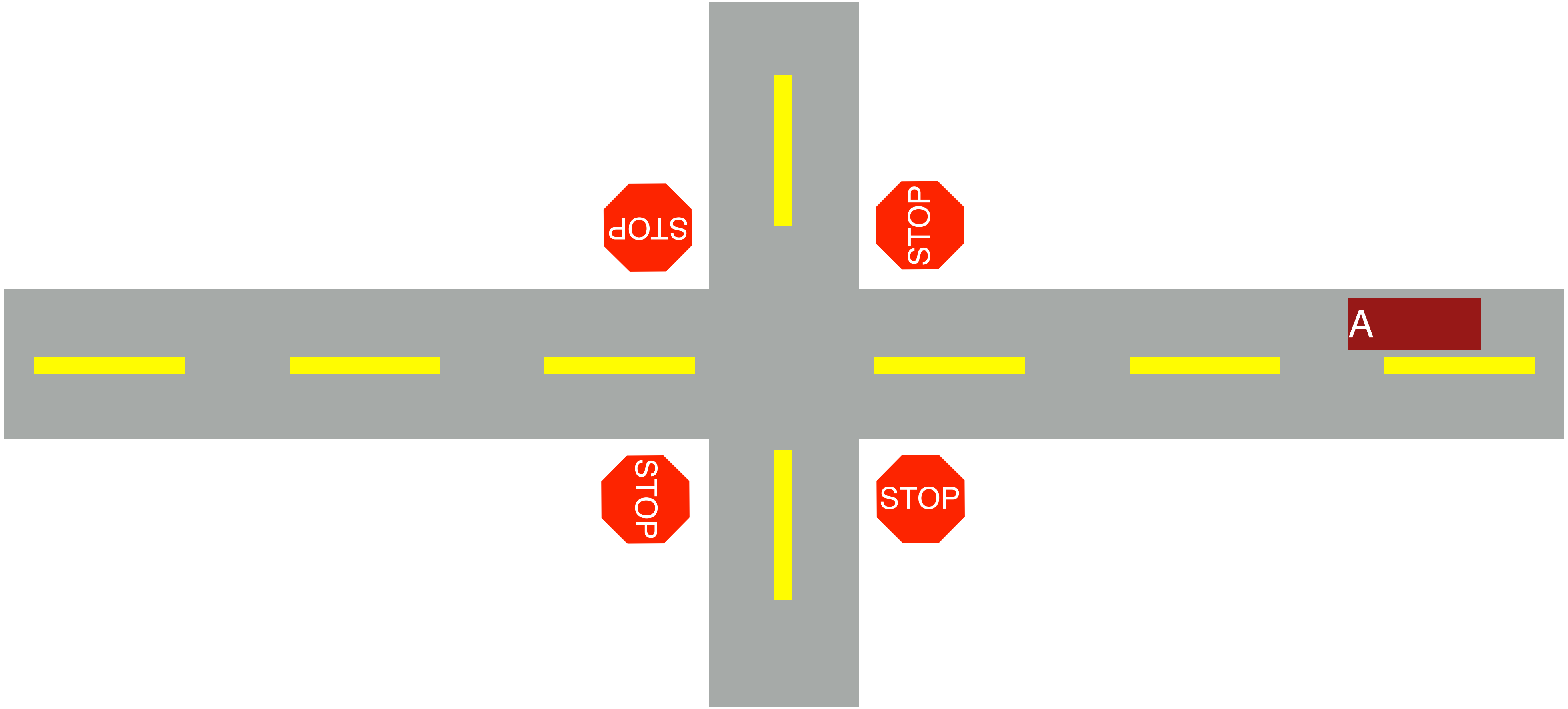
Conditional Wait | **Correct** Usage

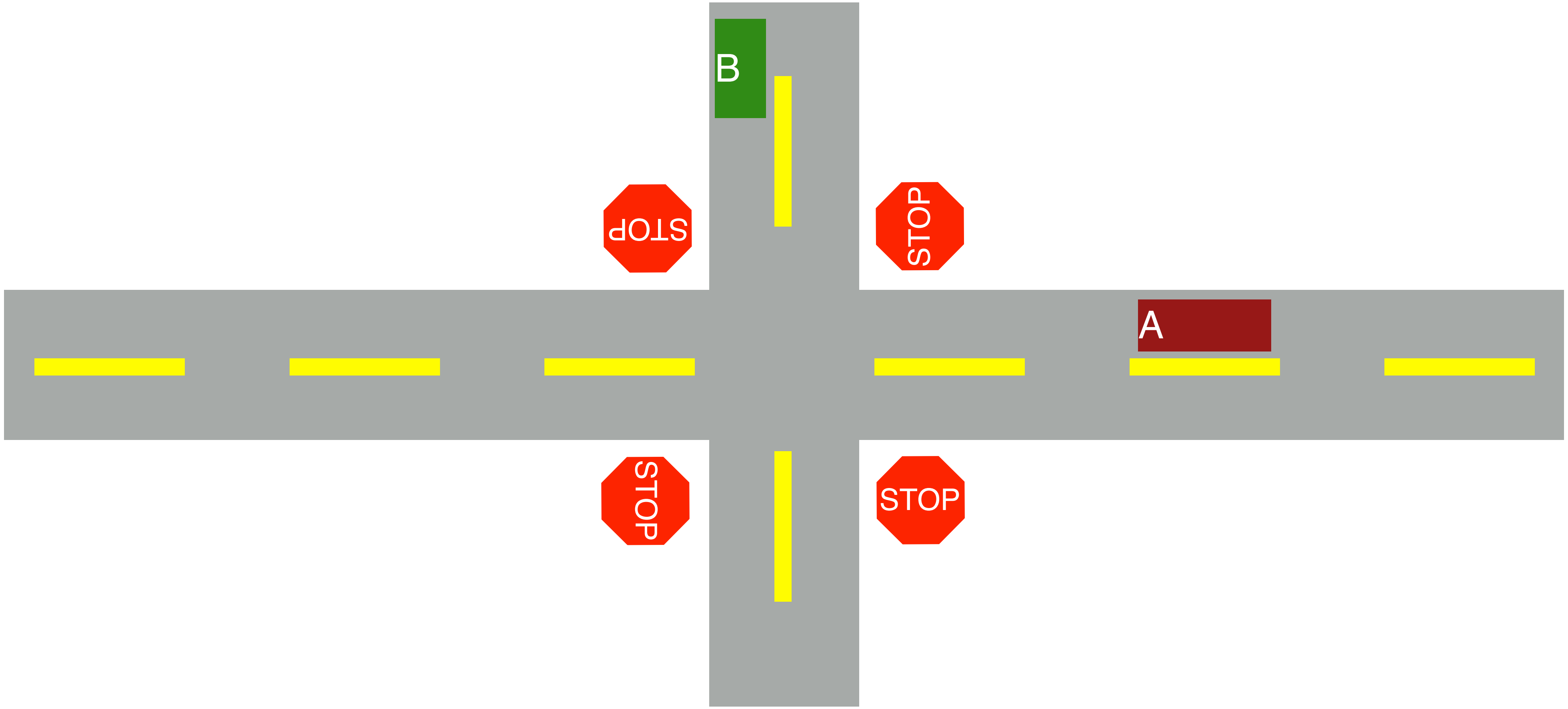
```
Thread 1::
void init() {
        ...
        mThread = PR_CreateThread(mMain, ...);

        //signal that thread has been created
        pthread_mutex_lock(&mtLock);
        mtInit = 1 ;
        pthread_cond_signal(&mtCond);
        pthread_mutex_unlock(&mtLock);
        ...
}
```

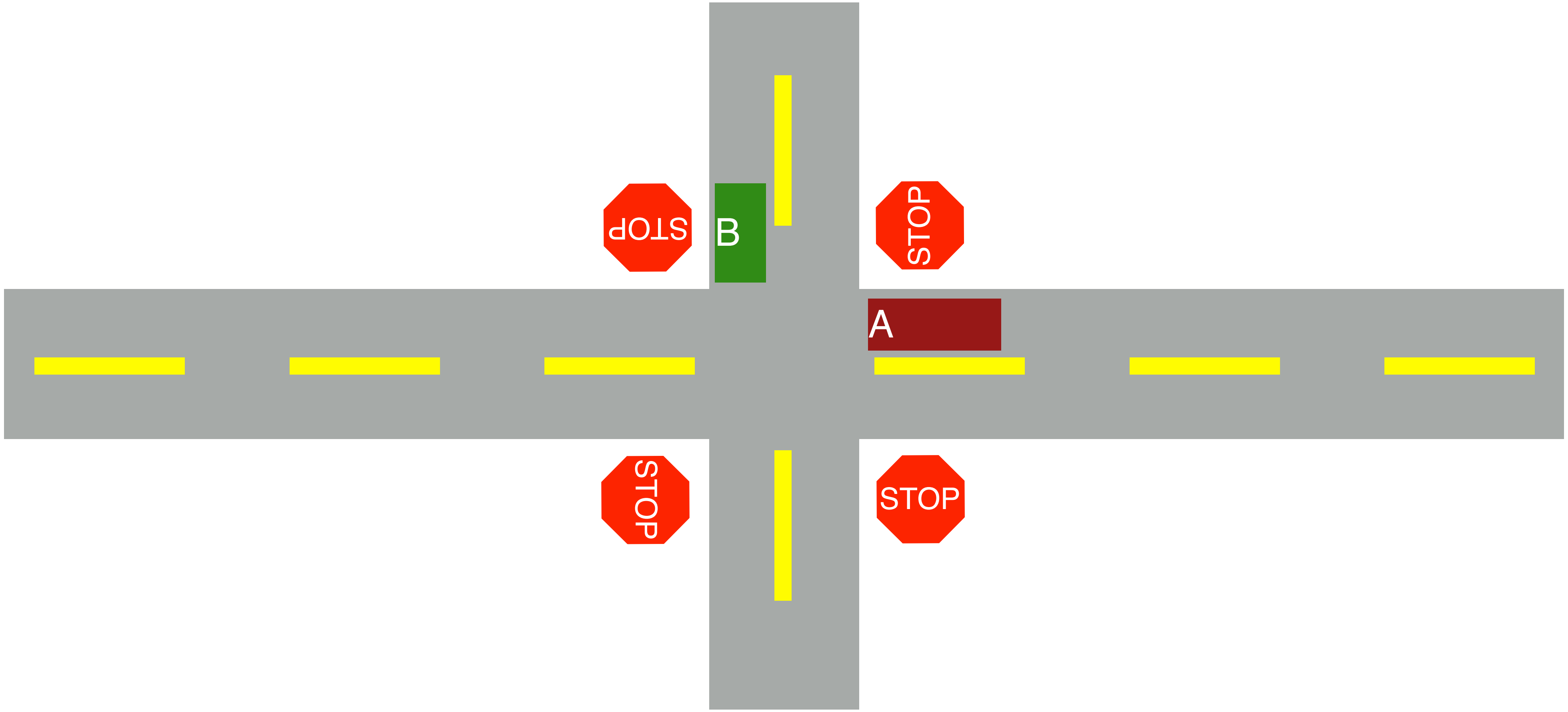
```
Thread 2::
void mMain(...){
        ...
        pthread_mutex_lock(&mtLock);
        while (mtInit ==0)
                pthread_cond_wait(&mtCond,
                                  &mtLock);
        pthread_mutex_unlock(&mtLock);

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

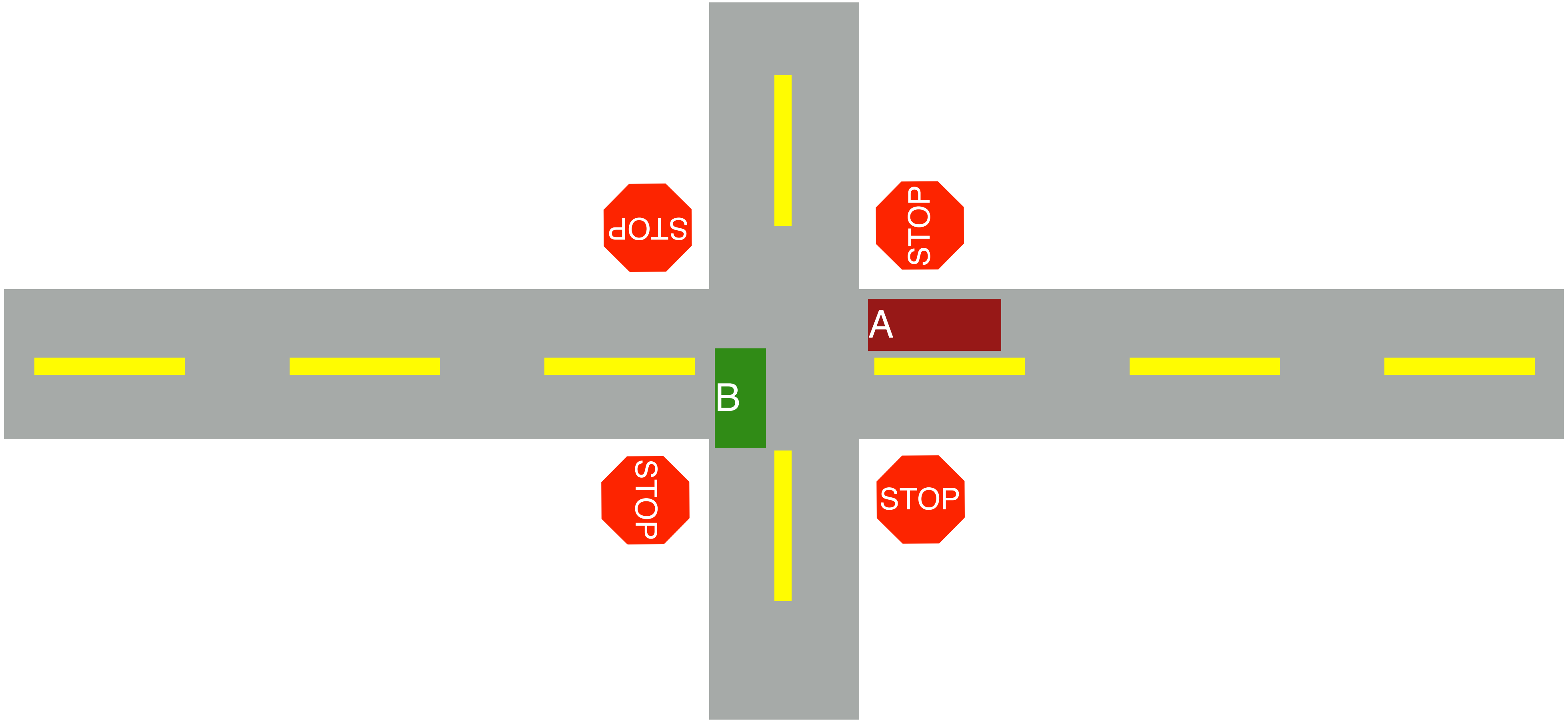


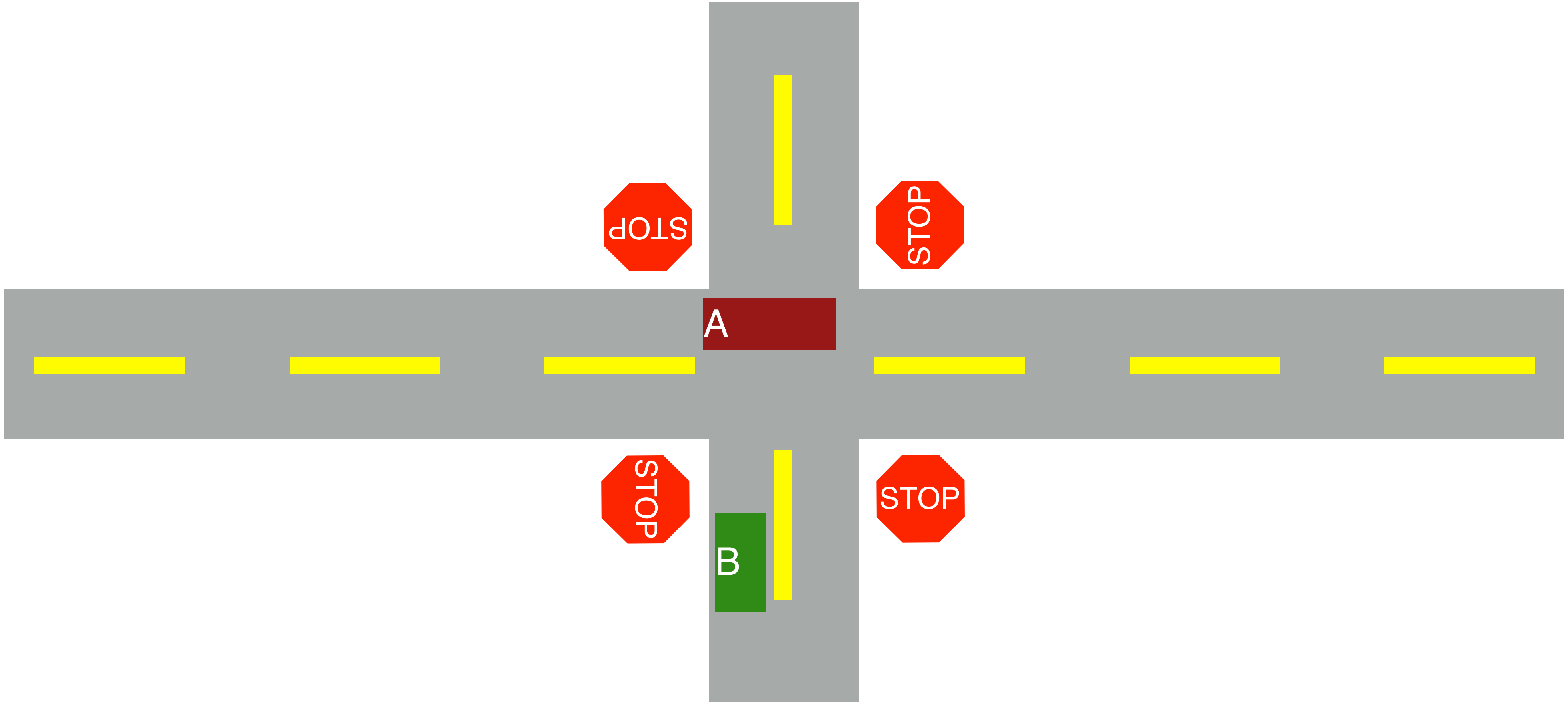
Deadlocks

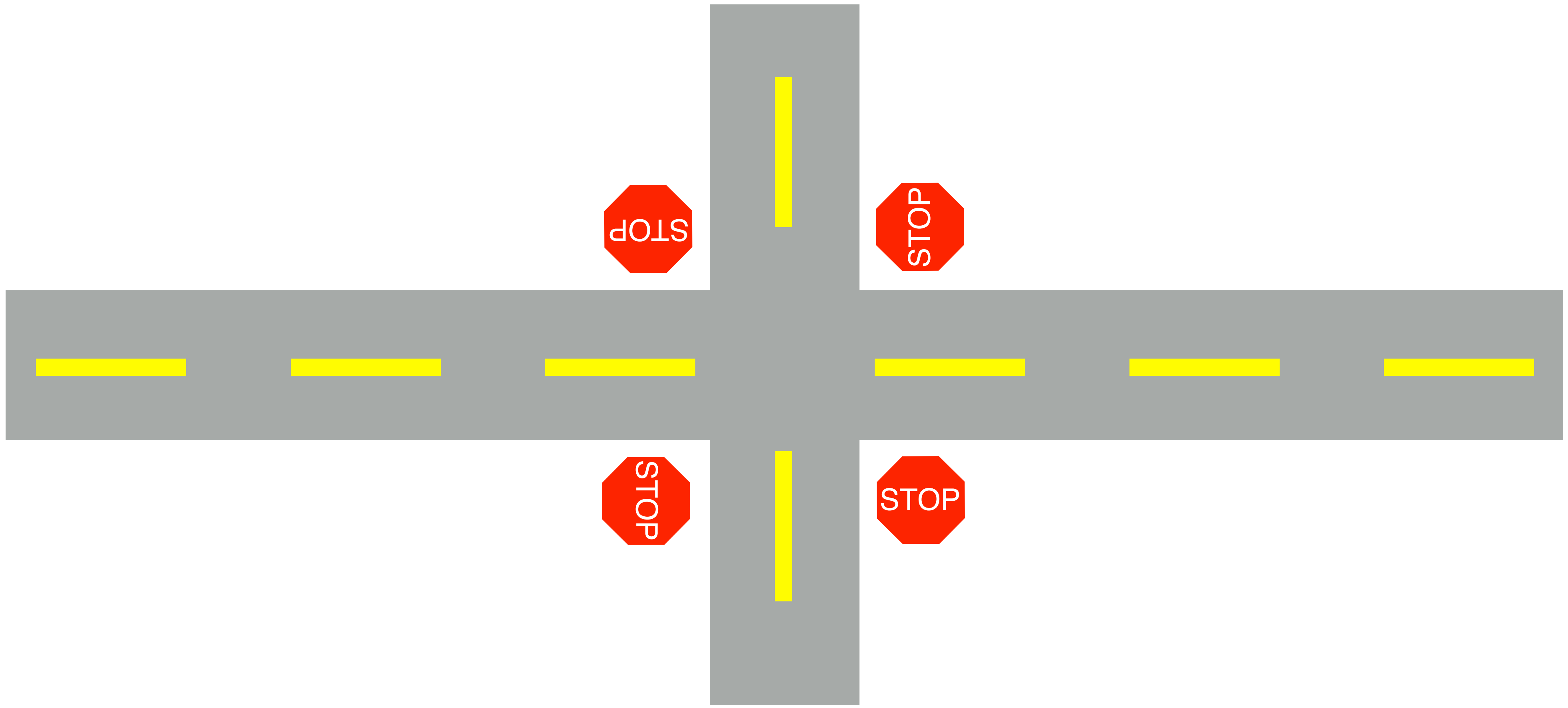

STOP
STOP
A
STOP
STOP

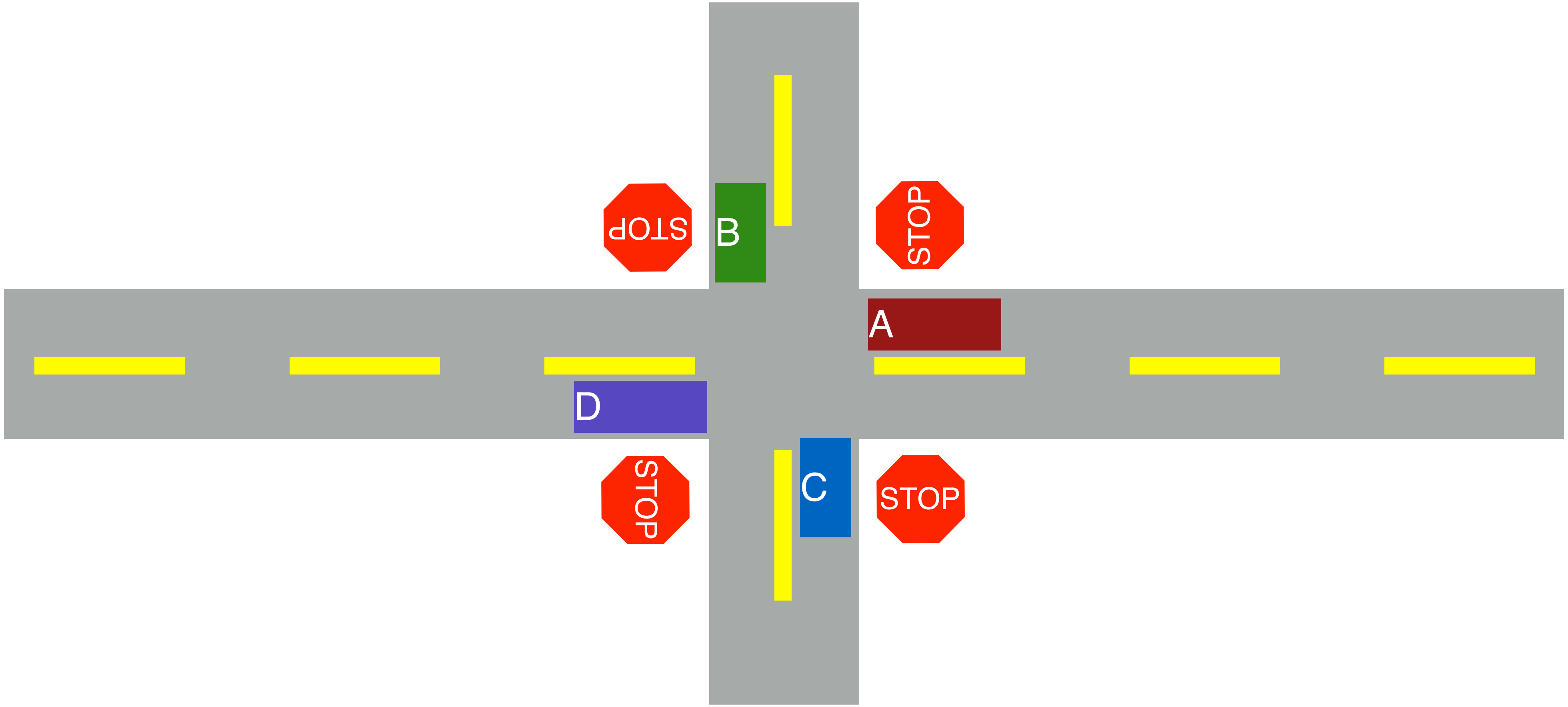

B
STOP
STOP
A
STOP
STOP

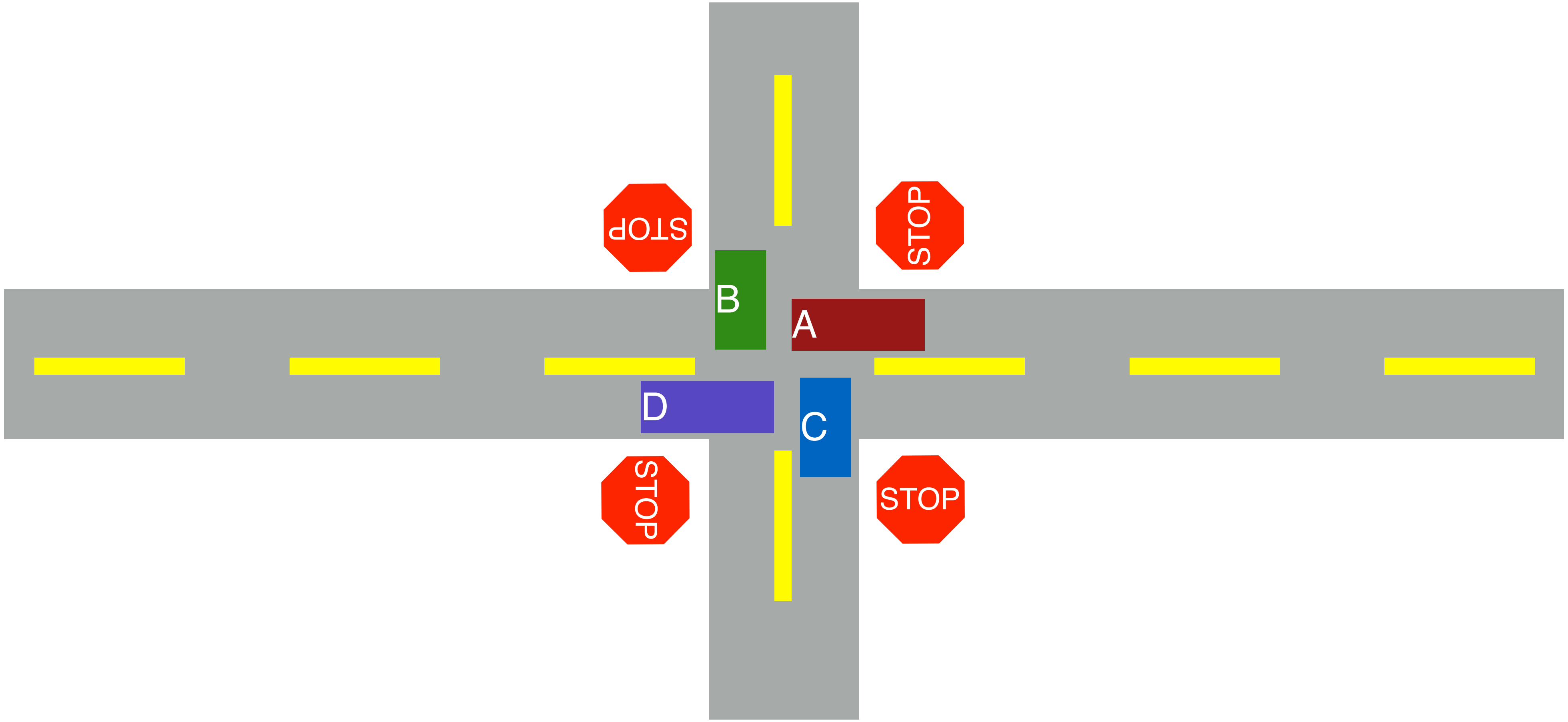

STOP
B
STOP
A
STOP
STOP


STOP
STOP
A
B
STOP
STOP

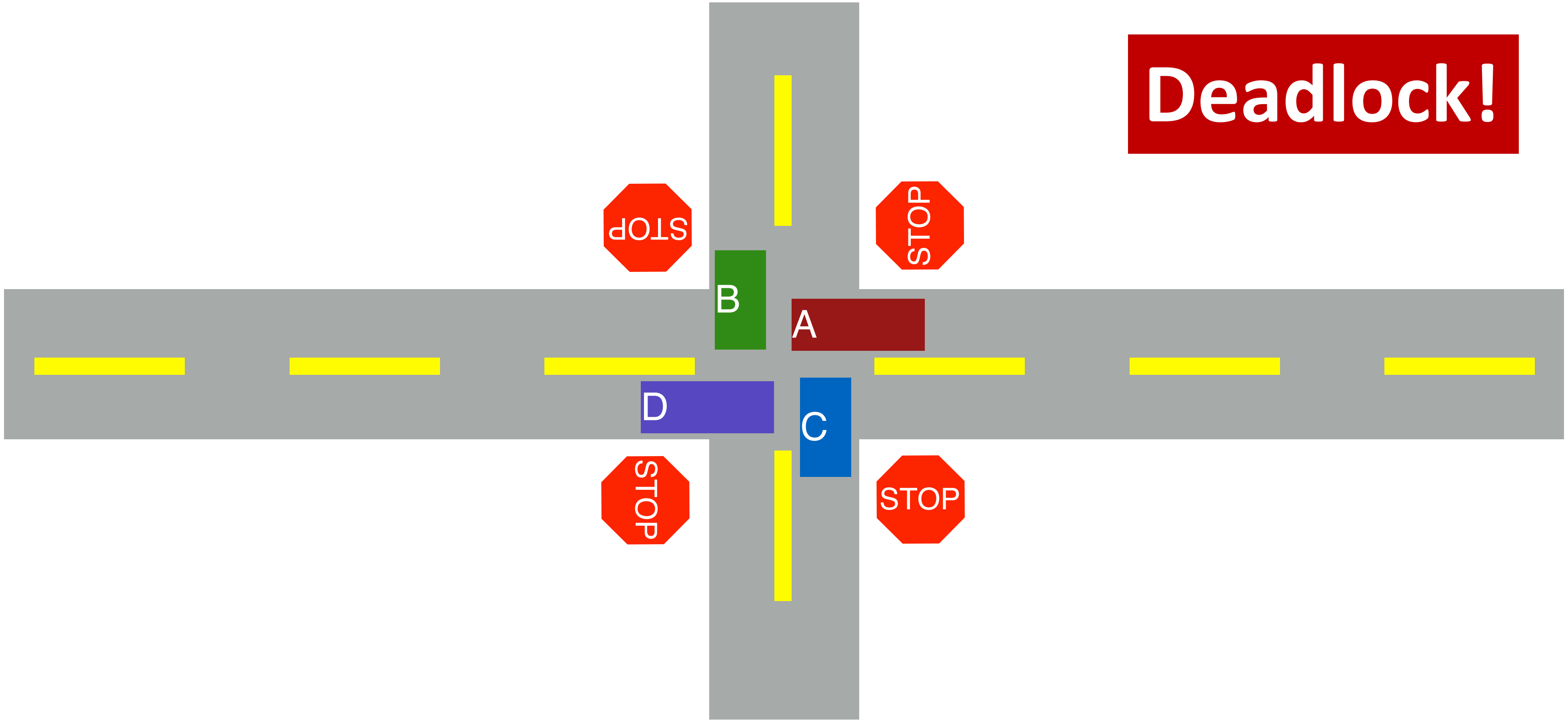

STOP
STOP
A
STOP
STOP
B


STOP
STOP
STOP
STOP


STOP
B
STOP
A
D
C
STOP
STOP

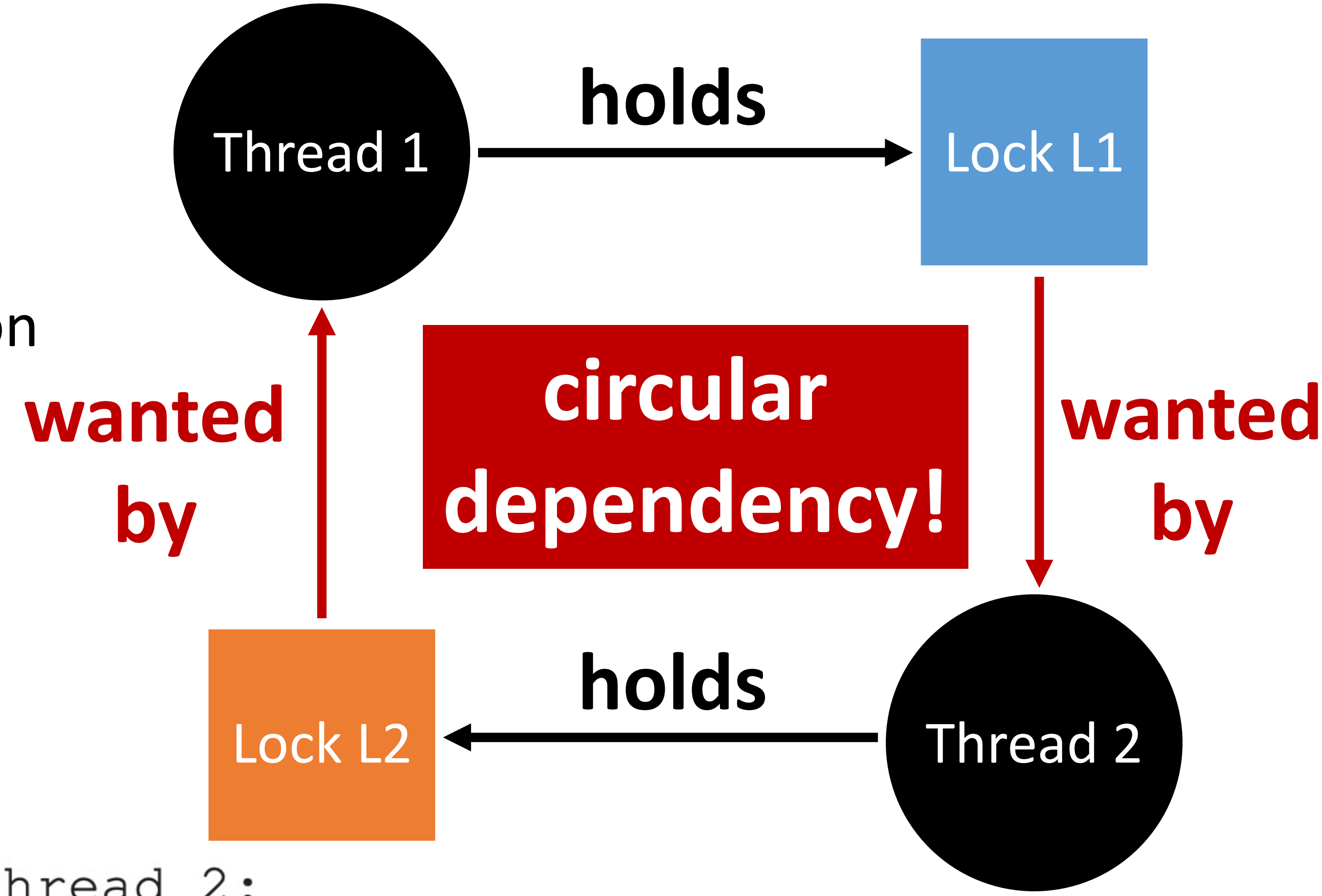

STOP
STOP
B
A
D
C
STOP
STOP

Deadlock!



Deadlock

- **Two or more** threads
- **waiting for other** to take action
- **neither** make any progress




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

How Can We Resolve Circular Dependency

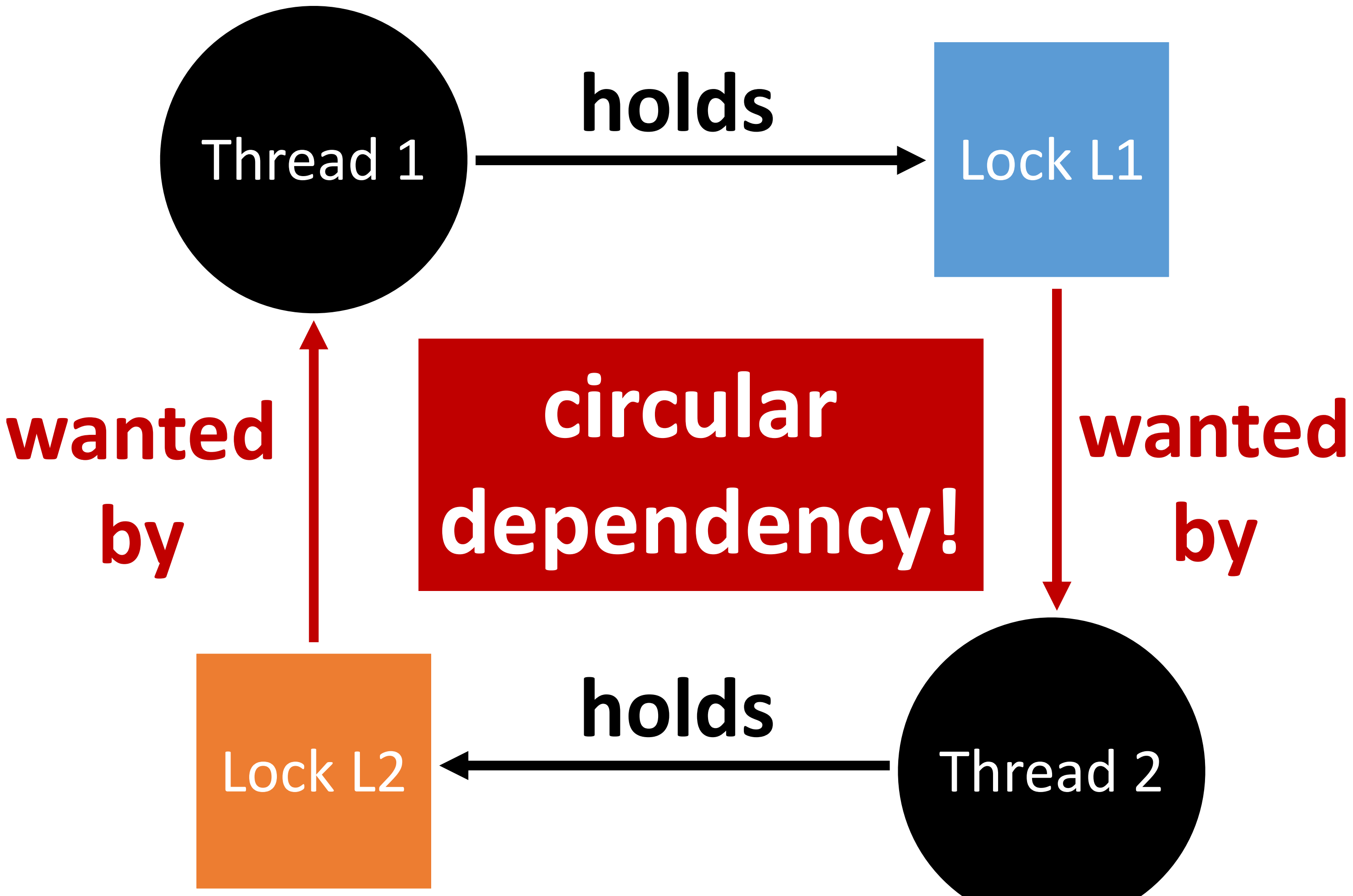
```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

→

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

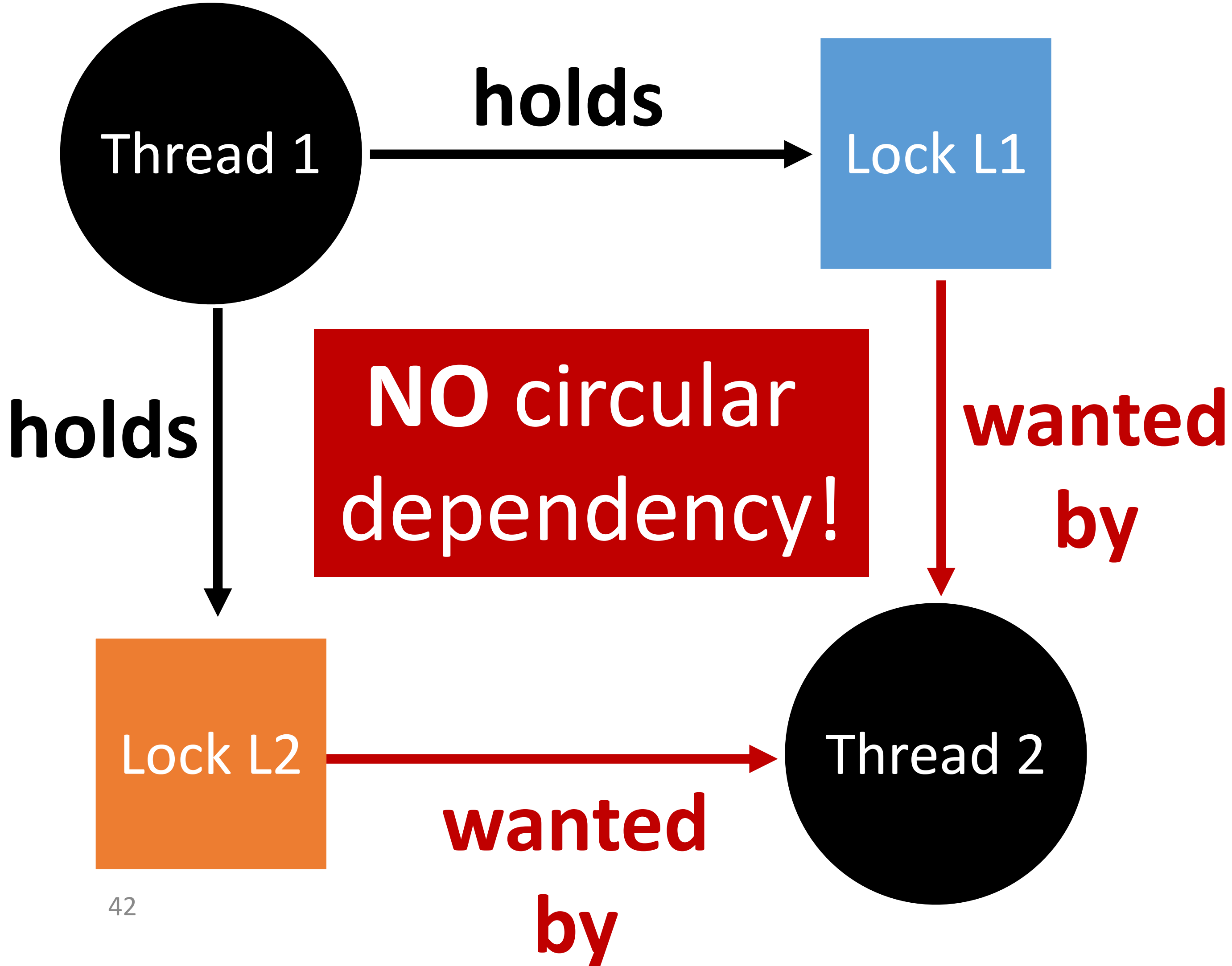
Breaking Circular Dependency

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```



Breaking Circular Dependency

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```



Thread-safe Datastructure

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = new set_t();
    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-safe Datastructure

Thread 1:

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

Thread 2:

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

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    …
    mutex_lock(&s1->lock);
    mutex_lock(&s2->lock);
    …
}
```

Thread-safe Datastructure

Thread 1:

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

   mutex_lock(&setA->lock);
   mutex_lock(&setB->lock);

   …
   mutex_unlock(&setB->lock);
   mutex_unlock(&setA->lock);
```

Thread 2:

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

mutex_lock(&setA->lock);
mutex_lock(&setB->lock);

…
mutex_unlock(&setB->lock);
mutex_unlock(&setA->lock);
```

Is This a Thread-safe Datastructure?

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    set_t *rv = new set_t();
    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);
}
```

Find a Problem..

Thread 1:

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

Thread 2:

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

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    …
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);
    …
}
```

Find the Problem

Thread 1:

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

    Mutex_lock(&setA->lock);
    Mutex_lock(&setB->lock);
```

Thread 2:

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

    Mutex_lock(&setB->lock);
    Mutex_lock(&setA->lock);
```

Deadlock!

Deadlock Theory

- Deadlocks can only happen if threads are having
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait

- We can eliminate deadlock by removing such conditions

Mutual Exclusion

- Definition
  - Threads claims an **exclusive control** of a resource
  - E.g., Threads grabs a lock

How to Remove Mutual Exclusion

- **Do not use lock**
  - What???
- **Replace locks with atomic primitives**
  - **`compare_and_swap(uint64_t *addr, uint64_t prev, uint64_t value);`**
  - if `*addr == prev`, then update `*addr = value`;
  - `lock cmpxchg` in x86

```
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(!comp_and_swap(val, old, old+amt));
}
```

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).

```
mutex_lock(&setA->lock);
mutex_lock(&setB->lock);
```

How to Remove Hold-and-Wait

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**, like this:

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

// Critical section code
unlock(&L1);
unlock(&L2);
```

Remove Hold-and-Wait

```
set_t *set_intersection (set_t *s1, set_t *s2) {
    Mutex_lock(&meta_lock)
    Mutex_lock(&s1->lock);
    Mutex_lock(&s2->lock);
    …
    Mutex_unlock(&s2->lock);
    Mutex_unlock(&s1->lock);
    Mutex_unlock(&meta_lock);
}
```

Remove Hold-and-Wait

Thread 1:

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

    Mutex_lock(&meta_lock);
    Mutex_lock(&setA->lock);
    Mutex_lock(&setB->lock);
    …
    Mutex_unlock(&setB->lock);
    Mutex_unlock(&setA->lock);
    Mutex_unlock(&meta_lock);
```

Thread 2:

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

Mutex_lock(&meta_lock);
Mutex_lock(&setB->lock);
Mutex_lock(&setA->lock);
```

Will wait until Thread 1 finishes (release meta_lock)!

No Preemption

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

```
lock(A);
lock(B);
…
```

In case if B is acquired by other thread

All other threads must wait for acquiring A

How to Remove No Preemption

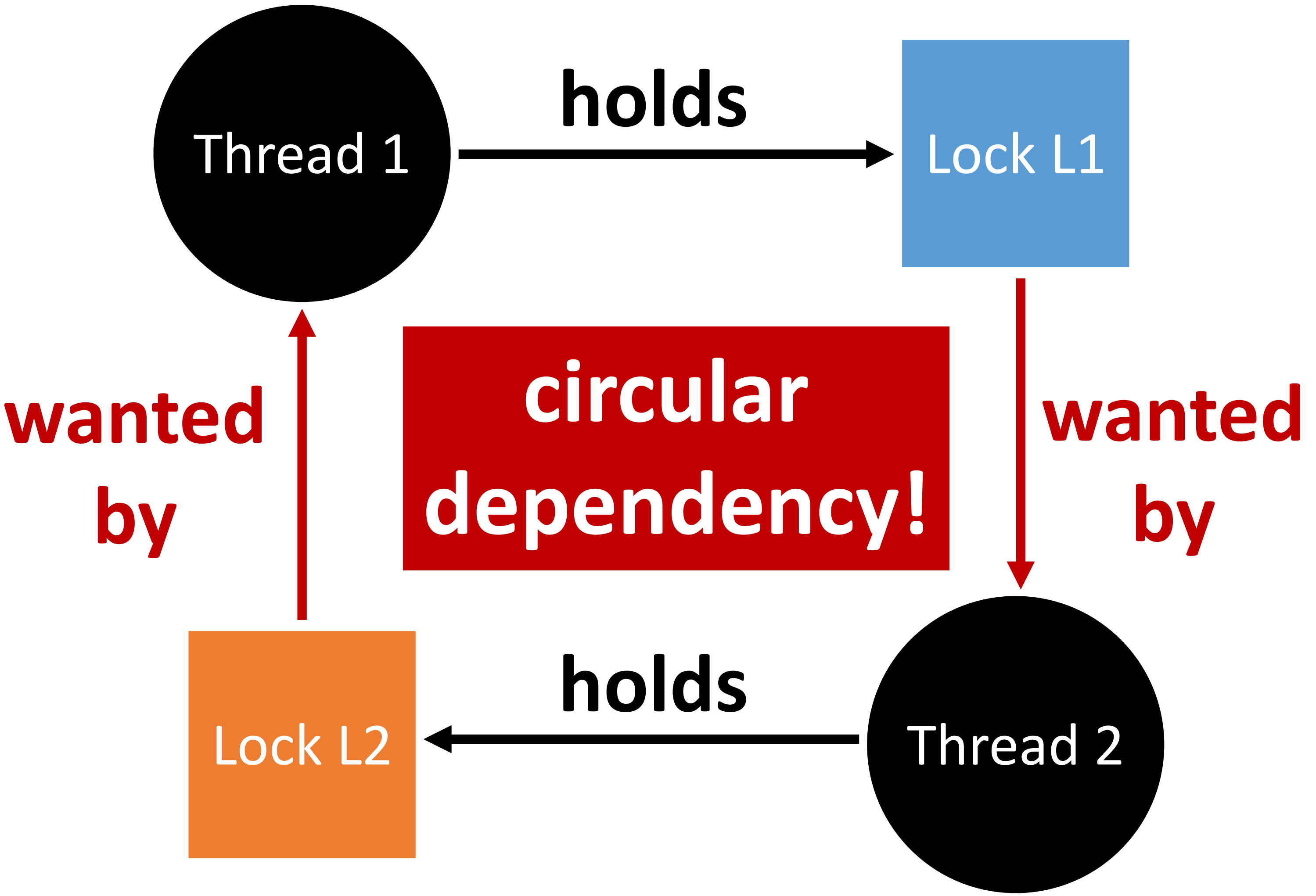
Release the lock if obtaining a resource fails…

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

Can't acquire B, then Release A!

Circular Wait

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.



How to Remove Circular Wait

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

How to Remove Circular Wait

```
Thread 1:
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

Thread 2:
pthread_mutex_lock(L2);
pthread_mutex_lock(L1);
```

Lock variable is mostly a pointer, then provide a correct order of having a lock

```
e.g.,
if(l1 > l2)  {
        mutex_lock(l1);
        mutex_lock(l2);
}
else {
        mutex_lock(l2);
        mutex_lock(l1);
}
```

References

- Some of slides borrowed from here:
  - http://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Andrea/
  - http://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Tyler/oct22/bugs.pdf
- Some of code snippets borrowed from here:
  - http://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf