Operating Systems

Semaphores

Lecture Overview

- Semaphores

Last Week

Concurrency

- Locks
- Lock Implementation
  - Data Structures
- Condition Variables
- Pipes

Concurrency Objectives

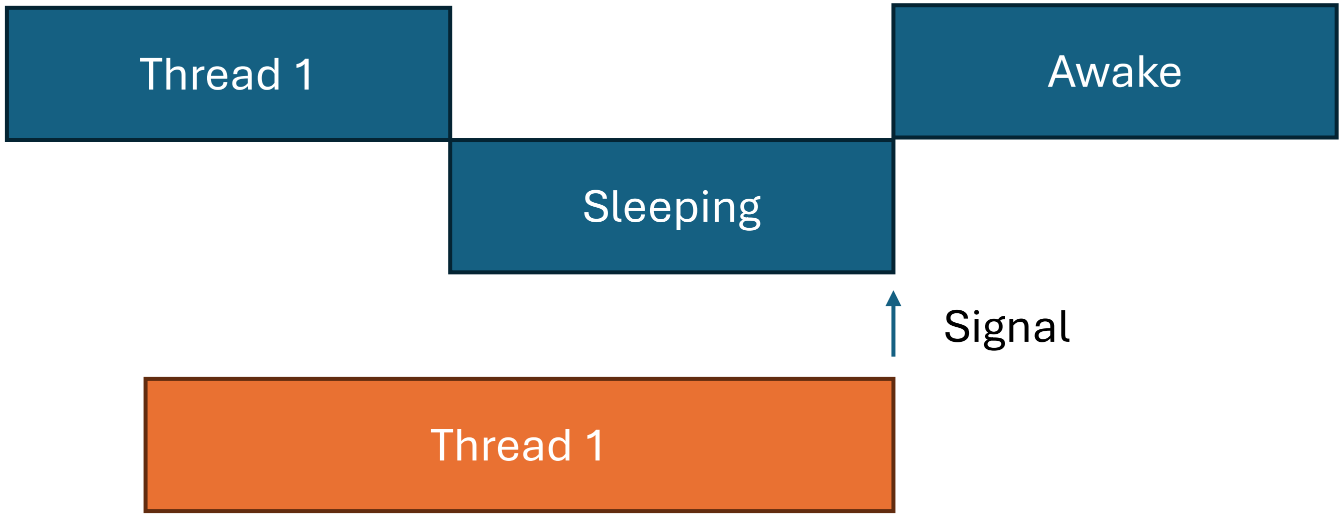
Main Objectives

- Mutual Exclusion (atomic logic)
  - Mutexes
- Ordering (A waits for B)
  - Condition Variables

Semaphores

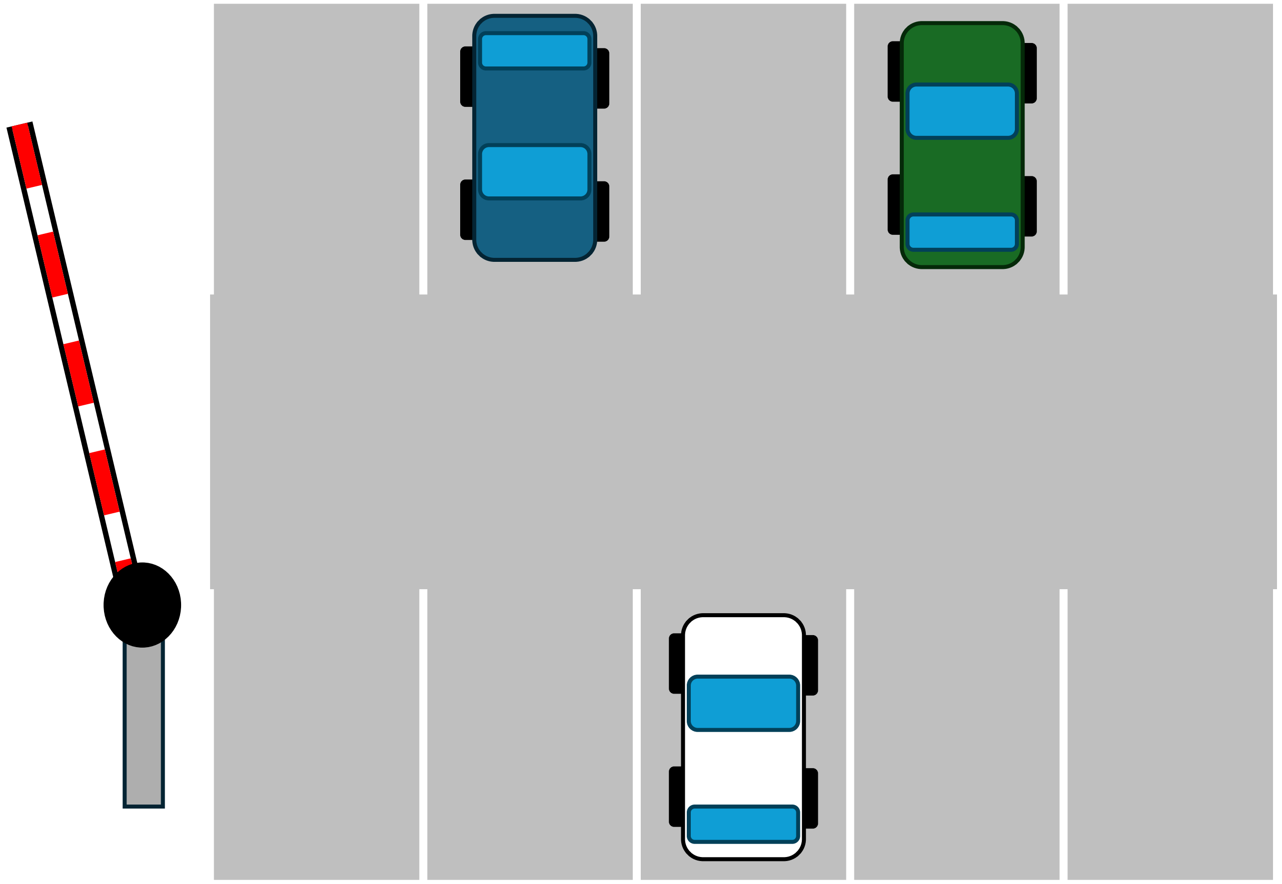
Condition Variables

A weakness



Condition Variables

Another weakness



Condition Variables

Condition Variables

- No ‘state’

Semaphores

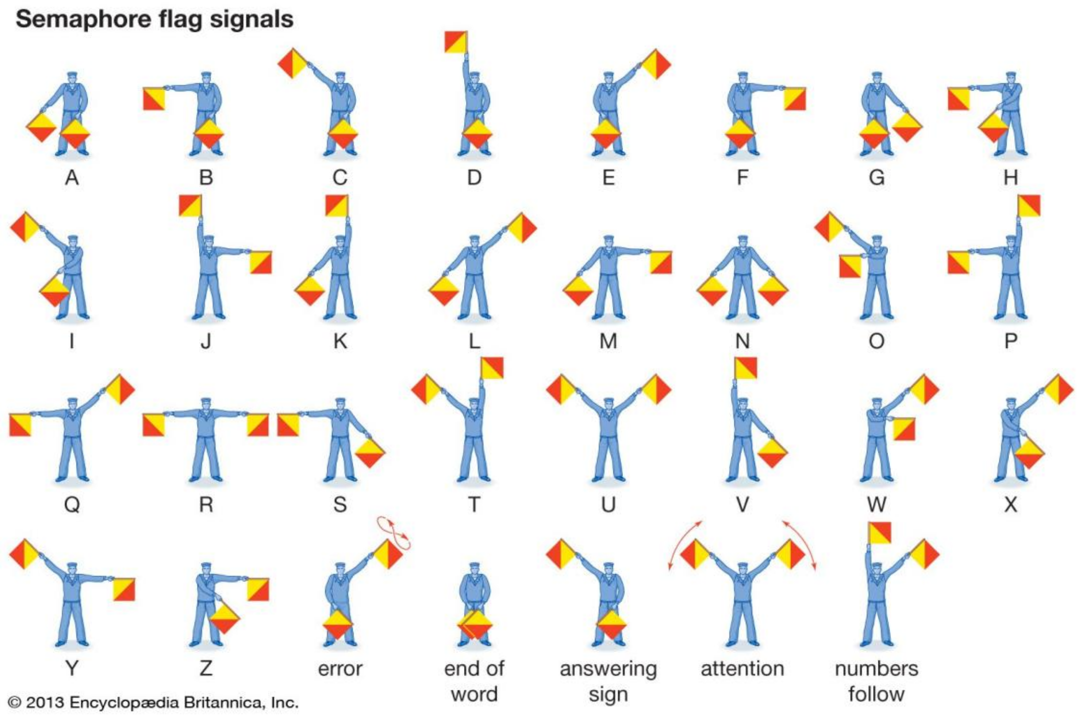
- Has a ‘state’
  - Non-negative **integer**

Semaphores

History

- Dijkstra (60s)

Etymology



Semaphores (In Practice)

wait():

Wait for the semaphore to become positive (>0) and then decrement.

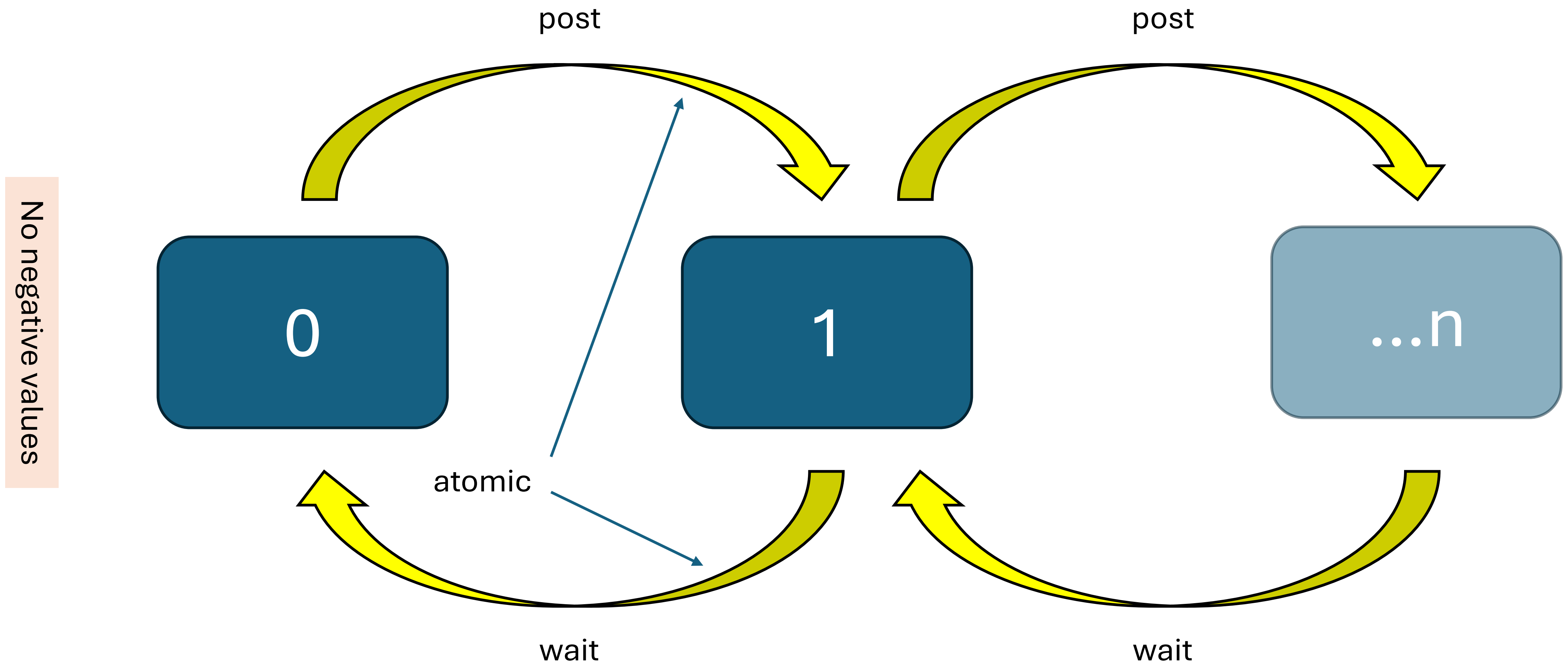
post():

Increment the semaphore by 1, waking up a waiting **wait()** (if any).

Sometimes referred to as V & P

verhogen: to increase
proberen: to probe

Semaphores



Semaphores (Code)

```
sem_t sem;

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

wait(), test(), P()

- Waits until the value of **sem** >0, then decrements **sem**

signal(), post(), V()

- Increases **sem** value, then wake a single waiter

CV vs Semaphore

CV

```
void thread_join() {
        mutex_lock (&lock);
        // do stuff
        done = true;
        cond_signal(&cond);
        mutex_unlock(&lock);
}
```

```
void thread_exit() {
        mutex_lock (&lock);
        if (!done)
                cond_wait(&cond, &lock);
        //do stuff
        mutex_unlock(&lock);
}
```

Semaphore

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

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

sem_t s;
semi_init(&s, ???);

Semaphores and Fairness…

Depends on implementation

- Some implementations provide queues
  - FIFO => Performance overhead

Equivalence

One of these things is just like the others...

So… which is best?

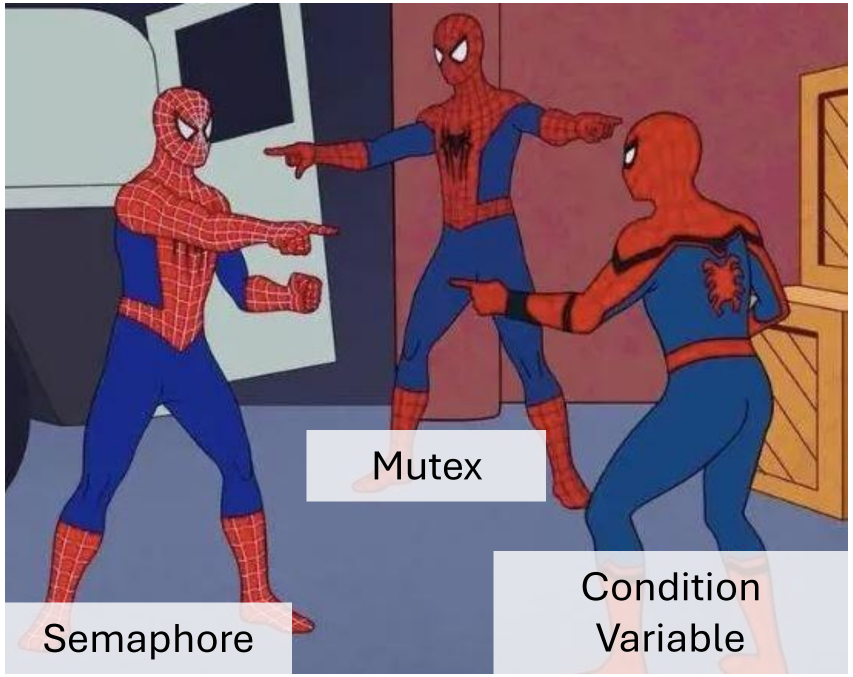
Equivalence

Differences:

- Implementations are different
- Some are more convenient

Similarity:

- One can make from the other…



Building a Lock from a 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, ??);
}
```

Oh wow…

Building a CV from a Semaphore

Can be done…

Much harder…

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        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);
}
```

```
void wait(sem_t *s) {
        ???
}

void post(sem_t *s) {
        ???
}
```

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        ???
} sem_t;

void sem_init(sem_t *s) {
        ???
}
```

```
void wait(sem_t *s) {
        ???
}

void post(sem_t *s) {
        ???
}
```

Building a Semaphore from a Lock and CV

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

void sem_init(sem_t *s) {
        ???
}
```

```
void wait(sem_t *s) {
        ???
}

void post(sem_t *s) {
        ???
}
```

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        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);
}
```

```
void wait(sem_t *s) {
        ???
}

void post(sem_t *s) {
        ???
}
```

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        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);
}
```

```
void wait(sem_t *s) {
        ???
}

void post(sem_t *s) {
        ???
}
```

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        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);
}
```

```
void wait(sem_t *s) {
        lock_acquire(&s->lock);
        // Stuff
        lock_release(&s->lock);
}

void post(sem_t *s) {
        lock_acquire(&s->lock);
        // Stuff
        lock_release(&s->lock);
}
```

Building a Semaphore from a Lock and CV

```c
typedef struct __sem_t {
        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);
}
```

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

void post(sem_t *s) {
        lock_acquire(&s->lock);
        // Stuff
        lock_release(&s->lock);
}
```

Building a Semaphore from a Lock and CV

```
typedef struct __sem_t {
        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);
}
```

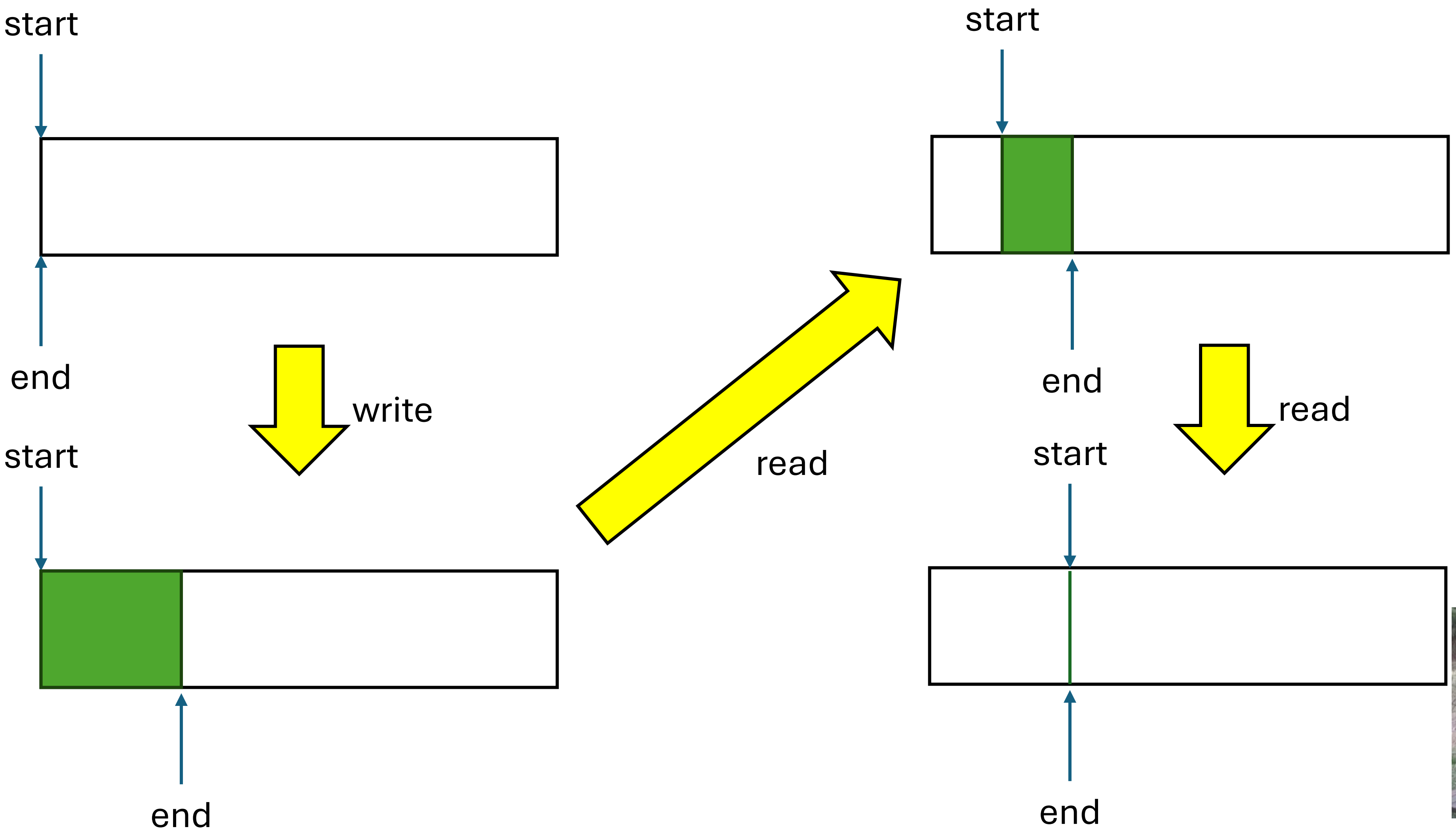
```
void wait(sem_t *s) {
        lock_acquire(&s->lock);
        while (s->value <= 0)
                cond_wait(&s->cond, &s->lock);
        s->value--;
        lock_release(&s->lock);
}

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

Building with Semaphores

Let’s make something

Revision: Buffer



Building a Buffer

How many semaphores?

How many constraints?

sem_t full;

sem_t empty;

sem_t mutex;

Bounded Buffer

```
sem_t fullSlots = 0;
sem_t emptySlots = bufferSize;
sem_t mutex = 1;
```

Bounded Buffer

```
sem_t fullSlots = 0;
sem_t emptySlots = bufferSize;
sem_t mutex = 1;

Producer(item) {
      sem_wait(&emptySlots);
      sem_wait(&mutex);
      Enqueue(item);
      sem_post(&mutex);
      sem_post(&fullSlots);
}
```

Avoids over-filling the queue

Avoids concurrency access issues

Signals the queue is not empty

Bounded Buffer

```
sem_t fullSlots = 0;
sem_t emptySlots = bufferSize;
sem_t mutex = 1;

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

Avoids over-emptying the queue

Avoids concurrency access issues

Signals the queue is not full

All Together (very simple implementation)

```
Producer(item) {
        sem_wait(&emptySlots);
        sem_wait(&mutex);
        Enqueue(item);
        sem_post(&mutex);
        sem_post(&fullSlots);
}
```

safe

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

Symmetry

Deadlock

```
Producer(item) {
      sem_wait(&mutex);
      sem_wait(&emptySlots);
      Enqueue(item);
      sem_post(&mutex);
      sem_post(&fullSlots);
}
```

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

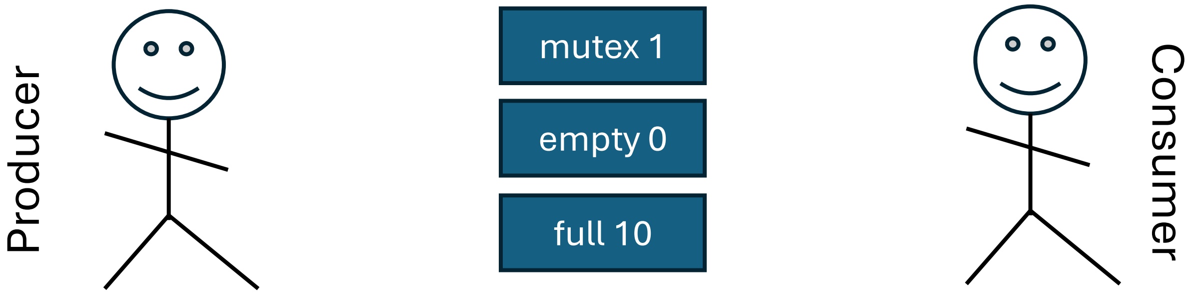
Order is important

Deadlock

```
Producer(item) {
        sem_wait(&mutex);
        sem_wait(&emptySlots);
        Enqueue(item);
        sem_post(&mutex);
        sem_post(&fullSlots);
}
```

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

Full Buffer



Semaphores: Summary

Summary:

- Equivalent to locks + condition variables
- Have a state (single integer)
- sem_wait()	wait until >0, then decrement (atomic)
- sem_post()	increment, then wake a single waiter (atomic)
- Convenient and ‘clean’ for:
  - producer/consumer
  - reader/writer

Timing

Dread...

Case Study: Therac-25

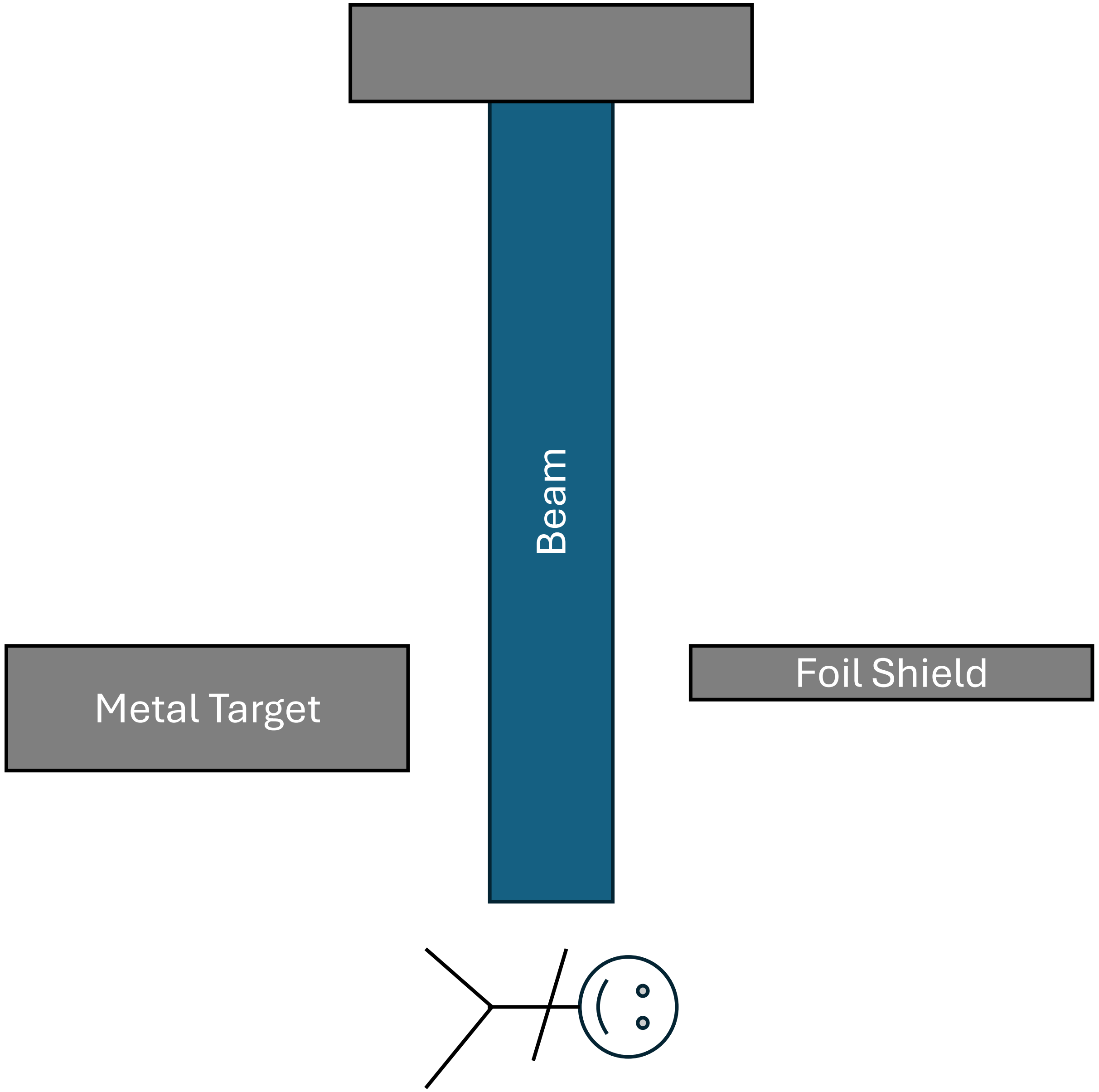
Computer controlled 'radiation theraphy machine'.

- High Dose, Wide Area
- Low Dose, Narrow Area

Case Study: Therac-25

Computer controlled ‘radiation theraphy machine’.

- High Dose, Wide Area
- Low Dose, Narrow Area



Problem

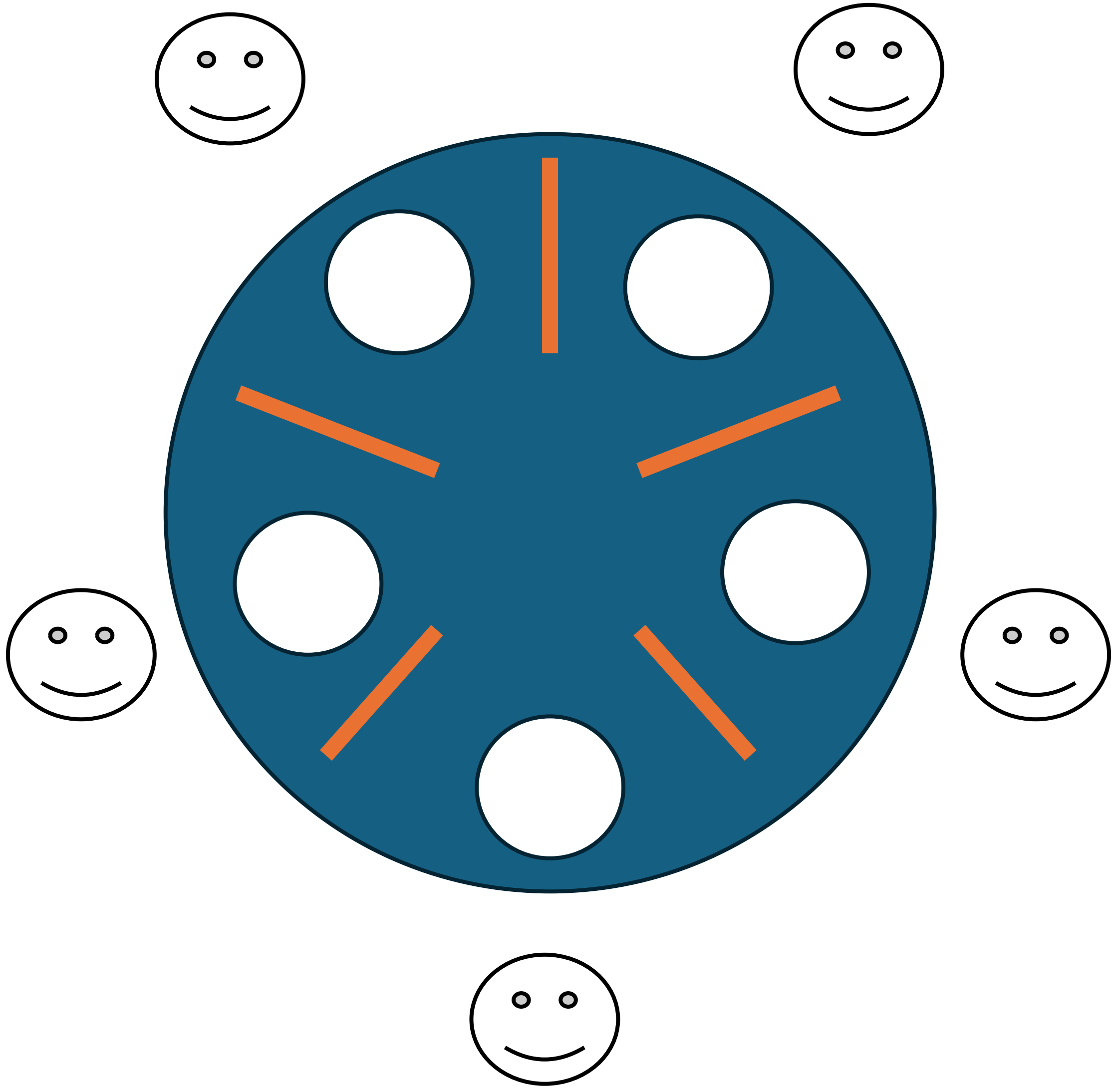
There are 5 people.

Each has a bowl of noodles.

To the left and right of each, is a single chopstick.

You can't talk.

You can only take one chopstick at a time.



Deadlock

Thread 1

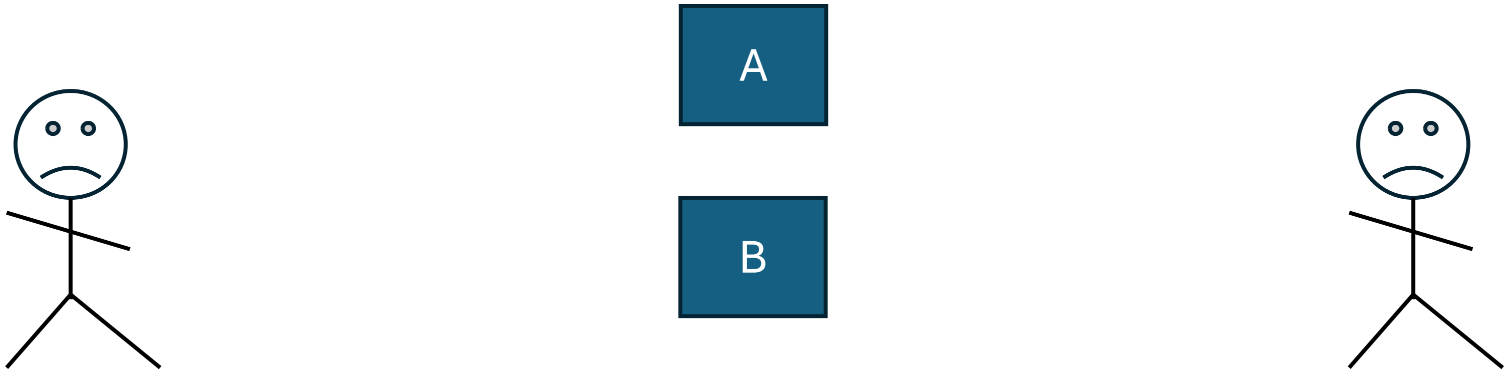
lock(&A)

lock(&B)

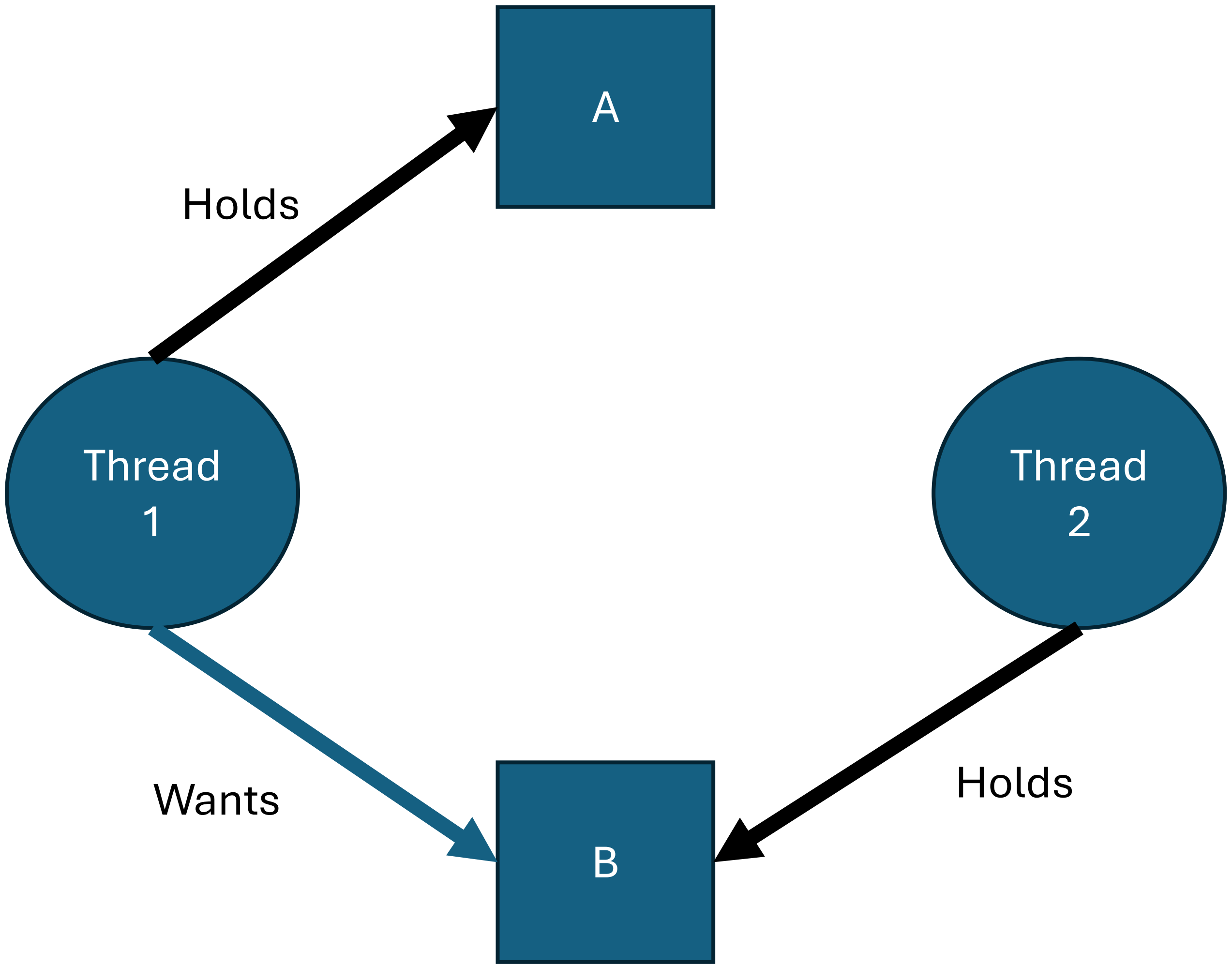
Thread 2

lock(&B)

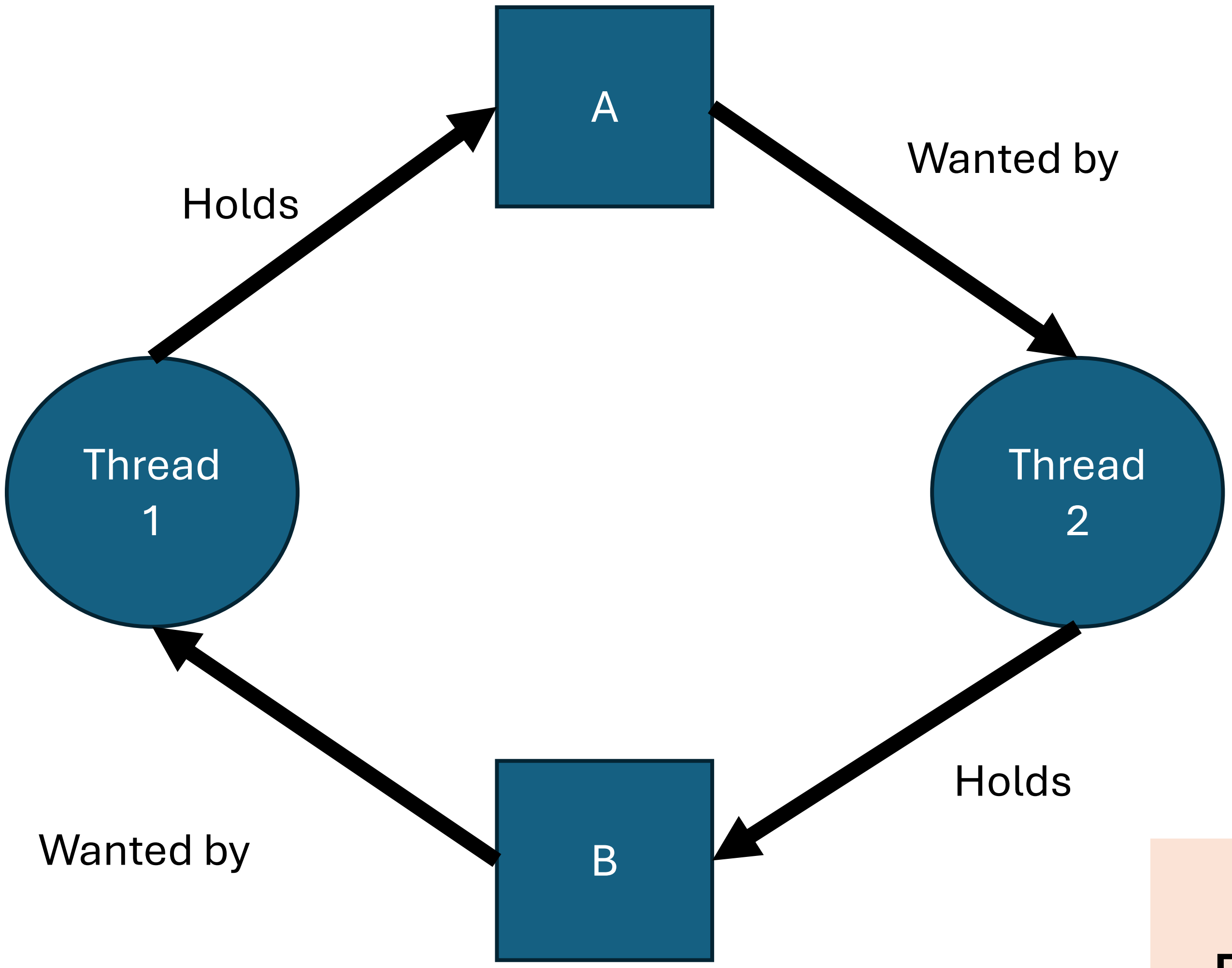
lock(&A)



How to think about this problem?



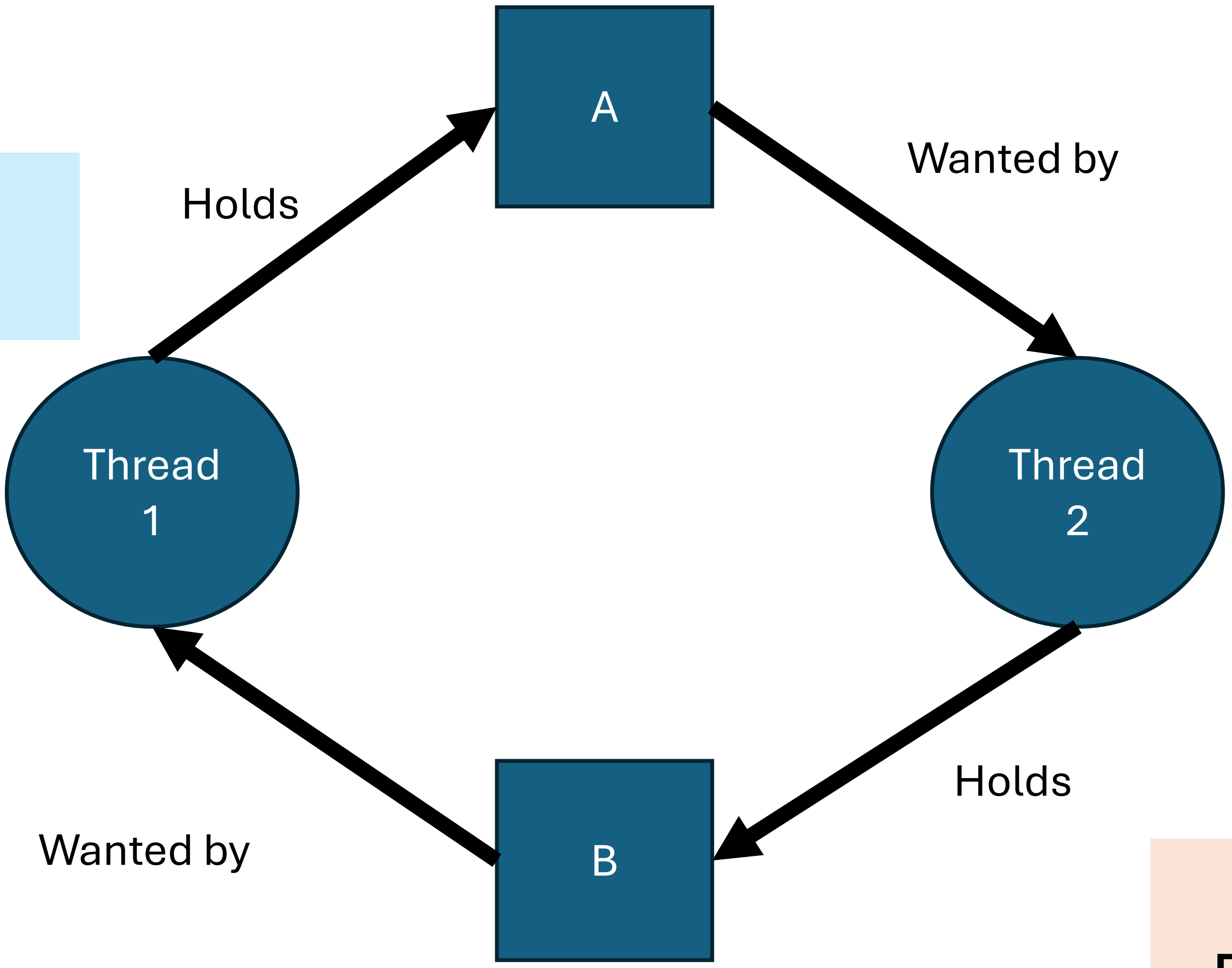
How to think about this problem?



Circular Dependency

How to think about this problem?

Thread 1
lock(&A)
lock(&B)

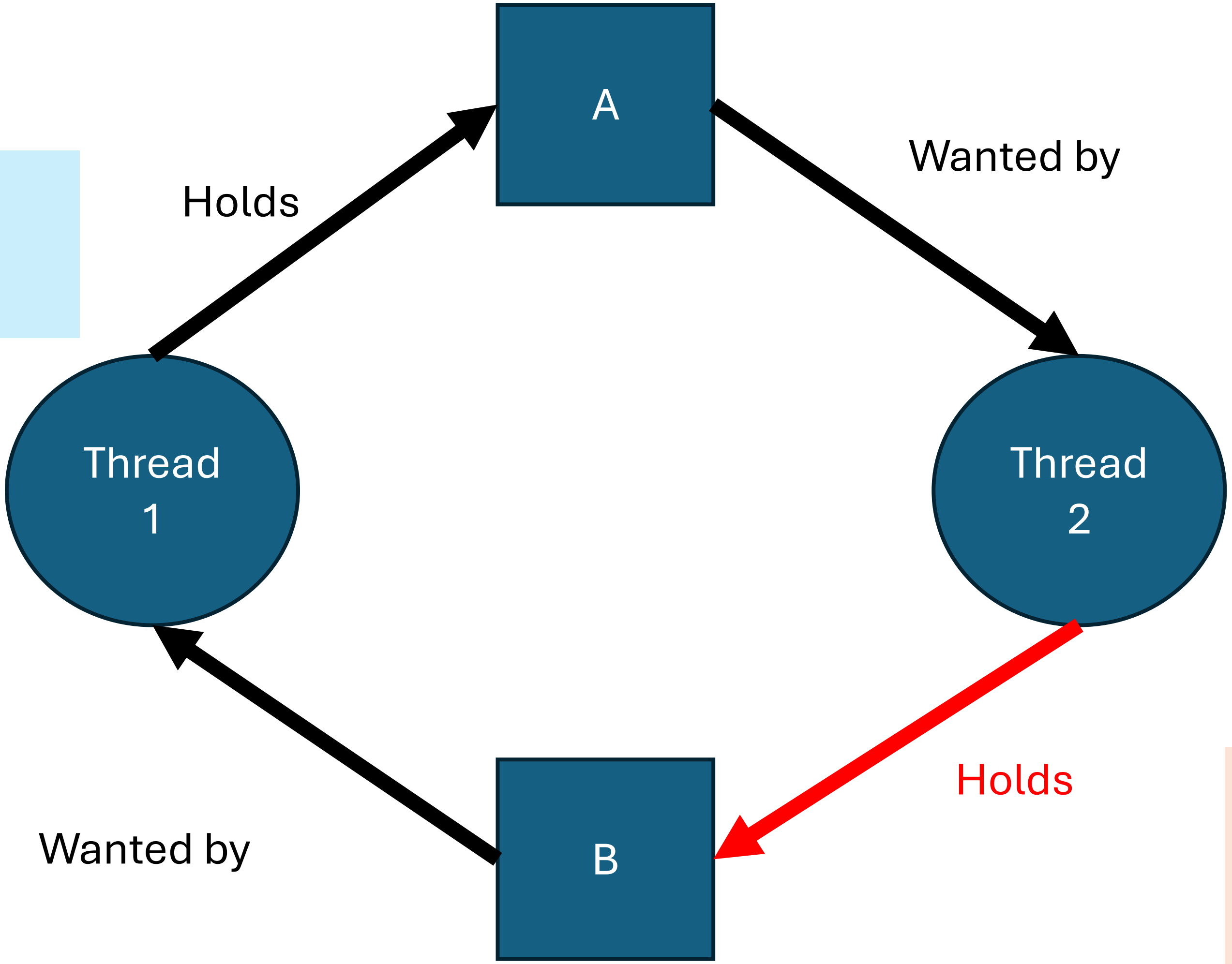


Thread 2
lock(&B)
lock(&A)

Circular Dependency

How to think about this problem?

Thread 1
lock(&A)
lock(&B)



Thread 2
lock(&A)
lock(&B)

Circular Dependency Defeated

Set Intersections: Encapsulation gone 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);
}
```

What is wrong here?

Are we controlling the lock order?

Thread 1

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

Thread 2

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

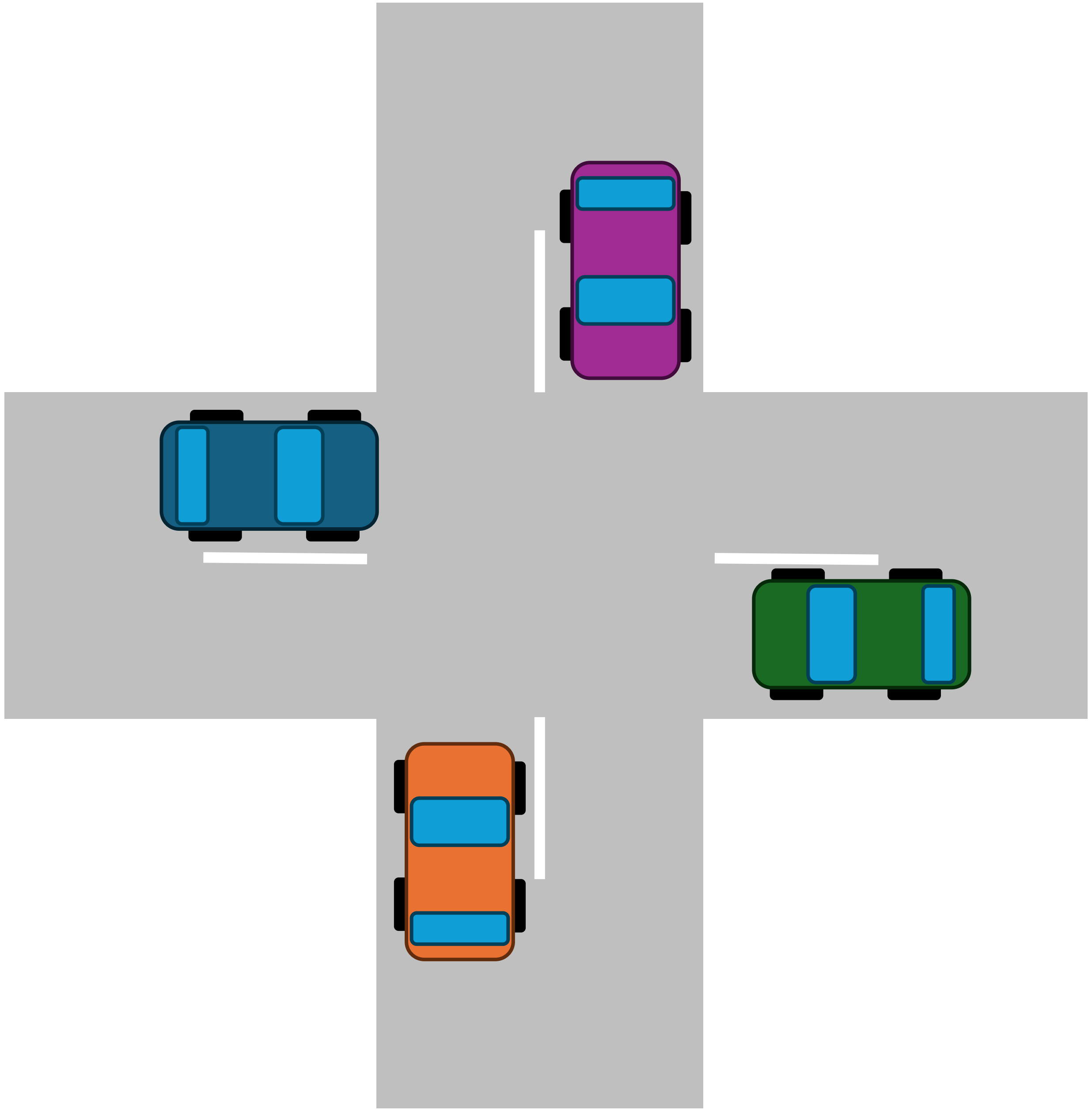
```
if (m1 > m2) {
        // Grab locks in high-to-low address order
        pthread_mutex_lock(m1);
        pthread_mutex_lock(m2);
} else {
        pthread_mutex_lock(m2);
        pthread_mutex_lock(m1);
}
```

Enforced Ordering

Deadlock Theory

Deadlock requires:

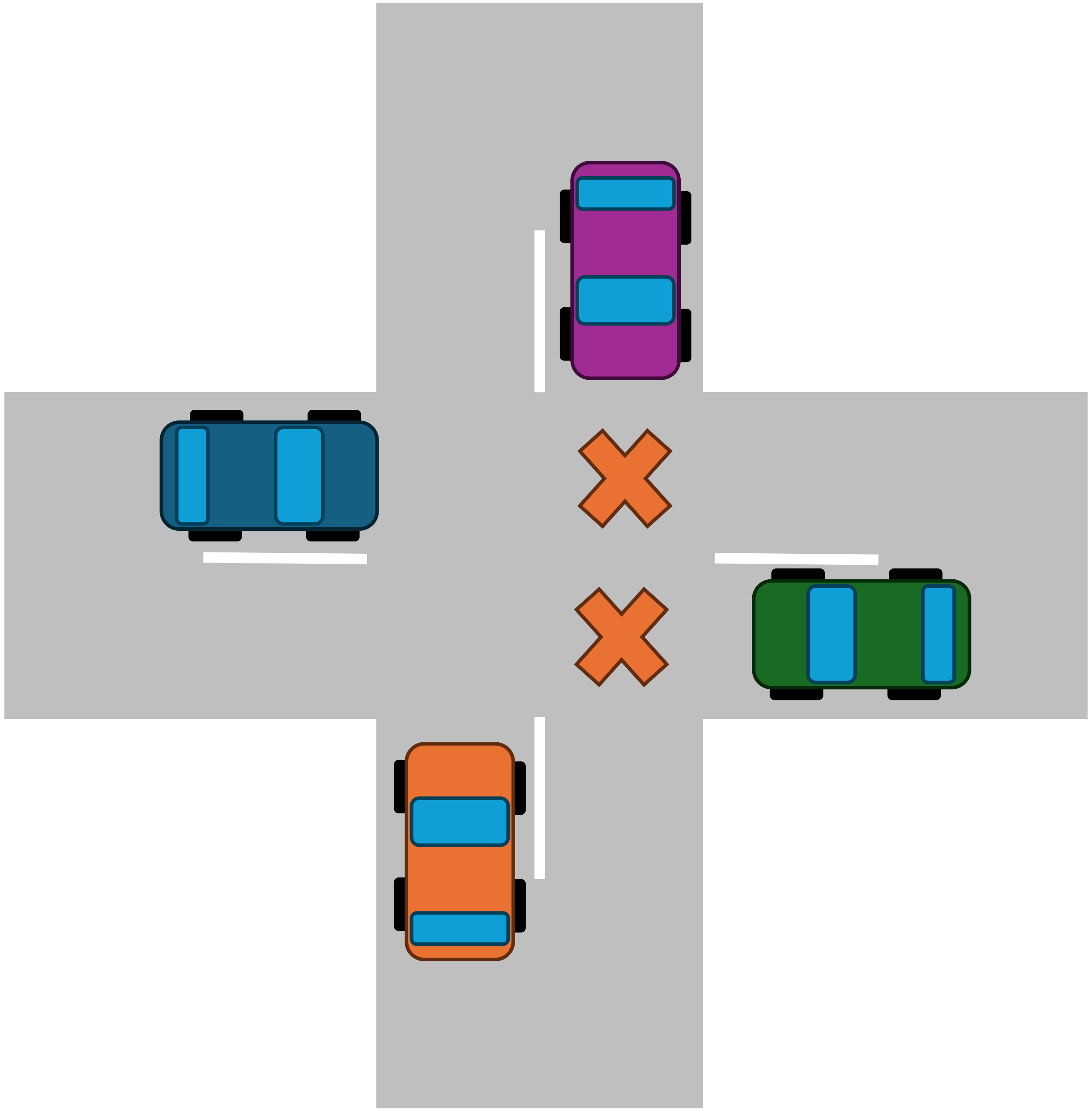
- Mutual Exclusion
- Hold and wait
- No Pre-emption
- Circular wait



Deadlock Theory

Deadlock requires:

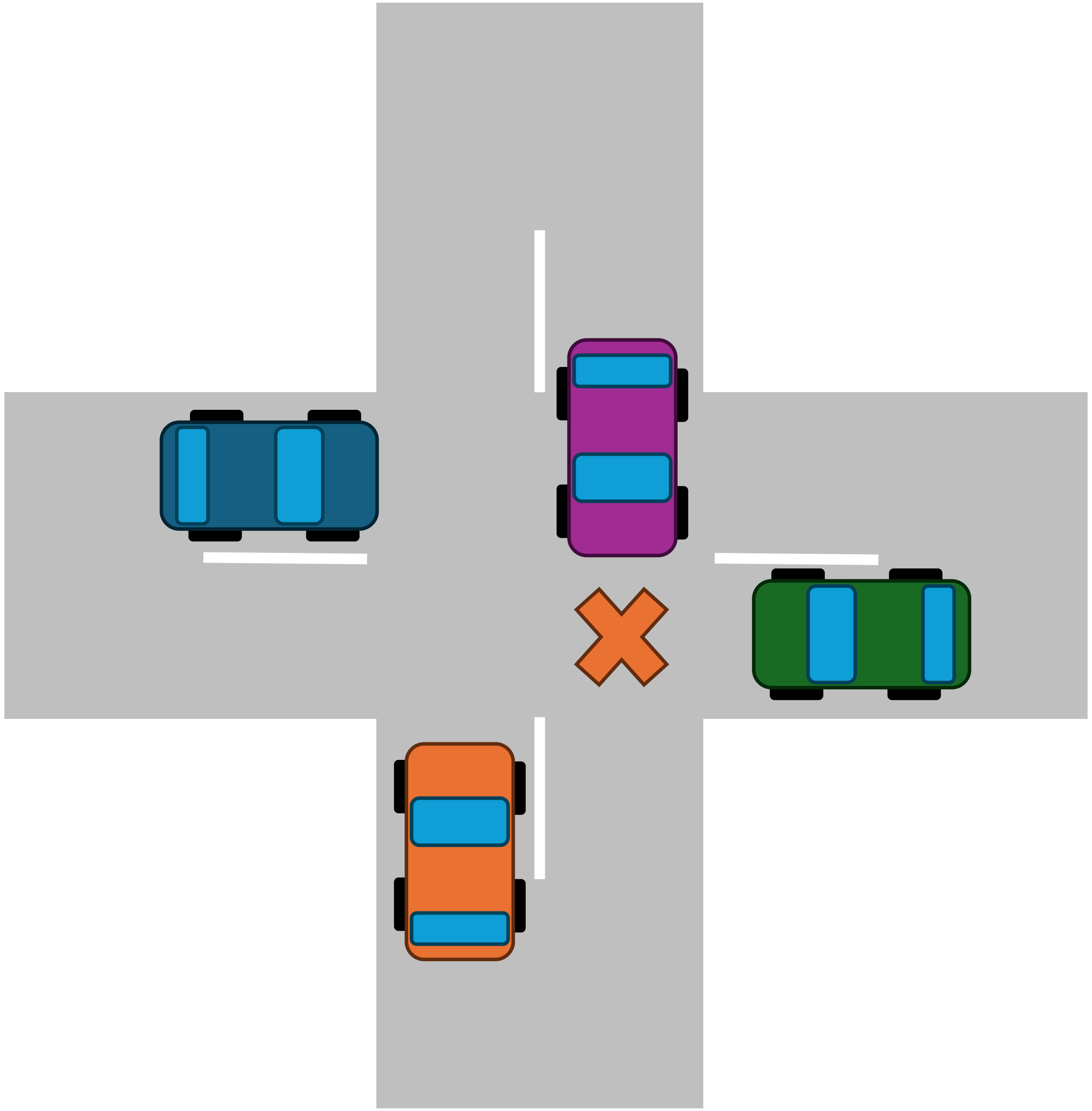
- **Mutual Exclusion**
- Hold and wait
- No Pre-emption
- Circular wait



Deadlock Theory

Deadlock requires:

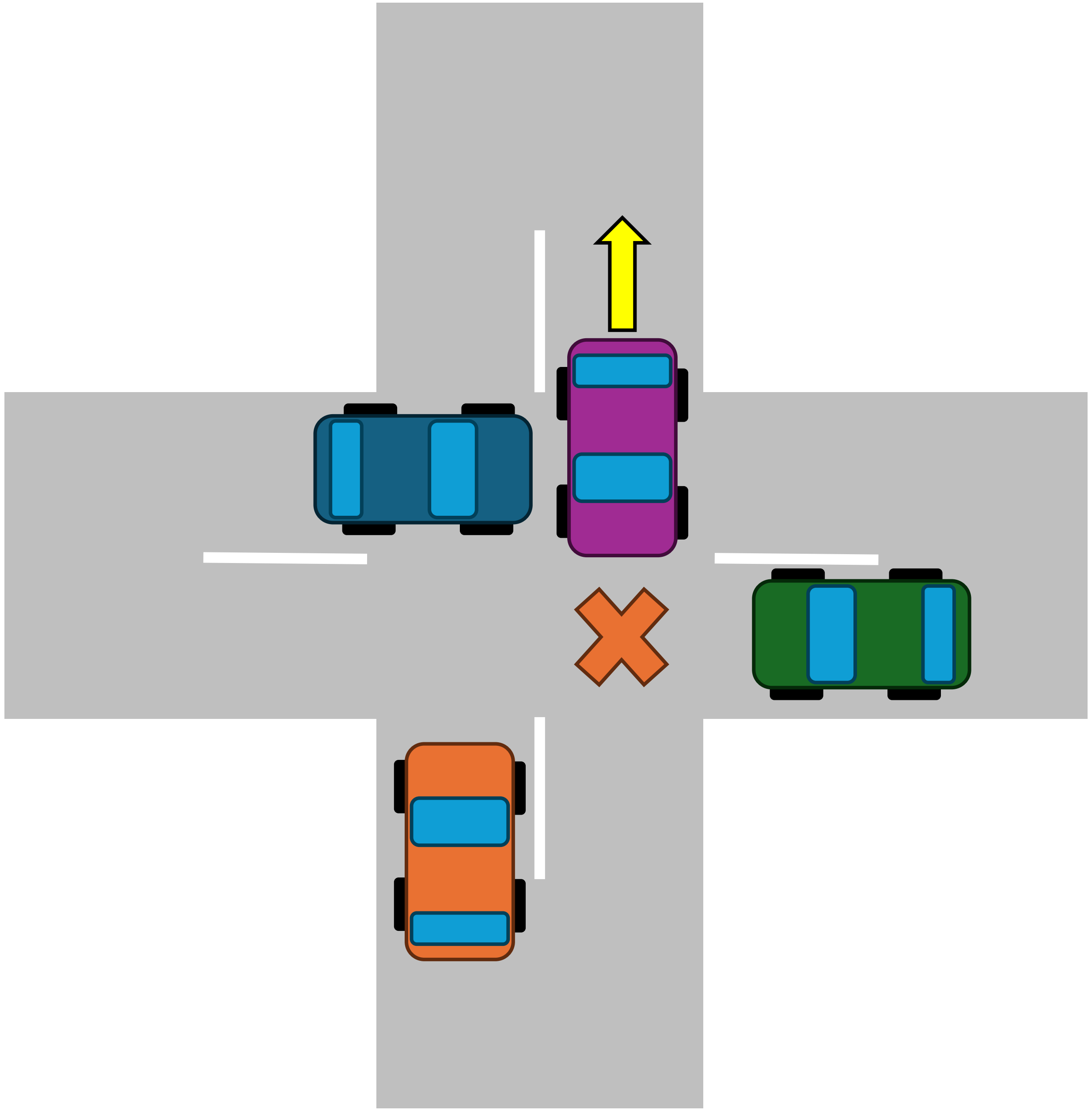
- Mutual Exclusion
- **Hold and wait**
- No Pre-emption
- Circular wait



Deadlock Theory

Deadlock requires:

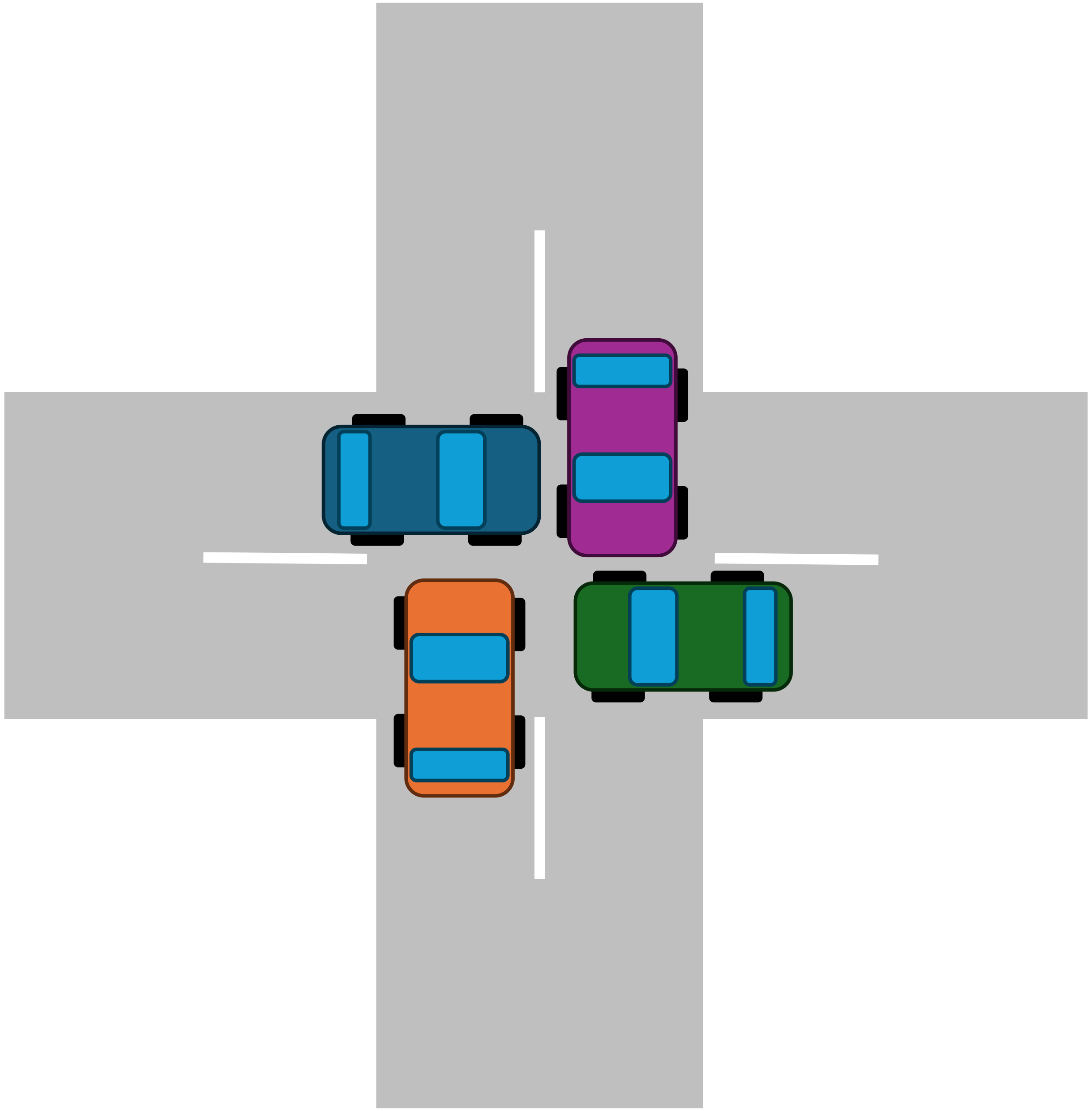
- Mutual Exclusion
- Hold and wait
- **No Pre-emption**
- Circular wait



Deadlock Theory

Deadlock requires:

- Mutual Exclusion
- Hold and wait
- No Pre-emption
- **Circular wait**



Eliminating Mutual Exclusion

Mutual Exclusion
Hold and wait
No Pre-emption
Circular wait

Eliminate the lock?

```
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);
}
```

Use the primitives and ignore locks entirely!

Eliminating Mutual Exclusion

Mutual Exclusion
Hold and wait
No Pre-emption
Circular wait

Eliminate the lock?

```
void insert(int * val) {
        node_t * n =
                malloc(sizeof(*n));
        n->val = val;
        lock(&m);
        n->next = head;
        head = n;
        unlock(&m);
}
```

Linked List

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

Use the primitives and ignore locks entirely!

Eliminate Hold and Wait?

Mutual Exclusion
Hold and wait
No Pre-emption
Circular wait

Meta-Lock

```
lock(&meta);
lock(&a);
lock(&b);
…

unlock(&meta);
```

Problems?

- Must know which locks are needed, ahead of time
- Must be conservative (acquiring everything)
- How is this different from 1 lock?
- Bottleneck…

Adding Pre-emption

Mutual Exclusion
Hold and wait
No Pre-emption
Circular wait

Give up!

top:

lock(a);

if (tryLock(b) == -1) {

unlock(A);

goto **top**;

}

Problems?

- Busy-waiting
- No guarantee of progress

Livelock:

- No process makes process, but the state of involved processes is in flux

Eliminate Circular Wait

Mutual Exclusion
Hold and wait
No Pre-emption
Circular wait

Solve the problem

- Decide ordering of locks
- Ensure no A->B + B->A etc.
- Document it and write the code

Advantages

- Works well with systems with distinct layers
- Useful within the OS kernel
- Enforces a strict lock hierarchy

Banker's Algorithm (Dijkstra)

Core Idea:

- Before acquiring a resource, will the system be in a 'safe state'.
  - Safe: Exists one sequence of processes such that all processes can finish with the currently available resources

What do we need to know?

- Maximum resources needed
  - [MAX] nxm
- How much do you have
  - [ALLOCATION] nxm
- How much is available
  - [AVAILABLE] m

Banker’s Algorithm: Example

	Allocation				**Max**				**Available**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0												
P_1												
P_2												
P_3												
P_4												

Banker's Algorithm: Example

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2								
P_1	1	0	0	0								
P_2	1	3	5	4								
P_3	0	6	3	2								
P_4	0	0	1	4								

Banker's Algorithm: Example

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Banker's Algorithm: Example

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Are we safe?

Banker's Algorithm: Example

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2	Are we safe?			
P_4	0	0	1	4	0	6	5	6				
Tot	3	14	12	12								

Banker's Algorithm: Example

<table>
<tr><th></th><th colspan="4">Allocation</th><th colspan="4">Max</th><th colspan="4">Available</th><th colspan="4">Need</th></tr>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>P_0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td><td>0</td><td>1</td><td>2</td><td>1</td><td>5</td><td>2</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P_1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>7</td><td>5</td><td>0</td><td colspan="4" rowspan="5">MAX
- ALLOCATION</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P_2</td><td>1</td><td>3</td><td>5</td><td>4</td><td>2</td><td>3</td><td>5</td><td>6</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P_3</td><td>0</td><td>6</td><td>3</td><td>2</td><td>0</td><td>6</td><td>5</td><td>2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P_4</td><td>0</td><td>0</td><td>1</td><td>4</td><td>0</td><td>6</td><td>5</td><td>6</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Tot</td><td>3</td><td>14</td><td>12</td><td>12</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	**0**	**0**	**1**	**2**	**0**	**0**	**1**	**2**	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0								
P_2	1	3	5	4	2	3	5	6								
P_3	0	6	3	2	0	6	5	2								
P_4	0	0	1	4	0	6	5	6								
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	**1**	**0**	**0**	**0**	**1**	**7**	**5**	**0**					0	7	5	0
P_2	1	3	5	4	2	3	5	6								
P_3	0	6	3	2	0	6	5	2								
P_4	0	0	1	4	0	6	5	6								
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	**1**	**0**	**0**	**0**	**1**	**7**	**5**	**0**					0	7	5	0
P_2	1	3	5	4	2	3	5	6								
P_3	0	6	3	2	0	6	5	2								
P_4	0	0	1	4	0	6	5	6								
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0					0	7	5	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				**Max**				**Available**				**Need**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	Can we using available satisfy need?				0	7	5	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker’s Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	**0**	**0**	**0**	**0**
P_1	1	0	0	0	1	7	5	0					0	7	5	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2	Can we using available satisfy need?				0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	**0**	**0**	**0**	**0**
P_1	1	0	0	0	1	7	5	0					0	7	5	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2	Yes!				0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	0	0	0	0
P_1	1	0	0	0	1	7	5	0	Can we using available satisfy need?				**0**	**7**	**5**	**0**
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				**Max**				**Available**				**Need**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	0	0	0	0
P_1	1	0	0	0	1	7	5	0					**0**	**7**	**5**	**0**
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2	No				0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	0	0	0	0
P_1	1	0	0	0	1	7	5	0					0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6					1	0	0	**2**
P_3	0	6	3	2	0	6	5	2	No				0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	0	0	0	0
P_1	1	0	0	0	1	7	5	0					0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6					1	0	0	**2**
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

P_0

P_3

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	**1**	**5**	**2**	**0**	0	0	0	0
P_1	1	0	0	0	1	7	5	0					0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6					1	0	0	**2**
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

P_0 →

0	0	1	2

1	5	3	2

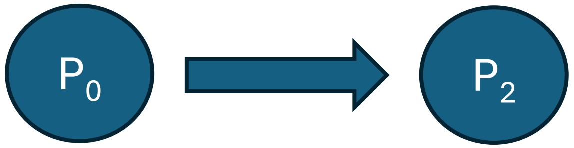
Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	**1**	**5**	**3**	**2**	0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6					1	0	0	**2**
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow$

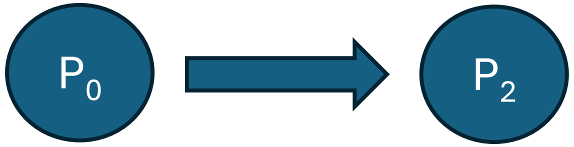
Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	**1**	**5**	**3**	**2**	0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6					**1**	**0**	**0**	**2**
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												



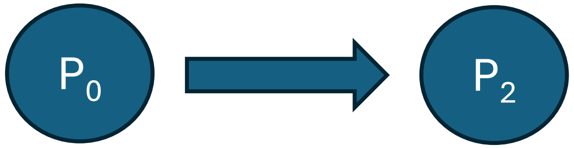
Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	**7**	**5**	0
P_2	**1**	**3**	**5**	**4**	2	3	5	6	**2**	**8**	**8**	**6**	1	0	0	2
P_3	0	6	3	2	0	6	5	2					0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												



Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	**2**	**8**	**8**	**6**	1	0	0	2
P_3	**0**	**6**	**3**	**2**	0	6	5	2					**0**	**0**	**2**	**0**
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												



Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6	**2**	**8**	**8**	**6**	1	0	0	2
P_3	**0**	**6**	**3**	**2**	0	6	5	2					**0**	**0**	**2**	**0**
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3$

Banker’s Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	**2**	**8**	**8**	**6**	1	0	0	2
P_3	**0**	**6**	**3**	**2**	0	6	5	2	**2**	**14**	**11**	**8**	**0**	**0**	**2**	**0**
P_4	0	0	1	4	0	6	5	6					0	6	4	2
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3$

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	**7**	**5**	0
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	**2**	**14**	**11**	**8**	0	0	2	0
P_4	0	0	1	4	0	6	5	6					0	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3$

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	**2**	**14**	**11**	**8**	0	0	2	0
P_4	0	0	1	4	0	6	5	6					**0**	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3$

Banker’s Algorithm: Example

	Allocation				**Max**				**Available**				**Need**			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	**2**	**14**	**11**	**8**	0	0	2	0
P_4	**0**	**0**	**1**	**4**	0	6	5	6					**0**	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3$

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	**2**	**14**	**11**	**8**	0	0	2	0
P_4	**0**	**0**	**1**	**4**	0	6	5	6					**0**	**6**	**4**	**2**
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

Banker's Algorithm: Example

	Allocation				Max				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0	1	5	3	2	0	7	5	0
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	**2**	**14**	**11**	**8**	0	0	2	0
P_4	**0**	**0**	**1**	**4**	0	6	5	6	**2**	**14**	**12**	**12**	0	6	4	2
Tot	**3**	**14**	**12**	**12**												

$P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

Banker’s Algorithm: Example

	Allocation				Max				Available				Need			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	**1**	**0**	**0**	**0**	1	7	5	0	1	5	3	2	**0**	**7**	**5**	**0**
P_2	1	3	5	4	2	3	5	6	2	8	8	6	1	0	0	2
P_3	0	6	3	2	0	6	5	2	2	14	11	8	0	0	2	0
P_4	0	0	1	4	0	6	5	6	**2**	**14**	**12**	**12**	0	6	4	2
Tot	**3**	**14**	**12**	**12**					**3**	**14**	**12**	**12**				

$P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1$

Banker's Algorithm: Summary

Summary

1. Set processes not ready.
2. Find processes needs! NEED = (MAX – ALLOCATION)
3. Find runnable processes! NEED ≤ AVAILABLE?
4. Run runnable processes! AVAILABLE += ALLOCATION_i
5. Repeat from #3

Also known as the… duh algorithm?

Summary

- Semaphores
- Deadlock
- Livelock
- Banker’s Algorithm

Questions?