Operating Systems

COMP SCI 3004 / COMP SCI 7064

Week 7 – Semaphores

Introduction

- Semaphores
- Race Condition problems
  - Deadlocks

THE UNIVERSITY of ADELAIDE

Review

THE UNIVERSITY of ADELAIDE

Higher-level Primitives

What is right abstraction for synchronizing threads that share memory?

Want as high a level primitive as possible

Good primitives and practices important!

Since execution is not entirely sequential, really hard to find bugs, since they happen rarely

UNIX is pretty stable now, but up until about mid-80s
(10 years after started), systems running UNIX would crash every week or so – concurrency bugs

Synchronization is a way of coordinating multiple concurrent activities that are using shared state

Concurrency Objectives

- **Mutual exclusion (e.g., A and B don’t run at same time)**
  - solved with locks
- **Ordering (e.g., B runs after A does something)**
  - solved with condition variables and semaphores

THE UNIVERSITY of ADELAIDE

Condition Variables

- **wait** `(cond_t *cv, mutex_t *lock)`
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- **signal** `(cond_t *cv)`
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

THE UNIVERSITY of ADELAIDE

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}
```

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

THE UNIVERSITY of ADELAIDE

Join Implementation: Correct

Parent:

```
void thread_join() {
        Mutex_lock(&m);         // w
        if (done == 0)          // x
            Cond_wait(&c, &m);// y
        Mutex_unlock(&m);       // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);     // a
        done = 1;           // b
        Cond_signal(&c);    // c
        Mutex_unlock(&m);   // d
}
```

Parent:	w	x	y				z
Child:				a	b	c	

Use mutex to ensure no race between interacting with state and wait/signal

THE UNIVERSITY of ADELAIDE

Join Implementation: Correct

- **Producers generate data (like pipe writers)**
- **Consumers grab data and process it (like pipe readers)**
  e.g., `grep 'xyz' data.txt | sort | uniq -c`

- **Use condition variables to:**
  - make producers wait when buffers are full
  - make consumers wait when there is nothing to consume

THE UNIVERSITY of ADELAIDE

Broken Implementation of Producer Consumer

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);             //p1
        while(numfull == max)       //p2
           Cond_wait(&cond, &m);  //p3
        do_fill(i);                 //p4
        Cond_signal(&cond);         //p5
        Mutex_unlock(&m);           //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);               //c1
        while(numfull == 0)           //c2
            Cond_wait(&cond, &m);  //c3
        int tmp = do_get();           //c4
        Cond_signal(&cond);           //c5
        Mutex_unlock(&m);             //c6
        printf("%d\n", tmp);          //c7
    }
}
```

			wait()			wait()				signal()				wait()			signal()
Producer:							p1	p2	p4	p5	p6	p1	p2	p3			
Consumer1:	c1	c2	c3														
Consumer2:				c1	c2	c3									c2	c4	c5

Does the last signal wake producer or consumer2

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);          //p1
        while(numfull == max)    //p2
           Cond_wait(&empty, &m); //p3
        do_fill(i);              //p4
        Cond_signal(&fill);      //p5
        Mutex_unlock(&m);        //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);              //c1
        if(numfull == 0)             //c2
            Cond_wait(&fill, &m);   //c3
        int tmp = do_get();          //c4
        Cond_signal(&empty);         //c5
        Mutex_unlock(&m);            //c6
        printf("%d\n", tmp);         //c7
    }
}
```

Is this correct? Can you find a bad schedule?

1. consumer1 waits because numfull == 0
2. producer increments numfull, wakes consumer1
3. before consumer1 runs, consumer2 runs, grabs entry, sets numfull=0.
4. consumer2 then reads bad data.

Producer:				p1	p2	p4	p5	p6			
Consumer1:	c1	c2	c3								c4! ERROR
Consumer2:										c1 c2 c4 c5 c6	

CV Rules

- **Whenever a lock is acquired, recheck assumptions about state!**
  - Use “while” intead of “if”
- **Possible for another thread to grab lock between signal and wakeup from wait**
  - Difference between Mesa (practical implementation) and Hoare (theoretical) semantics
  - signal() simply makes a thread runnable, does not guarantee thread run next
- **Note that some libraries also have “spurious wakeups”**
  - May wake multiple waiting threads at signal or at any time

THE UNIVERSITY of ADELAIDE

Producer/Consumer: Two CVs

```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        Mutex_lock(&m);             //p1
        while(numfull == max)       //p2
           Cond_wait(&empty, &m); //p3
        do_fill(i);                 //p4
        Cond_signal(&fill);         //p5
        Mutex_unlock(&m);           //p6
    }
}
```

```
void *consumer(void *arg) {
    while(1) {
        Mutex_lock(&m);              //c1
        while(numfull == 0)          //c2
            Cond_wait(&fill, &m);  //c3
        int tmp = do_get();          //c4
        Cond_signal(&empty);         //c5
        Mutex_unlock(&m);            //c6
        printf("%d\n", tmp);         //c7
    }
}
```

Is this correct? Can you find a bad schedule?

Correct!
- no concurrent access to shared state
- every time lock is acquired, assumptions are reevaluated
- a consumer will get to run after every `do_fill()`
- a producer will get to run after every `do_get()`

Summary: Rules CVs

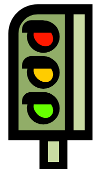
- Keep state in addition to CV's
- Always do wait/signal with lock held
- Whenever thread wakes from waiting, recheck state

THE UNIVERSITY of ADELAIDE

Condition Variables vs Semaphores

- **Condition variables have no state (other than the waiting queue)**
  - Programmer must track additional state
- **Semaphores have state: track integer value**
  - State cannot be directly accessed by user program, but state determines the behaviour of semaphore operations

Semaphores



Semaphores are a kind of generalized lock

First defined by Dijkstra in late 60s

Main synchronization primitive used in original UNIX

Definition: a Semaphore has a non-negative integer value and supports the following operations:

Set value when you initialize

wait(): an atomic operation that waits for semaphore to become positive, then decrements it by 1

post(): an atomic operation that increments the semaphore by 1, waking up a waiting `wait()`, if any. (This of this as the signal() operation).

Semaphores Like Integers Except…

Semaphores are like integers, except:

No negative values

Only operations allowed are *wait* and *post* – can't read or write value, except initially

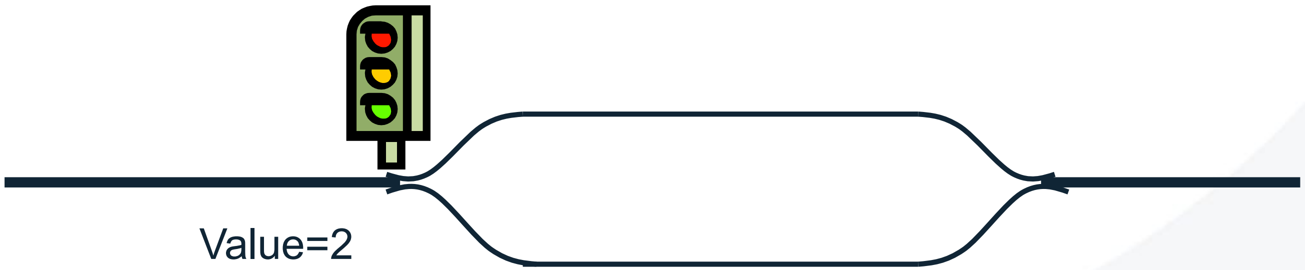
Operations must be atomic

- Two *wait*'s together can not decrement value below zero
- Thread going to sleep in *wait* will not miss wakeup from *post* – even if both happen at same time.

POSIX adds ability to read value, but technically not part of proper interface!

Semaphore from railway analogy

Here is a semaphore initialized to 2 for resource control:



Value=2

THE UNIVERSITY of ADELAIDE

Semaphore Operations

- **Allocate and Initialize**

```
sem_t sem;
sem_init(sem_t *s, int initval) {
    s->value = initval;
}
```

  - User cannot read or write value directly after initialization
- **Wait or Test (sometime P() for a Dutch word)**
  - Waits until value of sem is > 0, then decrements sem value
- **Signal or Increment or Post (sometime V() for a Dutch)**
  - Increment sem value, then wake a single waiter
  - wait and post are atomic

THE UNIVERSITY of ADELAIDE

Join with CV vs Semaphores

CVs:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
            Cond_wait(&c, &m);       // y
        Mutex_unlock(&m);            // z
}
```

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

Semaphores:

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

```
sem_t s;
sem_init(&s, ???);
```

Initialize to 0 (so sem_wait() must wait…)

```
void thread_join() {
        sem_wait(&s);
}
```

```
void thread_exit() {
        sem_post(&s)
}
```

Equivalence Claim

- **Semaphores are equally powerful to Locks+CVs**
  - what does this mean?
- **One might be more convenient, but that’s not relevant**
- **Equivalence means each can be built from the other**

Proof Steps

- **Want to show we can do these three things:**

Build Lock from Semaphore

```
typedef struct __lock_t {
    sem_t sem;
} lock_t;

void init(lock_t *lock) {
    sem_init(&lock->sem, ??);
}
void acquire(lock_t *lock) {
    sem_wait(&lock->sem);
}
void release(lock_t *lock) {
    sem_post(&lock->sem);
}
```

1 → 1 thread can grab lock

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Locks

Semaphores

THE UNIVERSITY of ADELAIDE

Building CV's over Semaphores

- **Possible, but really hard to do right**

CV's

Semaphores

- Read about Microsoft Research's attempts:
- https://research.microsoft.com/apps/pubs/default.aspx?id=64242&type=exact

Build Semaphore from Lock and CV

```
typedef struct {
    // what goes here?


} sem_t;

void sem_init(sem_t *s, int value) {
    // what goes here?


}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks CV's

THE UNIVERSITY of ADELAIDE

Build Semaphore from Lock and CV

```
typedef struct {
    int value;
    cond_t cond;
    lock_t lock;
} sem_t;

void sem_init(sem_t *s, int value) {
    s->value = value;
    cond_init(&s->cond);
    lock_init(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks | CV's

THE UNIVERSITY of ADELAIDE

Build Semaphore from Lock and CV

```
sem_wait{sem_t *s) {
    lock_acquire(&s->lock);
    // this stuff is atomic


    lock_release(&s->lock);
}
```

```
sem_post{sem_t *s) {
    lock_acquire(&s->lock);
     // this stuff is atomic

    lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks CV's

THE UNIVERSITY of ADELAIDE

Build Semaphore from Lock and CV

```
sem_wait{sem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond, &s->lock);
    s->value--;
    lock_release(&s->lock);
}
```

```
sem_post{sem_t *s) {
    lock_acquire(&s->lock);
     // this stuff is atomic

    lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores

Locks | CV's

Build Semaphore from Lock and CV

```
sem_wait{sem_t *s) {
    lock_acquire(&s->lock);
    while (s->value <= 0)
        cond_wait(&s->cond, &s->lock);
    s->value--;
    lock_release(&s->lock);
}
```

```
sem_post{sem_t *s) {
    lock_acquire(&s->lock);
    s->value++;
    cond_signal(&s->cond);
    lock_release(&s->lock);
}
```

Sem_wait(): Waits until value > 0, then decrement
Sem_post(): Increment value, then wake a single waiter

Semaphores	
Locks	CV's

THE UNIVERSITY of ADELAIDE

Revisit Bounded Buffer: Correctness constraints for solution

Correctness Constraints:

Consumer must *wait* for producer to fill buffers, if none full (scheduling constraint)

Producer must *wait* for consumer to empty buffers, if all full (scheduling constraint)

Only one thread can manipulate buffer queue at a time (mutual exclusion)

Remember we need mutual exclusion

General rule of thumb: Use a separate semaphore for each constraint

```
sem_t fullBuffers; // consumer's constraint
sem_t emptyBuffers;// producer's constraint
sem_t mutex;       // mutual exclusion
```

Bounded Buffer

```
sem_t fullSlots = 0;          // Initially, no slots
sem_t emptySlots = bufSize; // Initially, max empty slots
sem_t mutex = 1;              // No one using critical section

Producer(item) {
      sem_wait(&emptySlots); // Wait until space
      sem_wait(&mutex);      // Wait for critical section
      Enqueue(item);
      sem_post(&mutex);
      sem_post(&fullSlots);  // Tell consumers to consu
}
      Consumer() {
      sem_wait(&fullSlots);  // Check if there are item
      sem_wait(&mutex);      // Wait for critical section
      item = Dequeue();
      sem_post(&mutex);
      sem_post(&emptySlots); // tell producer need more
      return item;
}
```

Critical sections using mutex protect integrity of the queue

fullSlots signals consumer

emptySlots signals space

THE UNIVERSITY of ADELAIDE

Why asymmetry?

Decrease # of empty slots

Increase # of occupied slots

Producer does: `sem_wait(&emptyBuffer), sem_post(&fullBuffer)`

Consumer does: `sem_wait(&fullBuffer), sem_post(&emptyBuffer)`

Decrease # of occupied slots

Increase # of empty slots

Is order of wait's important?

Is order of post's important?

What if we have 2 producers or 2 consumers?

```
Producer(item) {
  sem_wait(&mutex);
  sem_wait(&emptySlots);
  Enqueue(item);
  sem_post(&mutex);
  sem_post(&fullSlots);
}
Consumer() {
  sem_wait(&fullSlots);
  sem_wait(&mutex);
  item = Dequeue();
  sem_post(&mutex);
  sem_post(&emptySlots);
  return item;
}
```

THE UNIVERSITY of ADELAIDE

Summary: Semaphores

- Semaphores are equivalent to locks + condition variables
- Semaphores contain state
- sem_wait(): Waits until value > 0, then decrement (atomic)
- sem_post(): Increment value, then wake a single waiter (atomic)
- Can use semaphores in producer/consumer relationships and for reader/writer locks

THE UNIVERSITY *of* ADELAIDE

COMP SCI 3004
Operating Systems

Concurrency Bugs

“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.”

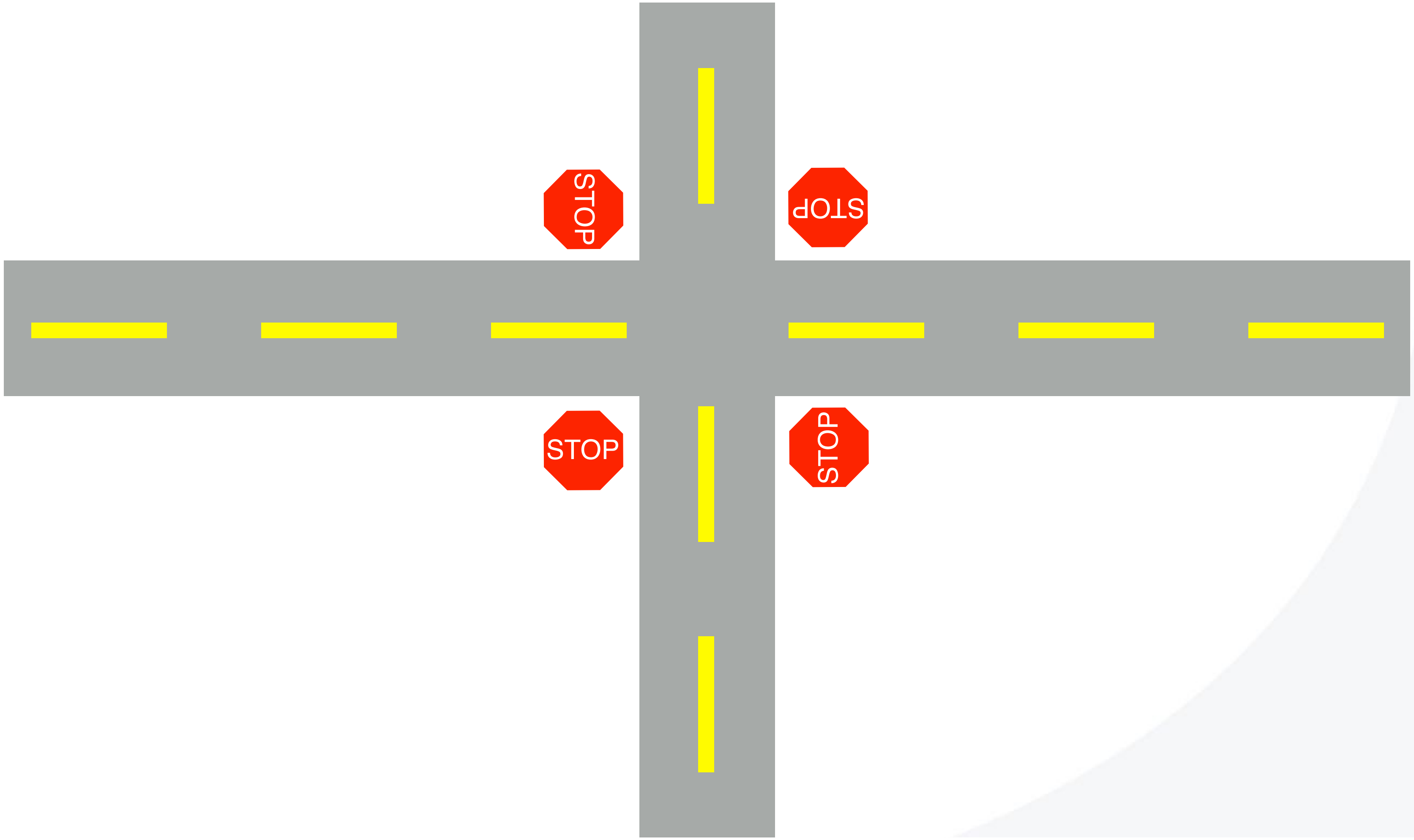
“…in three cases, the injured patients later died.”

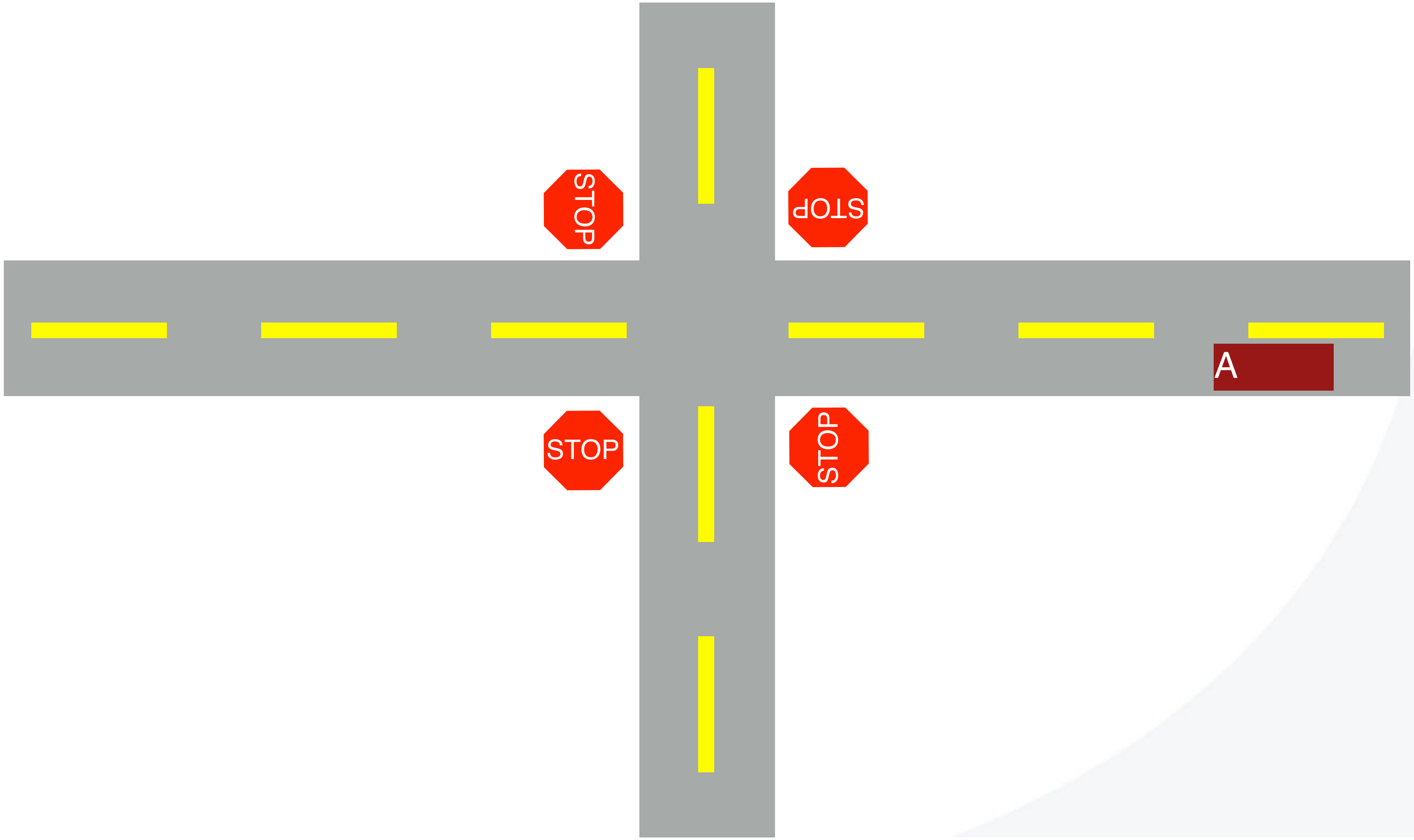
Source: http://en.wikipedia.org/wiki/Therac-25

THE UNIVERSITY of ADELAIDE

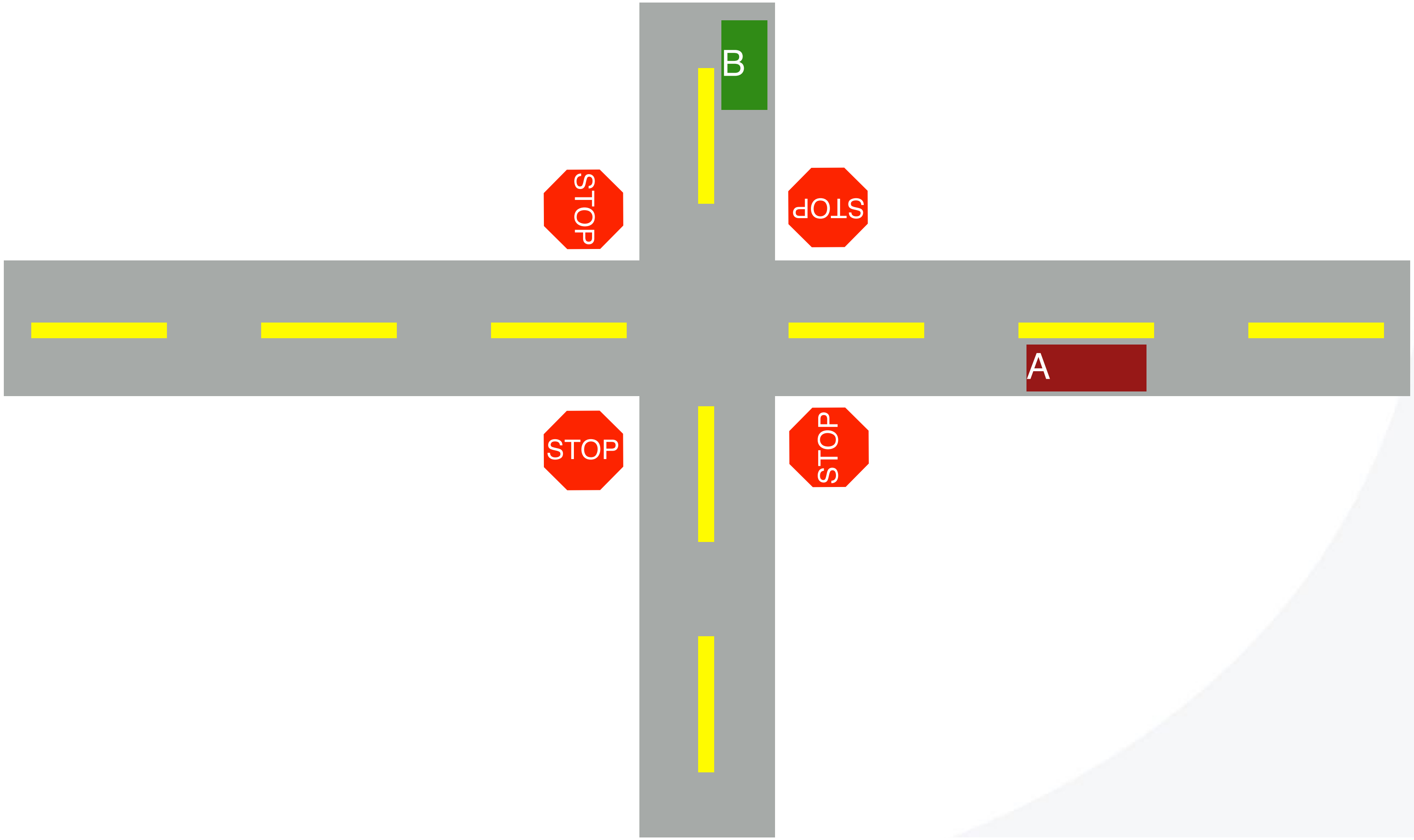
Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does

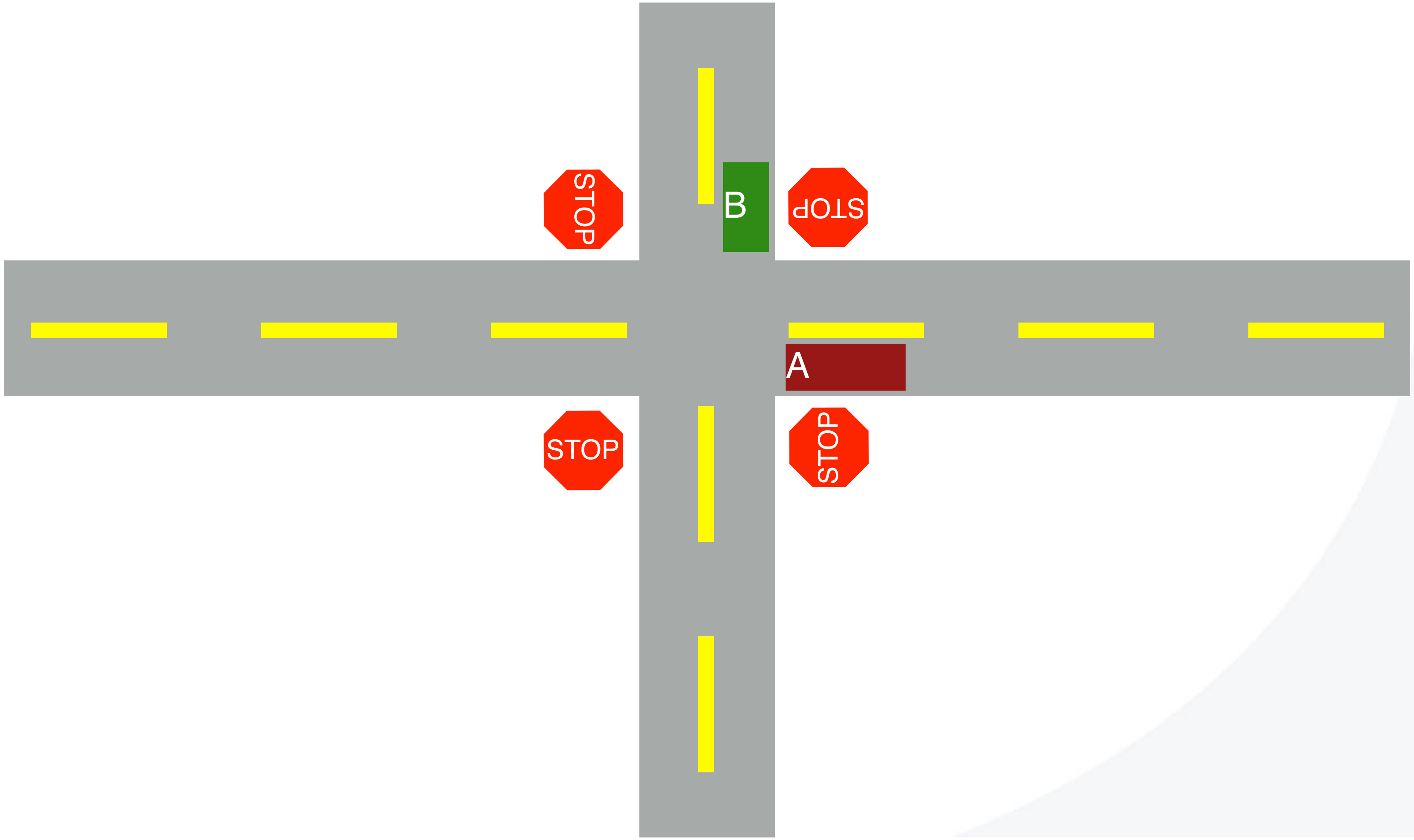
“Cooler" name: the deadly embrace (Dijkstra)

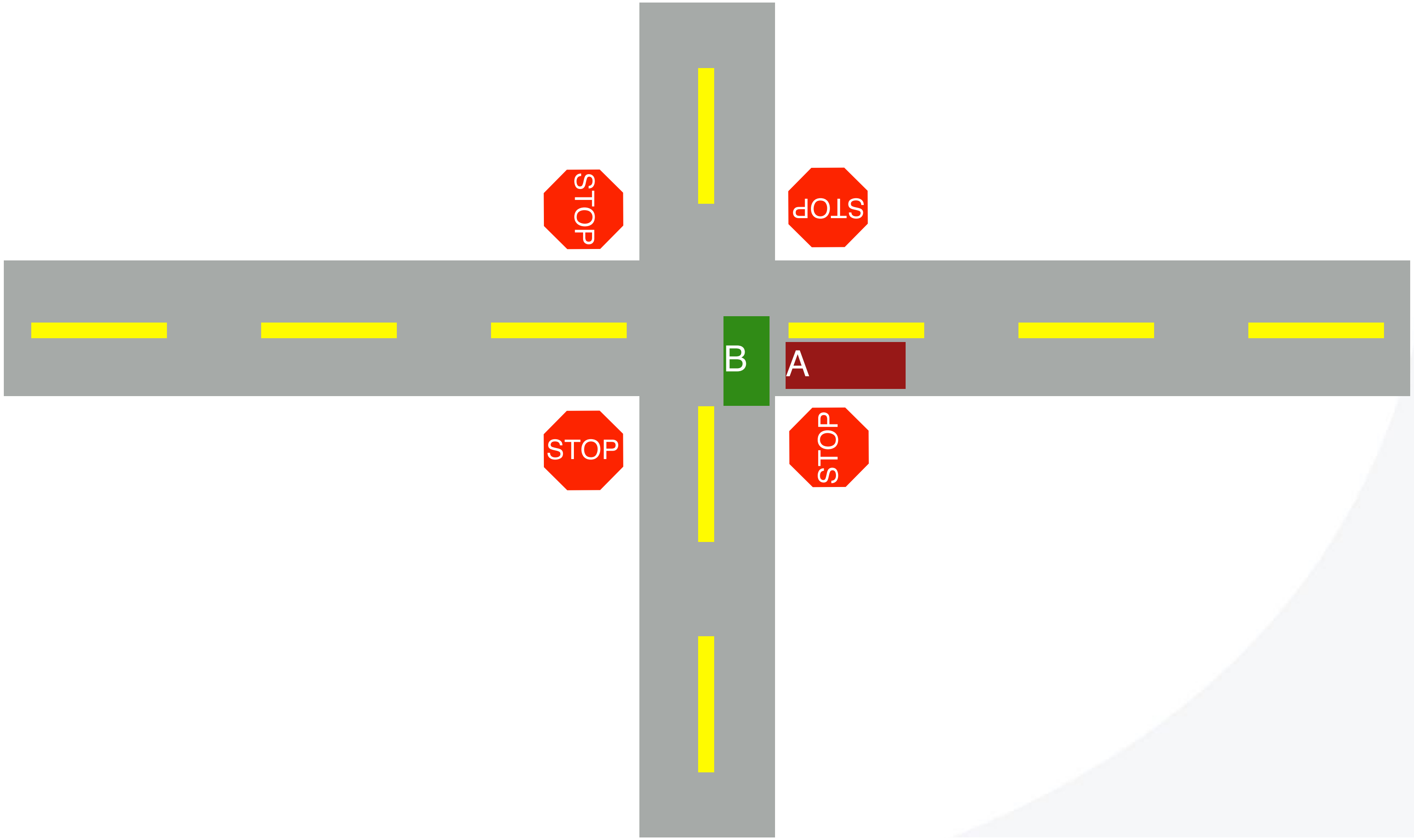

STOP
STOP
STOP
STOP



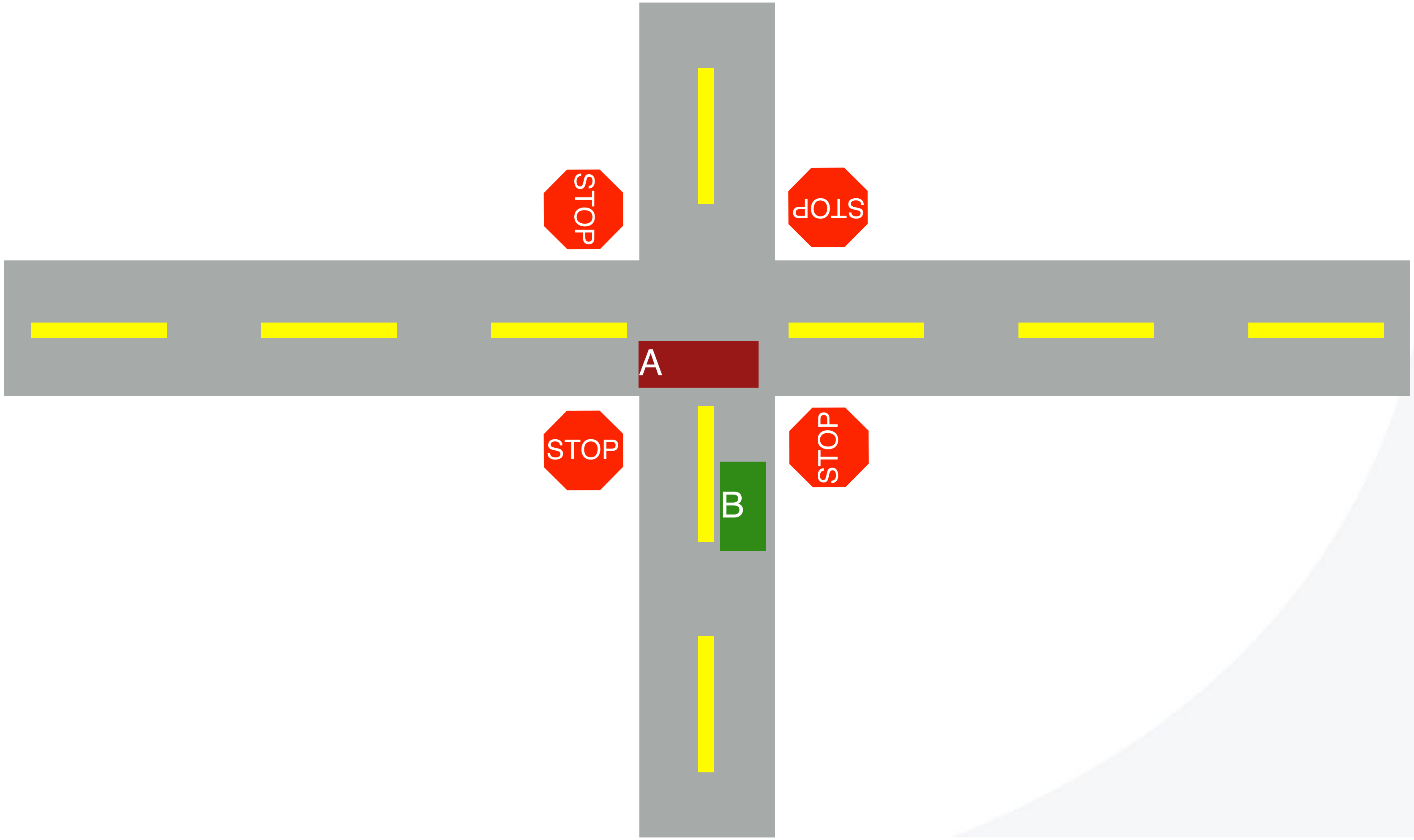
THE UNIVERSITY of ADELAIDE


B
STOP
STOP
A
STOP
STOP

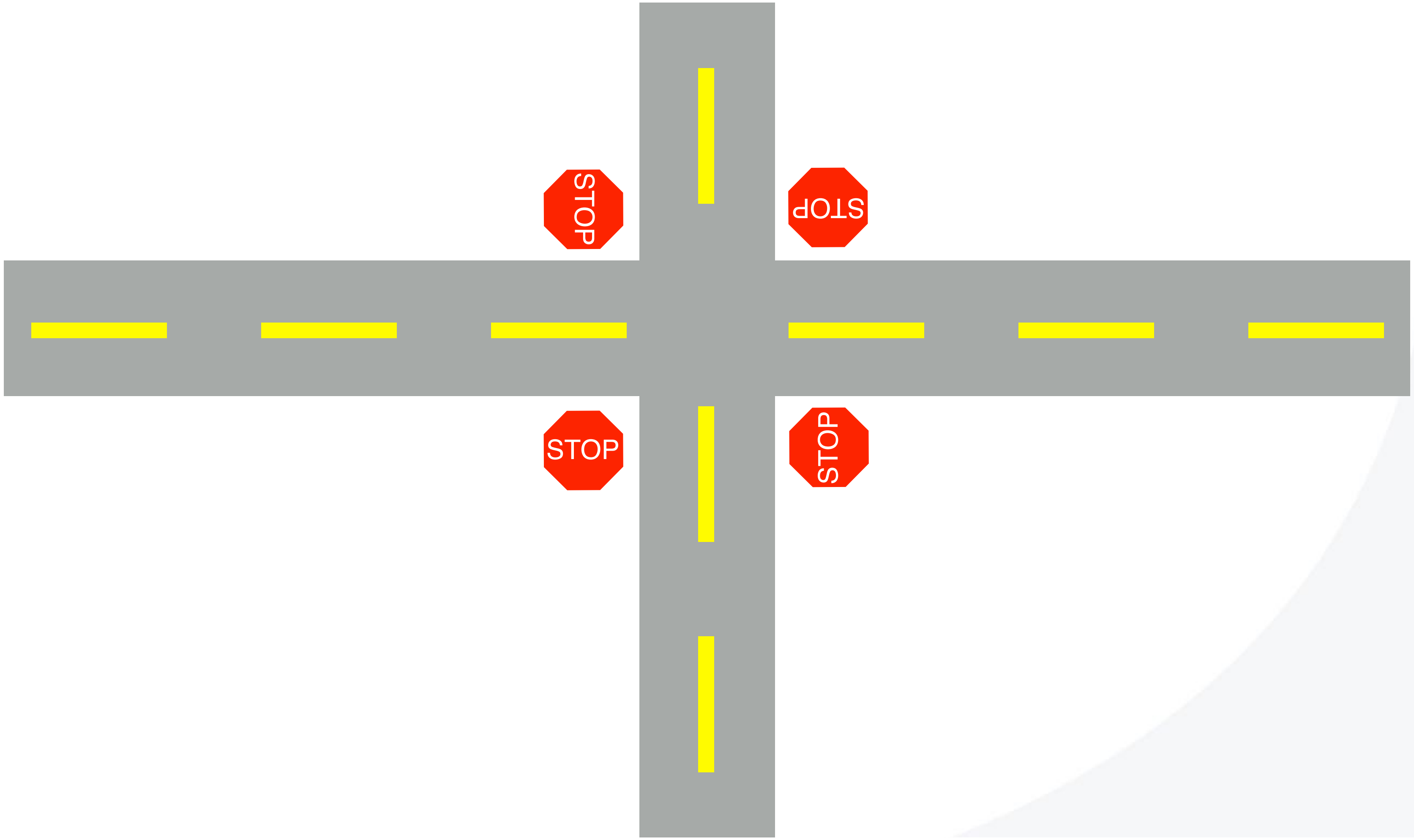

STOP
B
STOP
A
STOP
STOP



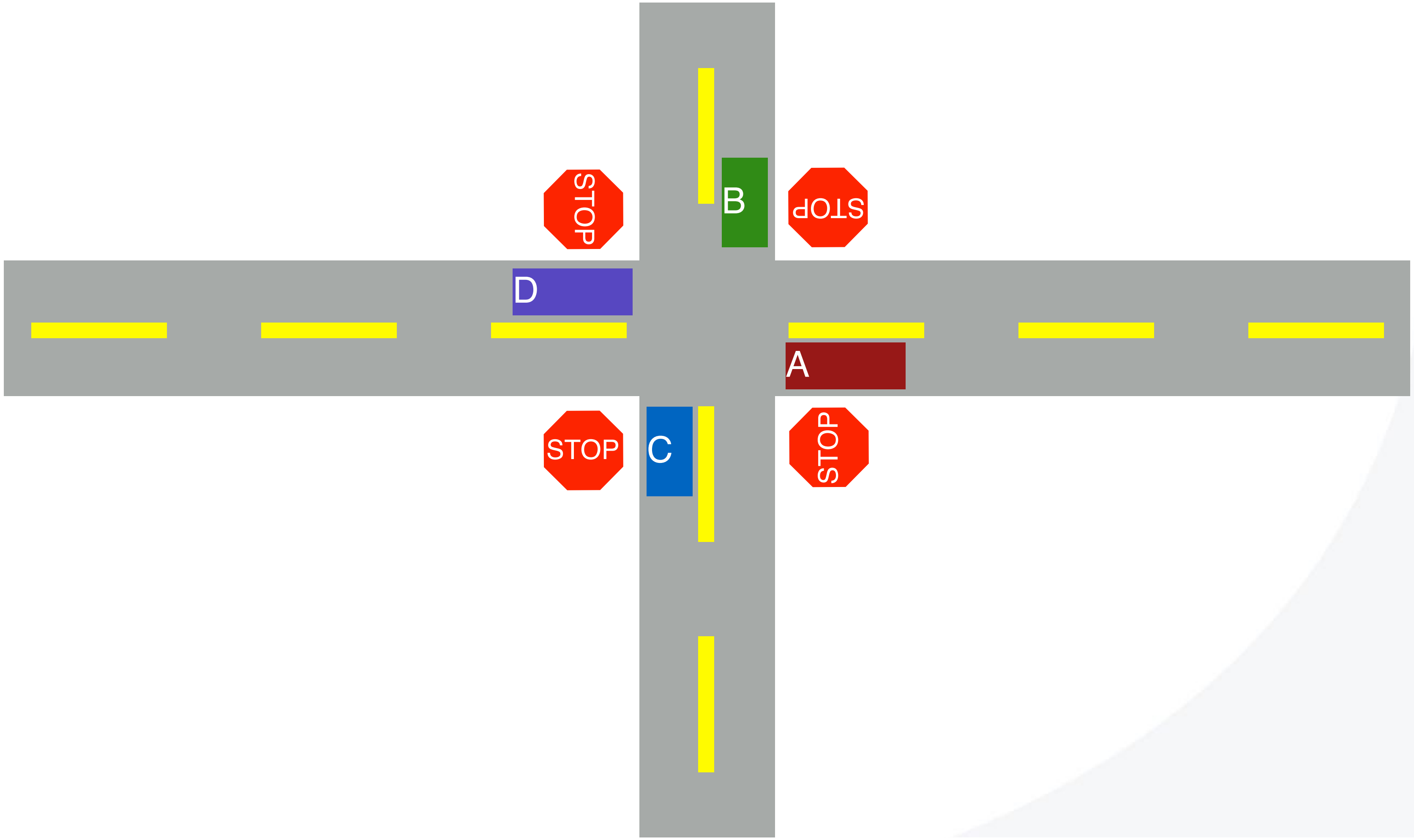
STOP
STOP
B
A
STOP
STOP



THE UNIVERSITY of ADELAIDE

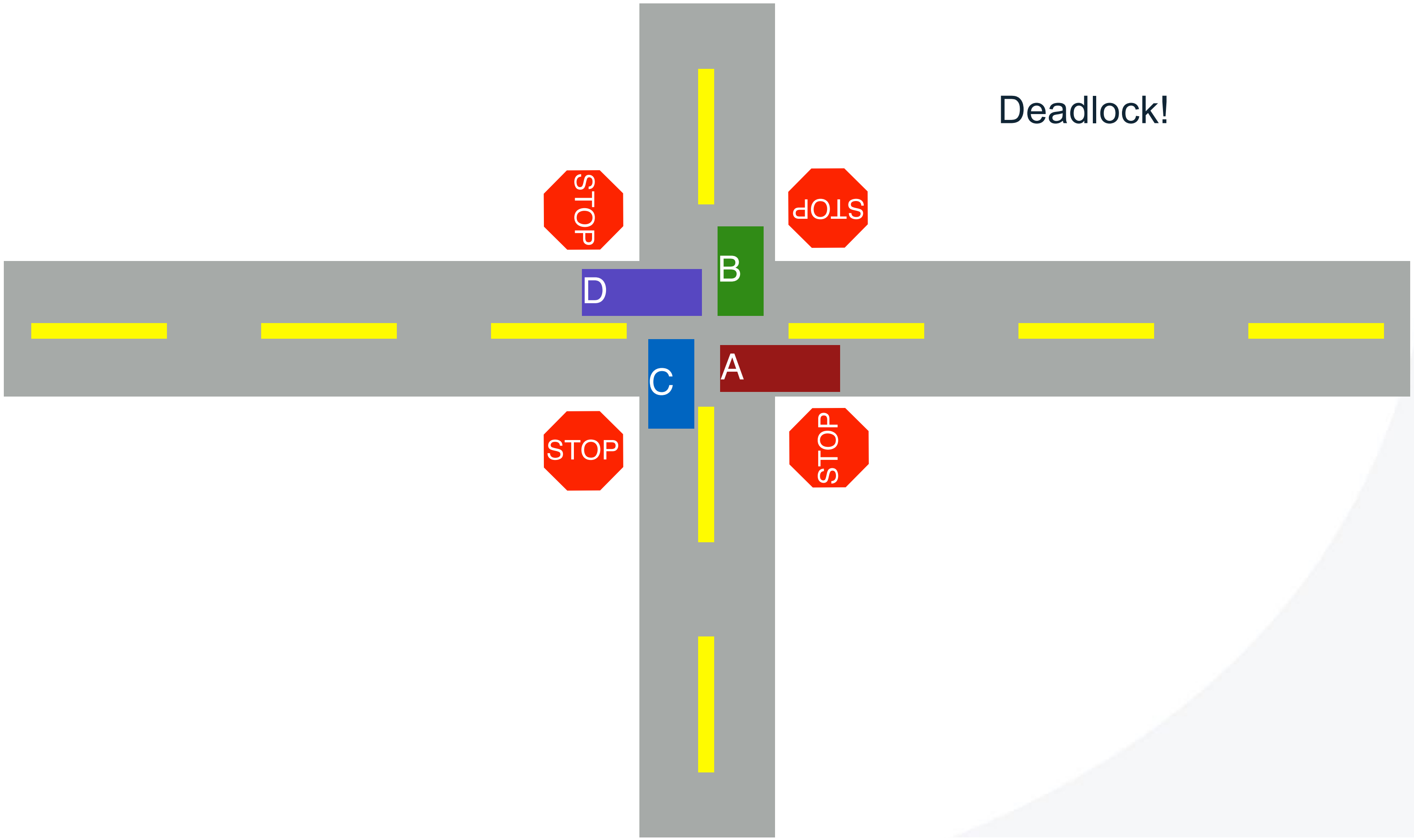


THE UNIVERSITY of ADELAIDE



STOP
B
STOP
D
A
STOP
C
STOP

Deadlock!



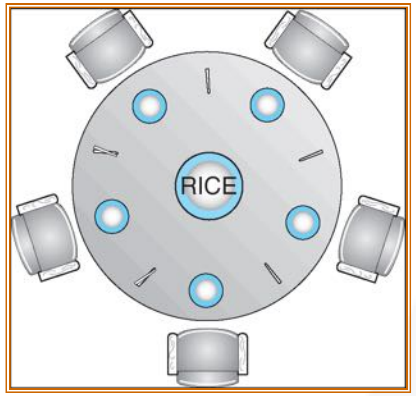
Dining Lawyers Problem

Five chopsticks/Five lawyers (really cheap restaurant)
Free-for all: Lawyer will grab any one they can
Need two chopsticks to eat

What if all grab at same time?
Deadlock!

How to fix deadlock?
Make one of them give up a chopstick (Hah!)
Eventually everyone will get chance to eat

How to prevent deadlock?
Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
Can we formalize this requirement somehow?



THE UNIVERSITY of ADELAIDE

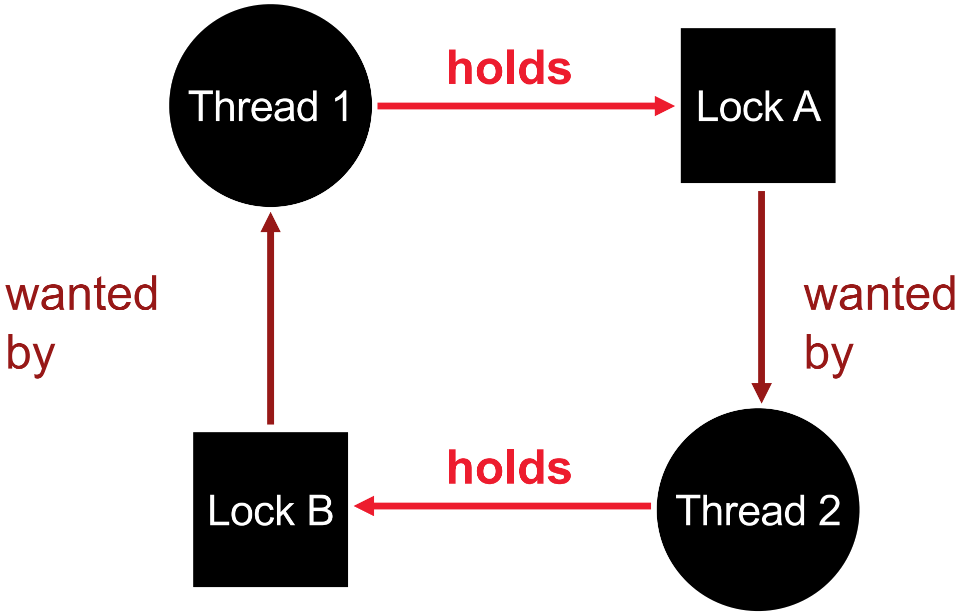
Example

```
Thread 1:

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

```
Thread 2:

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

Can deadlock happen with these two threads?

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

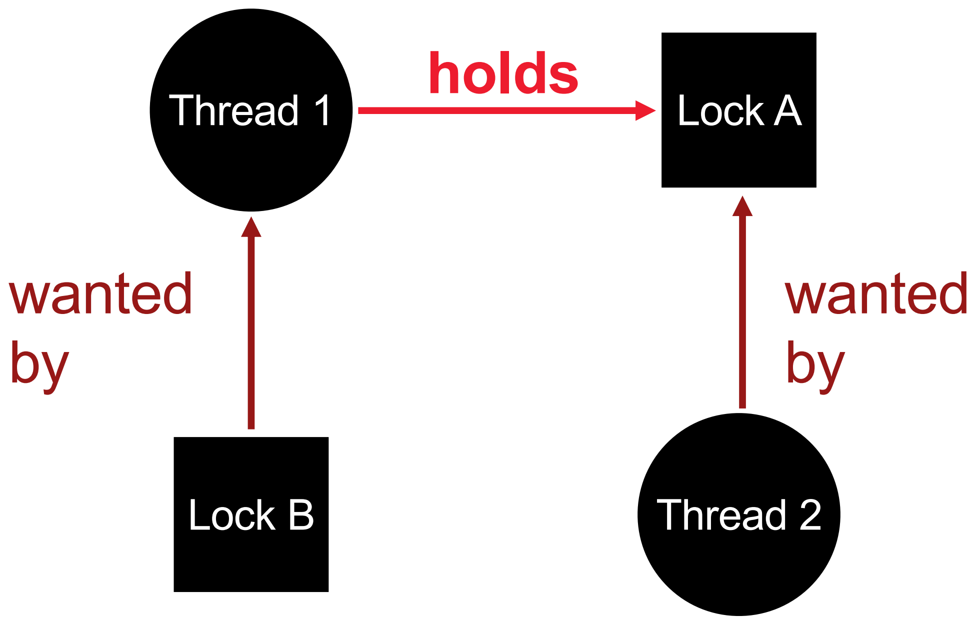
```
lock(&A);
lock(&B);
```

Thread 2

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

THE UNIVERSITY of ADELAIDE

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?

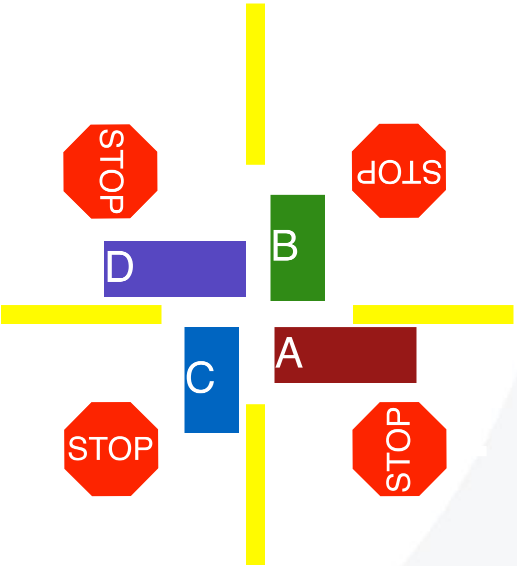
```
if (m1 > m2) {
    // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

Any other problems?

Code assumes m1 != m2 (not same lock)

Deadlock Theory

- **Deadlocks can only happen when these four conditions are met:**
  - mutual exclusion
  - hold-and-wait
  - no pre-emption
  - 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, old+amt);
}
```

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 when these four conditions are met:**
  - 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

THE UNIVERSITY of ADELAIDE

Deadlock Theory

- **Deadlocks can only happen when these four conditions are met:**
  - mutual exclusion
  - hold-and-wait
  - no pre-emption
  - 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 the 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**

THE UNIVERSITY of ADELAIDE

Circular Wait

- **Definition:**
  - There exists a circular chain of threads such that each thread holds a resource (e.g., lock) being requested by the 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 the system has distinct layers**

THE UNIVERSITY of ADELAIDE

Recall: non-deterministic deadlock

```
Thread 1:

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

```
Thread 2:

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

THE UNIVERSITY of ADELAIDE

Banker's Algorithm for Avoiding Deadlock

Toward right idea:

State maximum (max) resource needs in advance

Allow particular thread to proceed if:

(available resources - #requested) $\geq$ max remaining that might be needed by any thread

Banker's algorithm (less conservative):

Allocate resources dynamically

- Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

  ([Max$_{node}$]-[Alloc$_{node}$] <= [Avail]) for ([Request$_{node}$] <= [Avail])

  Grant request if result is deadlock free (conservative!)

THE UNIVERSITY of ADELAIDE

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
        done = true
        foreach node in UNFINISHED {
           if ([Request_node] <= [Avail]) {
              remove node from UNFINISHED
              [Avail] = [Avail] + [Alloc_node]
              done = false
           }
        }
     } until(done)
```

- Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
  ([Max$_{node}$]-[Alloc$_{node}$] <= [Avail]) for ([Request$_{node}$] <= [Avail])
  Grant request if result is deadlock free (conservative!)

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
      Add all nodes to UNFINISHED
      do {
         done = true
         foreach node in UNFINISHED {
            if ([Max_node]-[Alloc_node] <= [Avail]) {
               remove node from UNFINISHED
               [Avail] = [Avail] + [Alloc_node]
               done = false
            }
         }
      } until(done)
```

- Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

  $([Max_{node}]-[Alloc_{node}] <= [Avail])$ for $([Request_{node}] <= [Avail])$

  Grant request if result is deadlock free (conservative!)

THE UNIVERSITY of ADELAIDE

Banker’s Algorithm for Avoiding Deadlock

Toward right idea:

State maximum (max) resource needs in advance

Allow particular thread to proceed if:

> (available resources - #requested) $\geq$ max remaining that might be needed by any thread

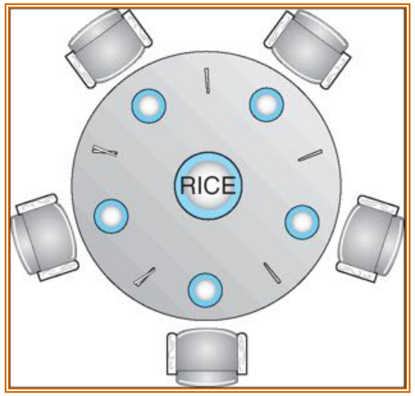
Banker’s algorithm (less conservative):

Allocate resources dynamically

- Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
  ([Max_{node}]-[$Alloc_{node}$] <= [Avail]) for ([$Request_{node}$] <= [Avail])
  Grant request if result is deadlock free (conservative!)
- Keeps system in a “SAFE” state: there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

THE UNIVERSITY of ADELAIDE

Banker's algorithm with dining lawyers



"Safe" (won't cause deadlock) if when try to grab chopstick either:

- Not last chopstick
- Is last chopstick but someone will have two afterwards

What if k-handed lawyers? Don't allow if:

- It's the last one, no one would have k
- It's 2nd to last, and no one would have k-1
- It's 3rd to last, and no one would have k-2
- …

THE UNIVERSITY of ADELAIDE

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