Operating Systems

Concurrency

Lecture Overview

- Locks
- Lock Implementation
  - Data Structures
- Condition Variables

Last Week

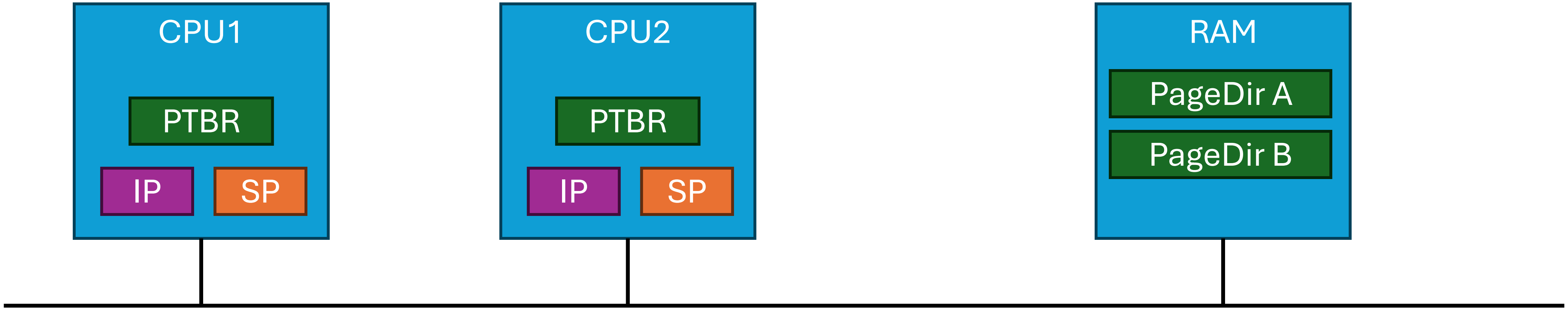
Memory Virtualisation

- Page Swapping

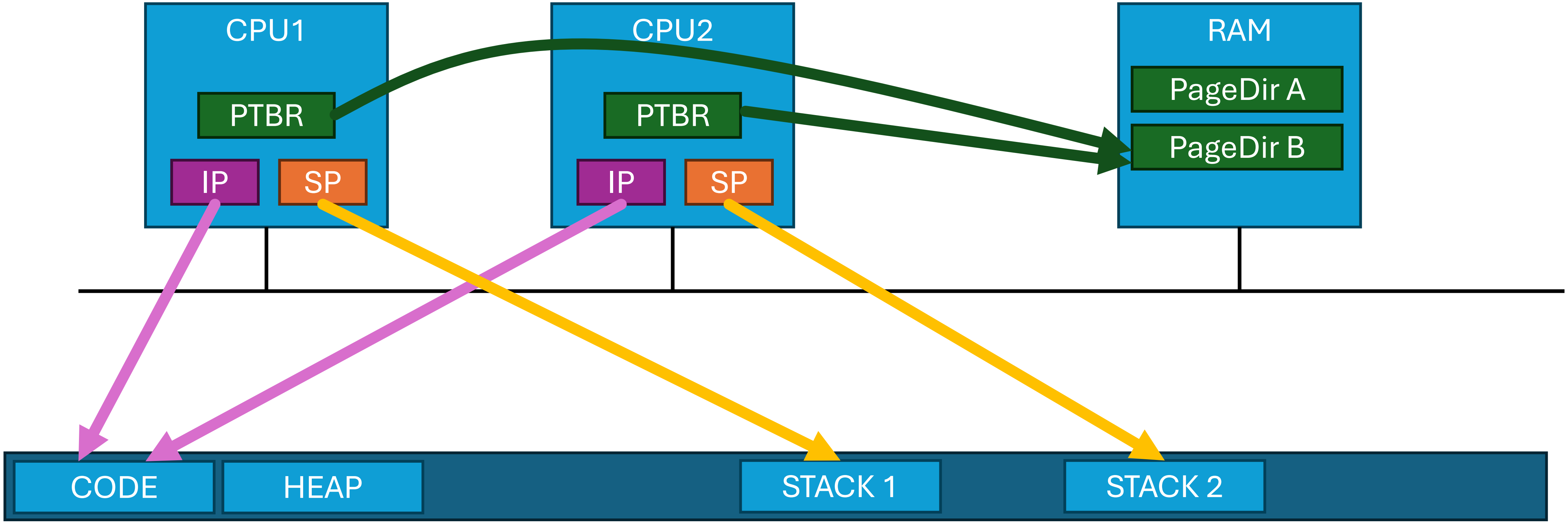
Atomicity and Concurrency

Things happening at the same time or not at the same time

Multiple CPUs



Multiple CPUs



One Process, Two Threads

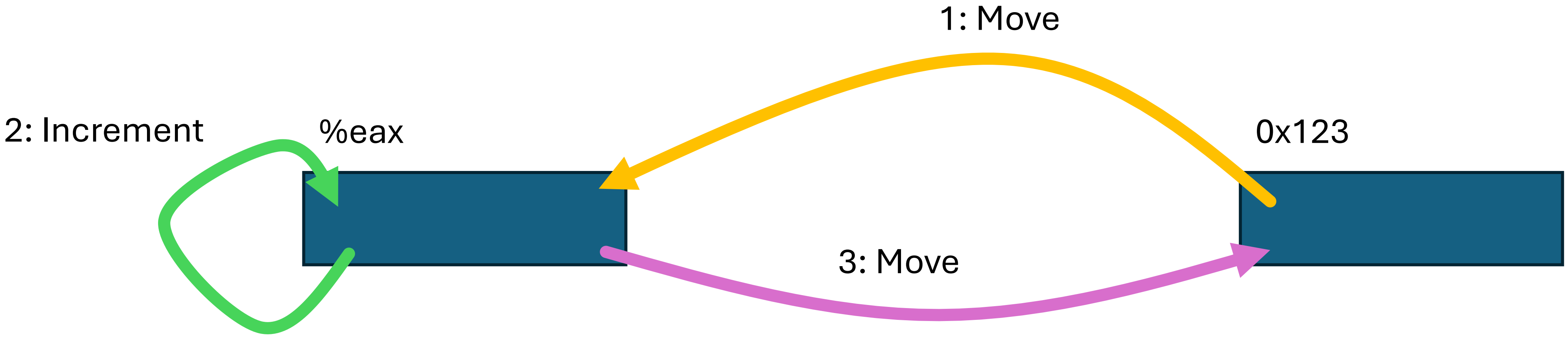
The problem of Atomicity

Consider:

```
incrementBalance()
{
        balance = balance + 1;
}
```

What this translates to:

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```



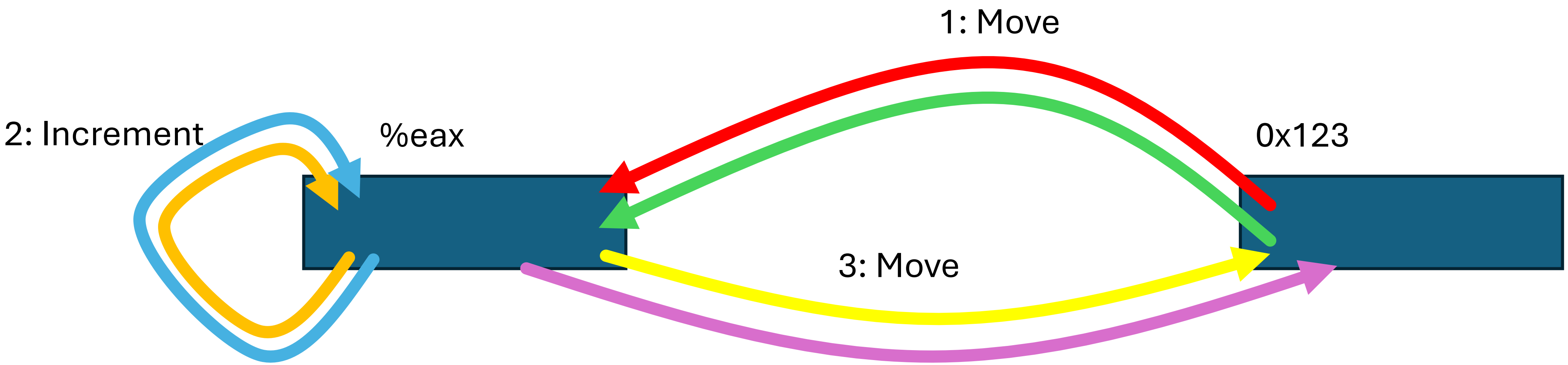
The problem of Atomicity

Consider:

```
incrementBalance()
{
        balance = balance + 1;
}
```

What this translates to:

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```



Activity

You have two threads.

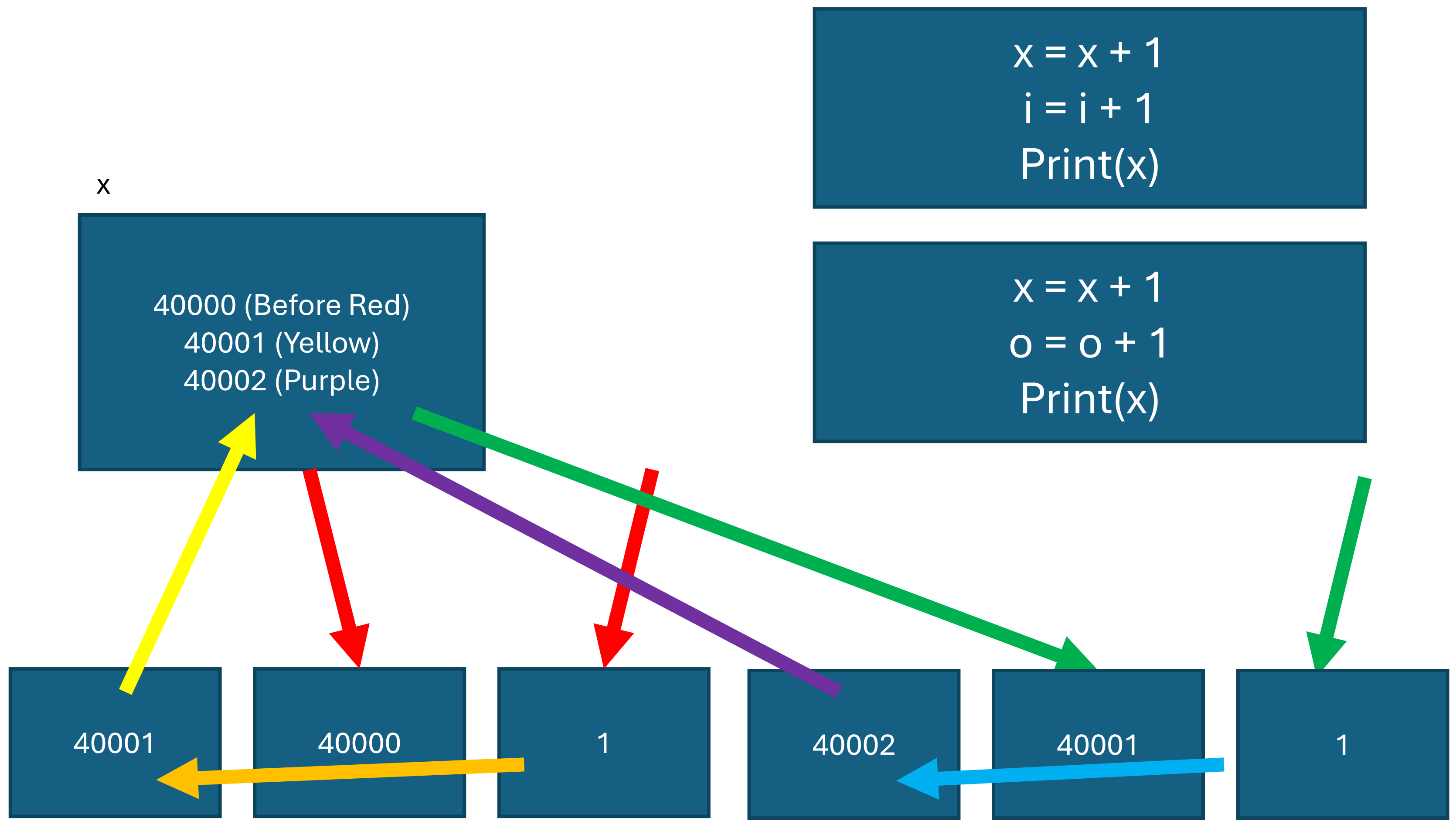
x, i, o all start = 0

What will be printed out?

```
x = x + 1
i = i + 1
Print(x, i)
```

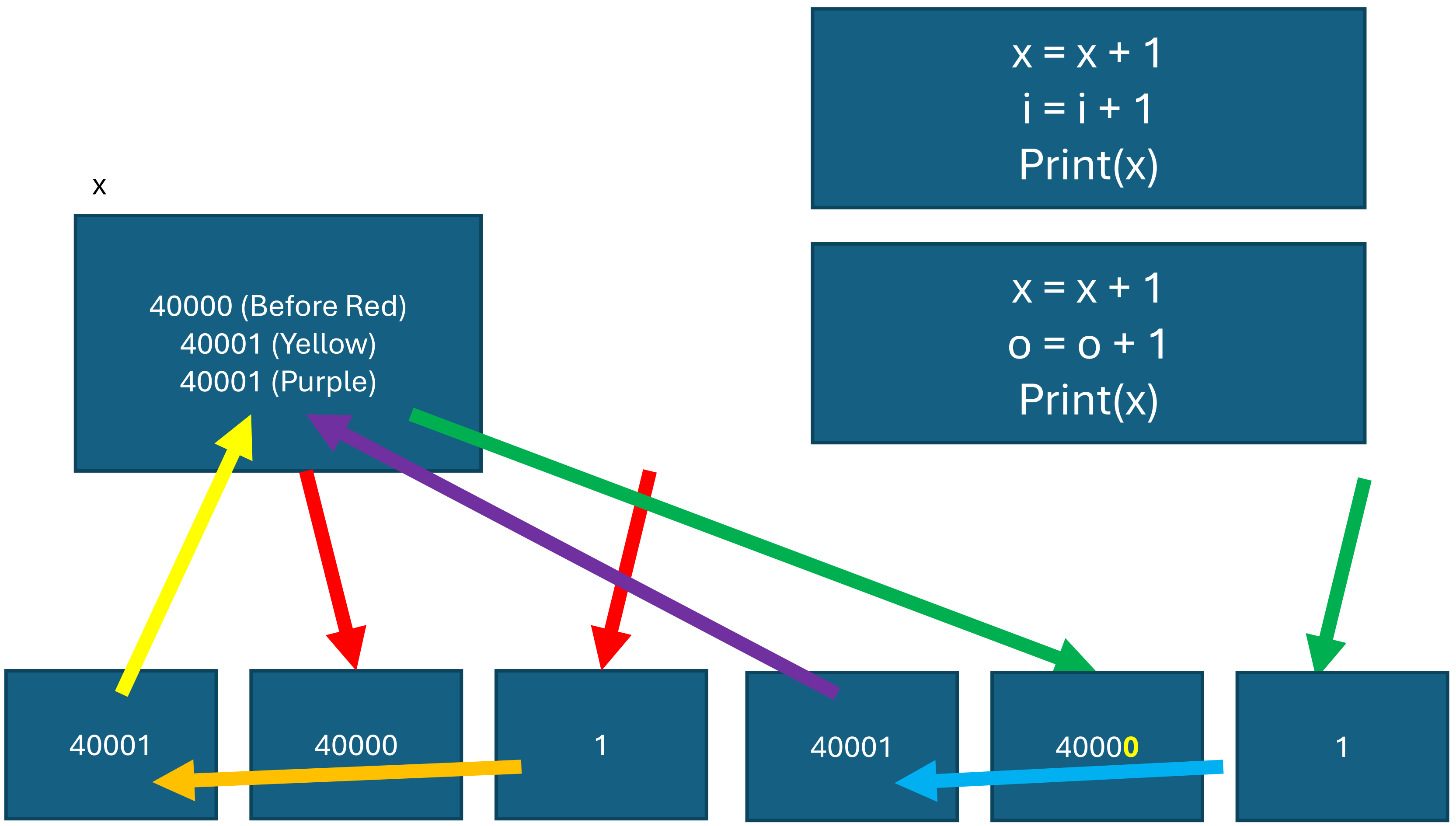
```
x = x + 1
o = o + 1
Print(x, o)
```

Red, Orange, Yellow, Green, Blue, Purple = GOOD
Red, Green, Anything = BAD



Red, Orange, Yellow, Green, Blue, Purple = GOOD

Red, Green, Anything = BAD



Race Conditions...

This is a single 'logical' step

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

Mess with 0x123

The atomic steps are no longer atomic

The Conch Shell

Lord of the Flies

- Only the person with the conch shell can speak

Kicking the Can Down the Road

What if?

```
incrementBalance()
{
        while (editing)
        {
                // Do nothing
        }
        editing = true;
        balance = balance + 1;
        editing = false;
}
```

Explanation:

- Two processes call increment balance…
- One sets editing to **true**
- The other waits until editing becomes **false**

Kicking the Can Down the Road

What if?

```
incrementBalance()
{
          while (someone_has_conch)
          {
                    // Do nothing
          }
          someone_has_conch = true;
          balance = balance + 1;
          someone_has_conch = false;
}
```

Explanation:

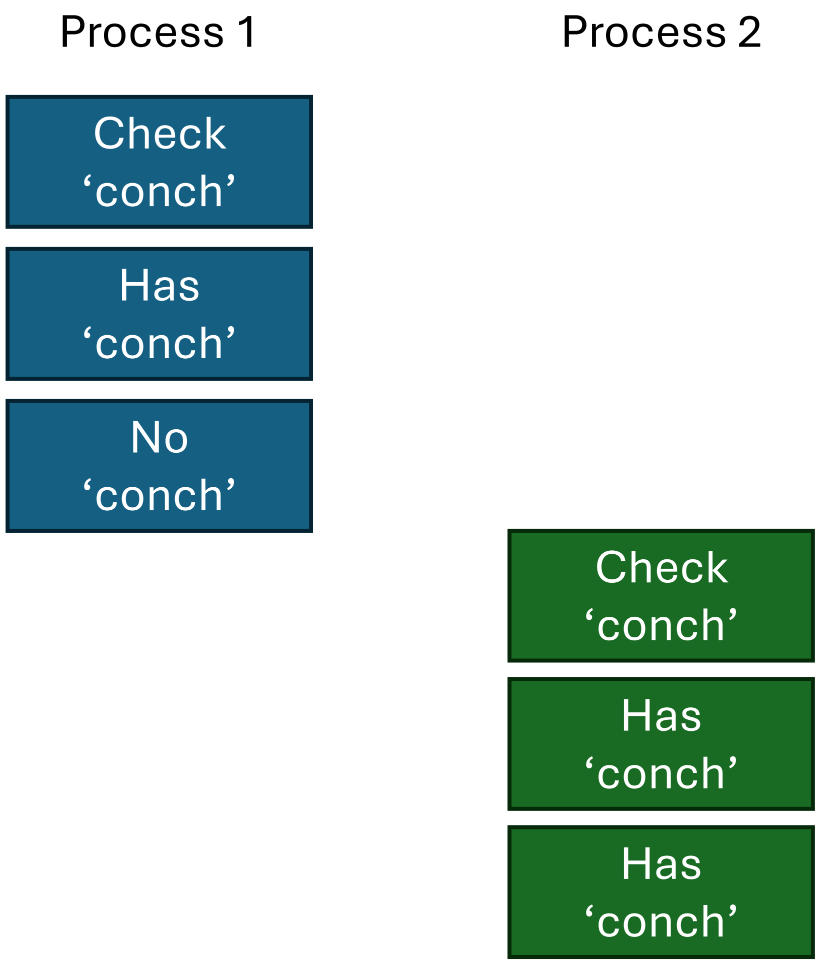
- Two processes call increment balance…
- One sets `someone_has_conch` to **true**
- The other waits until `someone_has_conch` becomes **false**

What is the problem here?

Kicking the Can Down the Road

What if?

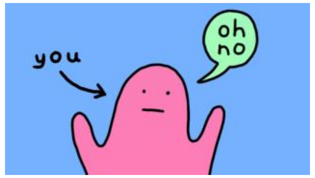
```
incrementBalance()
{
        while (someone_has_conch)
        {
                // Do nothing
        }
        someone_has_conch = true;
        balance = balance + 1;
        someone_has_conch = false;
}
```



Kicking the Can Down the Road

What if?

```
incrementBalance()
{
        while (someone_has_conch)
        {
                // Do nothing
        }
        someone_has_conch = true;
        balance = balance + 1;
        someone_has_conch = false;
}
```



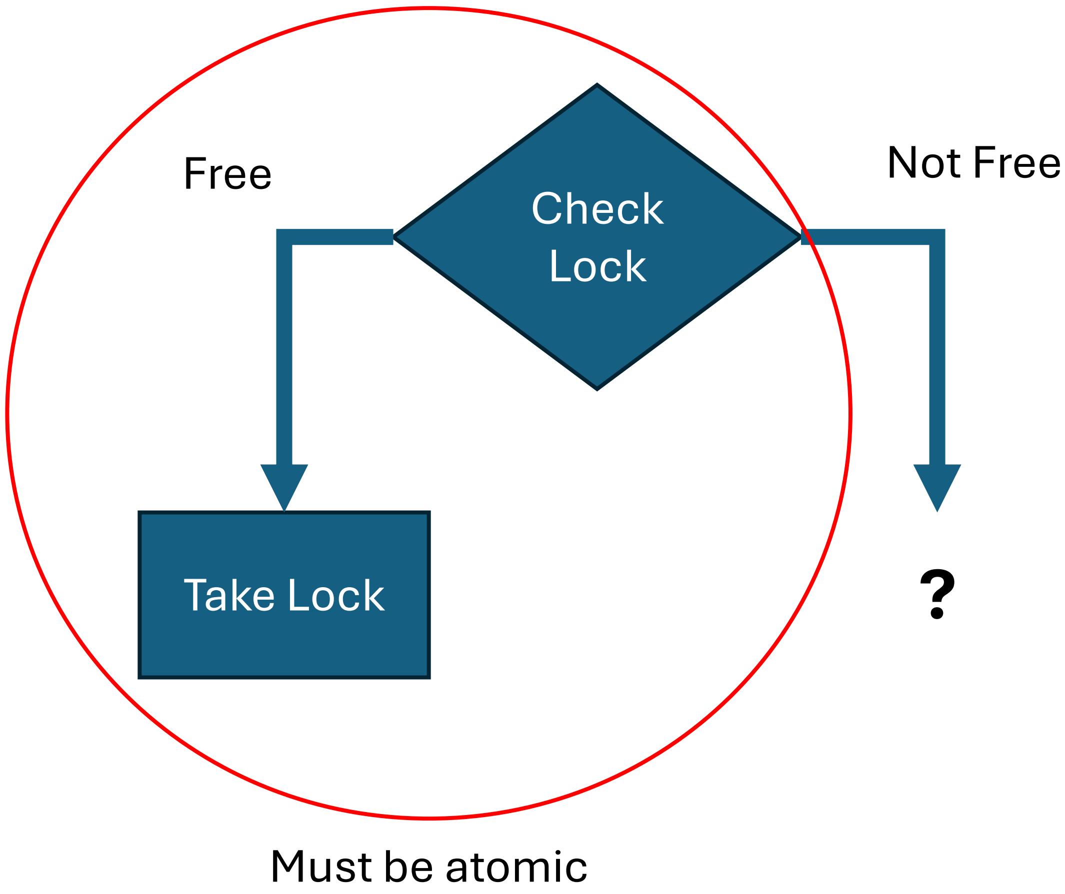
Process 1	Process 2
Check ‘conch’	
	Check ‘conch’
Has ‘conch’	
	Has ‘conch’
No ‘conch’	
	Has ‘conch’

Conch is not enough...

Requirement:

- Check availability
- Take

Must be atomic



Mutexes in C

How to write the code...

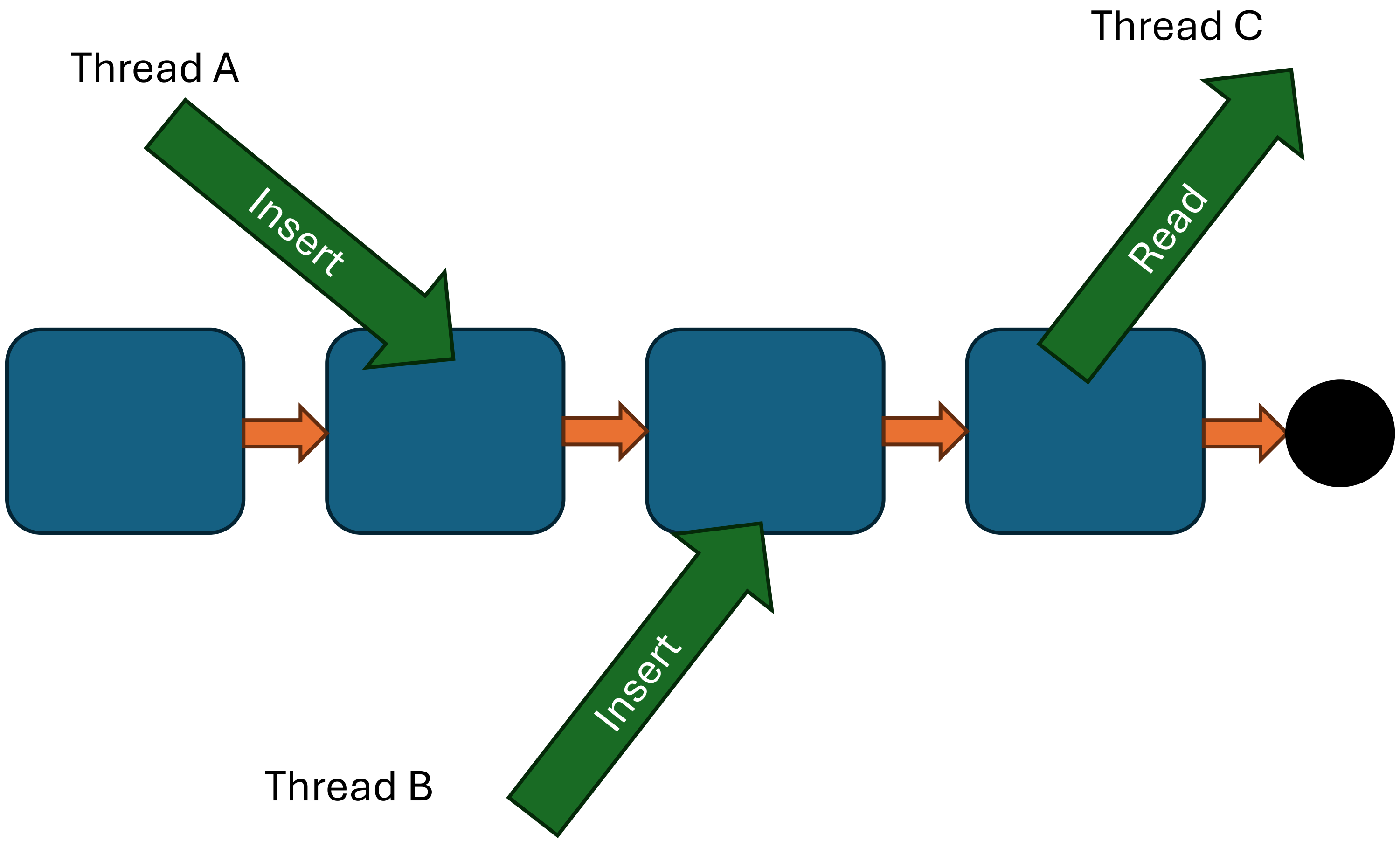
Mutex

Mutual Exclusion in C

```
// Setup
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
…
// Use
pthread_mutex_lock(%lock);
//Atomic code
balance = balance + 1;
pthread_mutex_unlock(%lock);
```

We can have many ☺

Example: Linked List



Example: Linked List

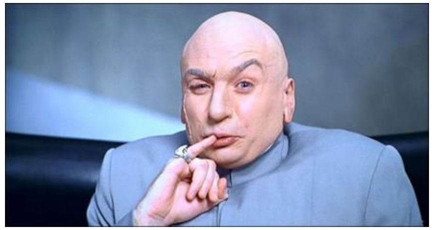
```
typedef struct __node_t
{
        int key;
        struct __node_t *next;
} node_t;

typedef struct __list_t
{
        node_t * head;
} list_t;

void ListInit(list_t *L)
{
        L->head = NULL;
}
```

Example: Linked List

```c
void ListInsert(list_t *L, int key)
{
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        new_node->next = L->head;
        L->head = new_node;
}
```

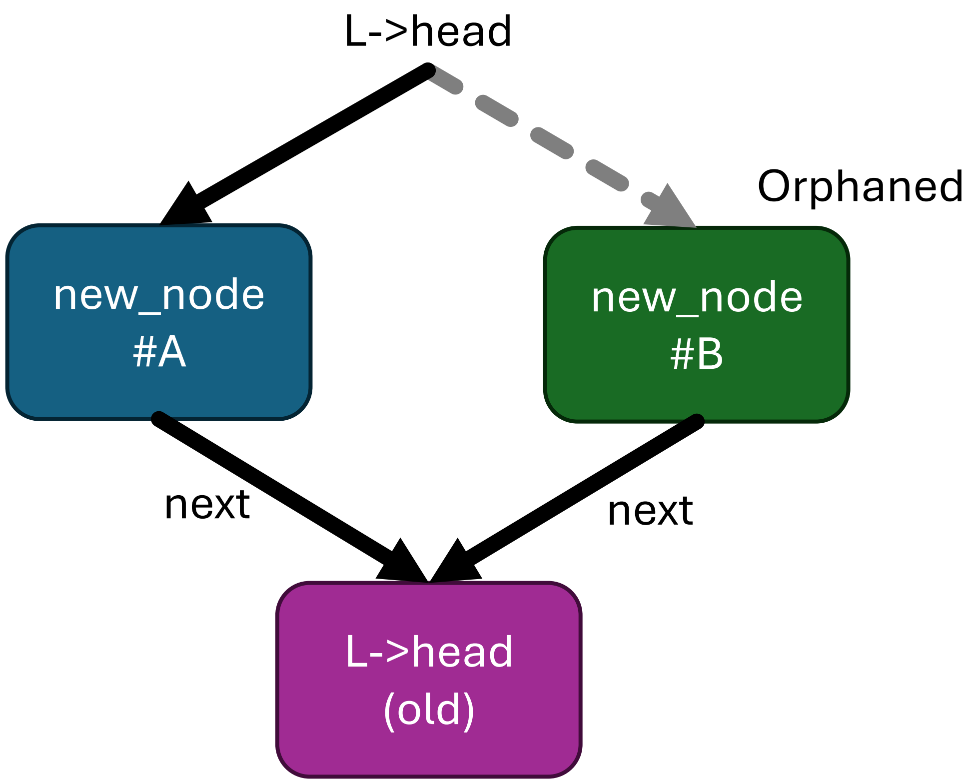
```c
int ListLookup(list_t *L, int key)
{
        node_t *temp = L->head;
        while (temp)
        {
                if (temp->key == key)
                        return 1;
                temp = temp->next;
        }
        return 0;
}
```

How to break this?



Example: Linked List

```
void ListInsert(list_t *L, int key)
{
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        new_node->next = L->head;
        L->head = new_node;
}
```



Thread A

```
new_node->key = key;
new_node->next = L->head;
```

Thread B

```
new_node->key = key;
new_node->next = L->head;
L->head = new_node;
```

```
L->head = new_node;
```

Example: Linked List

```
typedef struct __node_t
{
            int key;
            struct __node_t *next;
} node_t;

typedef struct __list_t
{
            node_t * head;
            pthread_mutex_t lock;
} list_t;

void ListInit(list_t *L)
{
            L->head = NULL;
            pthread_mutex_init(&L->lock, NULL);
}
```

Initialisation

Example: Linked List: Approach #1

```
void ListInsert(list_t *L, int key)
{
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        new_node->next = L->head;
        L->head = new_node;
}
```

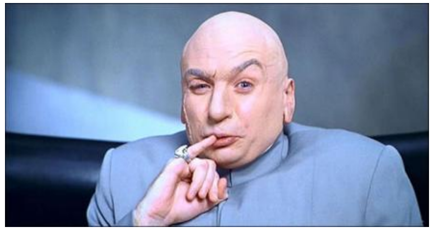
atomic action

```
int ListLookup(list_t *L, int key)
{
        node_t *temp = L->head;
        while (temp)
        {
                if (temp->key == key)
                        return 1;
                temp = temp->next;
        }
        return 0;
}
```

Where are the critical sections?

Example: Linked List: Approach #1

```c
void ListInsert(list_t *L, int key)
{
        pthread_mutex_lock(&L->lock);
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        new_node->next = L->head;
        L->head = new_node;
        pthread_mutex_unlock(&L->lock);
}
```



```c
int ListLookup(list_t *L, int key)
{
        pthread_mutex_lock(&L->lock);
        node_t *temp = L->head;
        while (temp)
        {
                if (temp->key == key)
                        return 1;
                temp = temp->next;
        }
        pthread_mutex_unlock(&L->lock);
        return 0;
}
```

How to break this?

Example: Linked List: Approach #2

```
void ListInsert(list_t *L, int key)
{
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        new_node->next = L->head;
        L->head = new_node;
}
```

atomic action

```
int ListLookup(list_t *L, int key)
{
        node_t *temp = L->head;
        while (temp)
        {
                if (temp->key == key)
                        return 1;
                temp = temp->next;
        }
        return 0;
}
```

Where are the critical sections?

Example: Linked List: Approach #2

```
void ListInsert(list_t *L, int key)
{
        node_t *new_node = malloc(sizeof(node_t));
        assert(new_node);
        new_node->key = key;
        pthread_mutex_lock(&L->lock);
        new_node->next = L->head;
        L->head = new_node;
        pthread_mutex_unlock(&L->lock);
}
```

```
int ListLookup(list_t *L, int key)
{
    pthread_mutex_lock(&L->lock);
    node_t *temp = L->head;
    while (temp)
    {
        if (temp->key == key)
        {
            pthread_mutex_unlock(&L->lock);
            return 1;
        }
        temp = temp->next;
    }
    pthread_mutex_unlock(&L->lock);
    return 0;
}
```

What is the smallest atomic action?

Lock Implementation

How to make it actually work?

Implementation

Design Goals

- Correct
  - Mutual Exclusion
  - No Deadlock
  - Bounded (starvation free)
- Fast & Cheap
- Fair

Notes

- Software solution appears impossible…

Bad Idea #1: Disable interrupts

Rationale

- No interrupts prevents the dispatcher from running another thread.

Problems

- Lock the CPU (evil process)
- Only works in a uni-processor environment

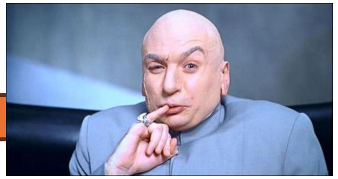
```
void lock(locT *l)
{
        disableInterrupts();
}

void lock(locT *l)
{
        enableInterrupts();
}
```

This is sometimes a viable tactic in **kernel** programming (embedded systems)

Bad Idea #2: Load & Store

```
typedef struct __lock_t {
        int flag;
} lock_t;

void init(lock_t * mutex)
{
        mutex->flag = 0;
}
```

```
void lock(lock_t * mutex)
{
        while (mutex->flag == 1)
        {
        }
        mutex->flag = 1;
}
void unlock(lock_t * mutex)
{
        mutex->flag = 0;
}
```



"Spin locks" are **really** inefficient

Bad Idea #2: Load & Store

```c
void lock(lock_t * mutex)
{
        while (mutex->flag == 1)
        {
        }
        mutex->flag = 1;
}
```

```c
void lock(lock_t * mutex)
{
        while (mutex->flag == 1)
        {
        }
        mutex->flag = 1;
}
```

```c
while (mutex->flag == 1)
{}
```

```c
while (mutex->flag == 1)
{}
mutex->flag = 1;
```

```c
mutex->flag = 1;
```

Idea #3: Peterson's Algorithm

```
int turn = 0;
boolean flag[2];

void init()
{
        flag[0] = flag[1] = false;
        turn = 0;
}
```

```
void lock()
{
        flag[self] = true;
        turn = 1 - self;
        while(flag[1-self] && (turn == 1-self))
        {}
}

void unlock()
{
        flag[self] = false;
}
```

There are two flags? One per thread

Idea #3: Peterson's Algorithm

Intuition

- Do stuff if:
  - Other thread isn't interested
  - Other thread gave you permission
- Progress
  - No infinite waiting (one thread or other must pass)
- Starvation
  - Each thread only waits once

But…

- Cache?
- The caches across CPUs may not be consistent
- Instructions may not actually be run in order (modern processors)
- Write to memory in execution order (cache activity, optimisations)

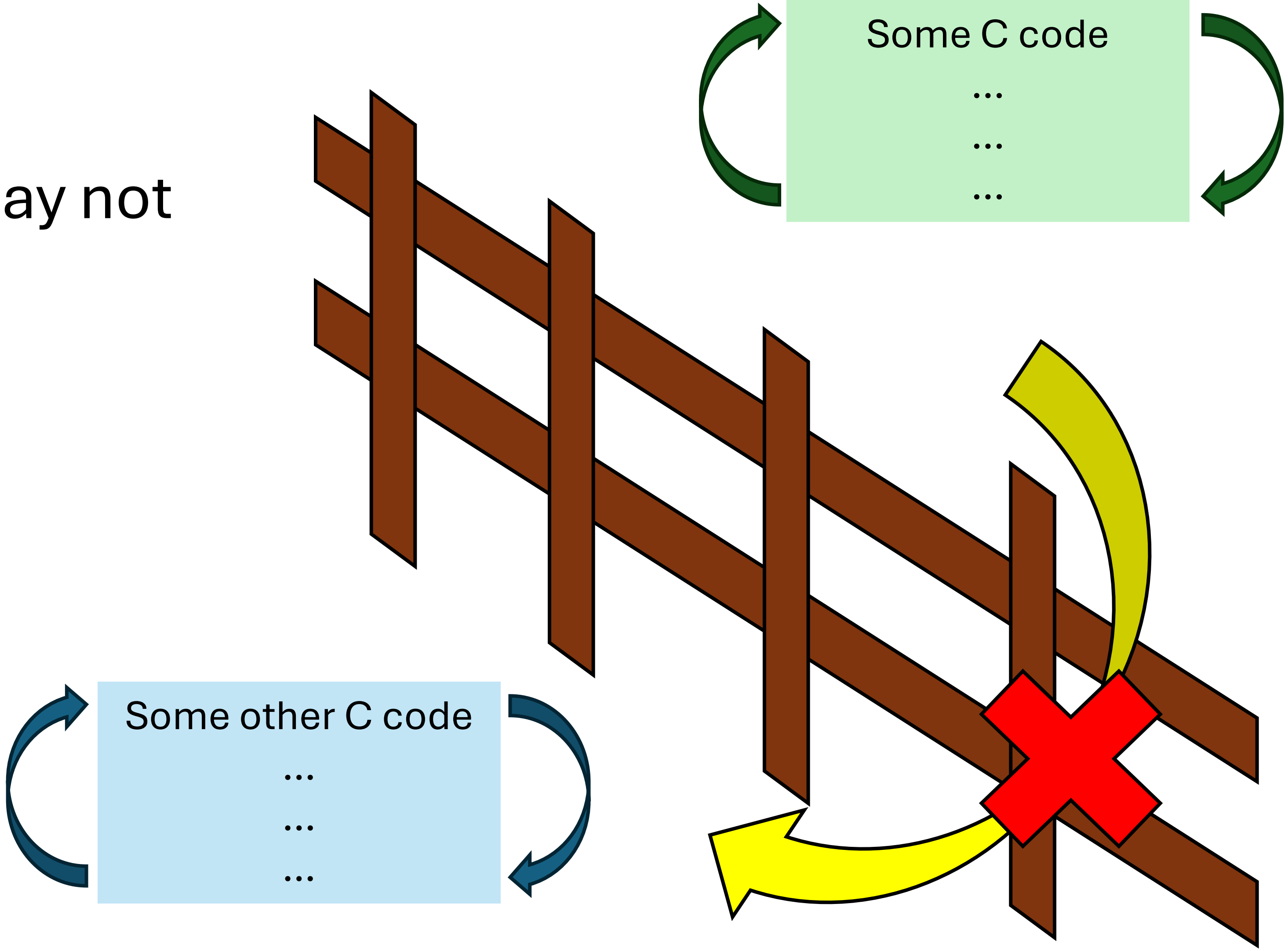
Memory Barrier Instruction

Idea:

- Memory accesses may not cross the barrier

C:

```
__sync_synchronize();
```



XCHG: Atomic Exchange "Test and Set"

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval inaddr

int xchg(int * addr, int newval)
{
        int old = *addr;
        *addr = newval;
        return old;
}
```

atomic action

Rationale?

- You can set something and ensure that the 'thing' changed? Or didn't change?

XCHG: Atomic Exchange “Test and Set”

```
void lock(lock_t * lock)
{
        while (xchg(&lock->flag, 1) == 1)
        {
        }
}

void unlock(lock_t * lock)
{
        lock->flag = 0;
}
```

Logic:

- Set lock to one…
- But if it is already one…
- Do nothing and try again

Compare and Swap

```
int CAS(int * addr, int expected, int new)
{
        int actual = *addr;
        if (actual == expected)
                *addr = new;
        return actual;
}
void lock(lock_t * lock)
{
        while (CAS(&lock->flag, 0, 1) == 1)
        {
        }
}
```

Subtle Difference:

- What is it? What should it be?
- If it is what it should be update it…
- Either way return what it is?

Implementation

Design Goals

- Correct
  - **Mutual Exclusion**
  - No Deadlock
  - Bounded (starvation free)
- Fast & Cheap
- Fair **?**

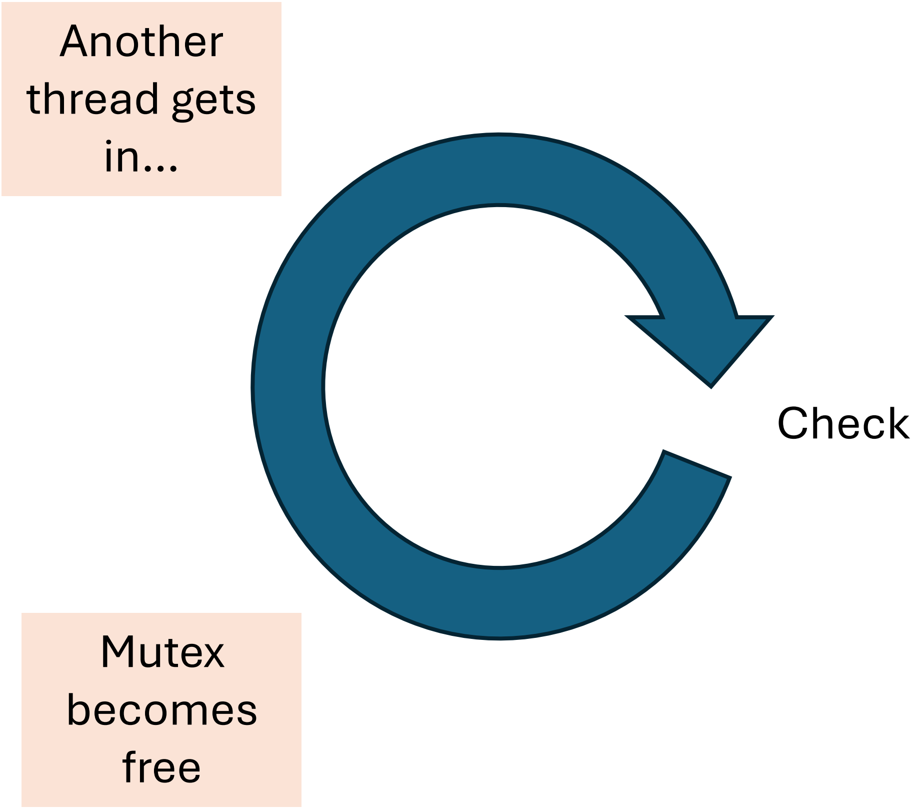
Notes:

- Hardware support

Ticket Locks

Issue:

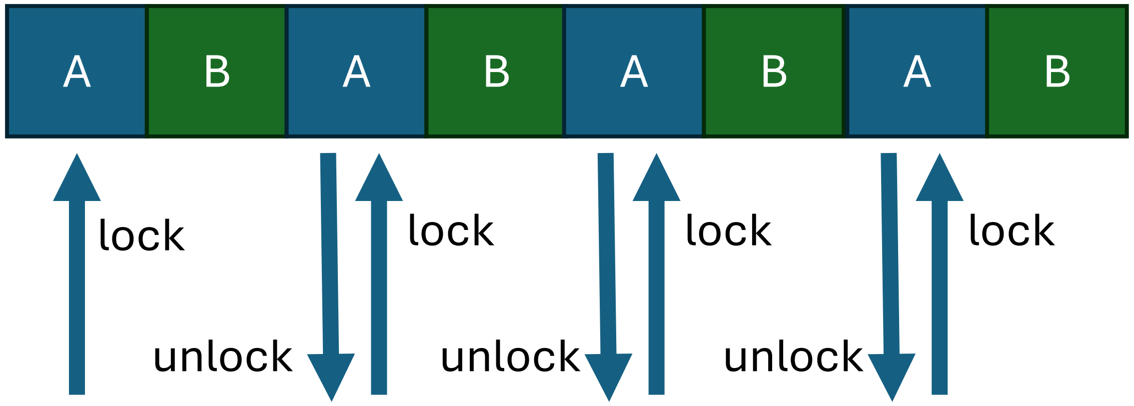
- Spin-locks are arbitrary?



Ticket Locks

Issue:

- Spin-locks are arbitrary?



Ticket Locks

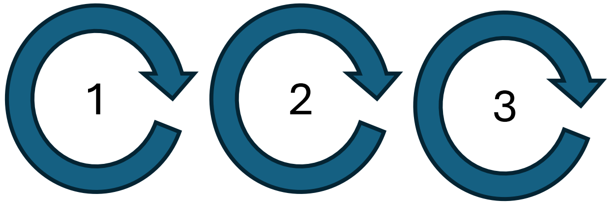
Issue:

- Spin-locks are arbitrary?

Idea:

- Have an incremented value (atomically) while returning the old value
- Spin while 'not your turn'
- Increment turn when done...

```
int FetchAndAdd(int *ptr)
{
        int old = *ptr;
        *ptr = old + 1;
        return old;
}
```



Ticket Locks

A lock()

B lock()

C lock()

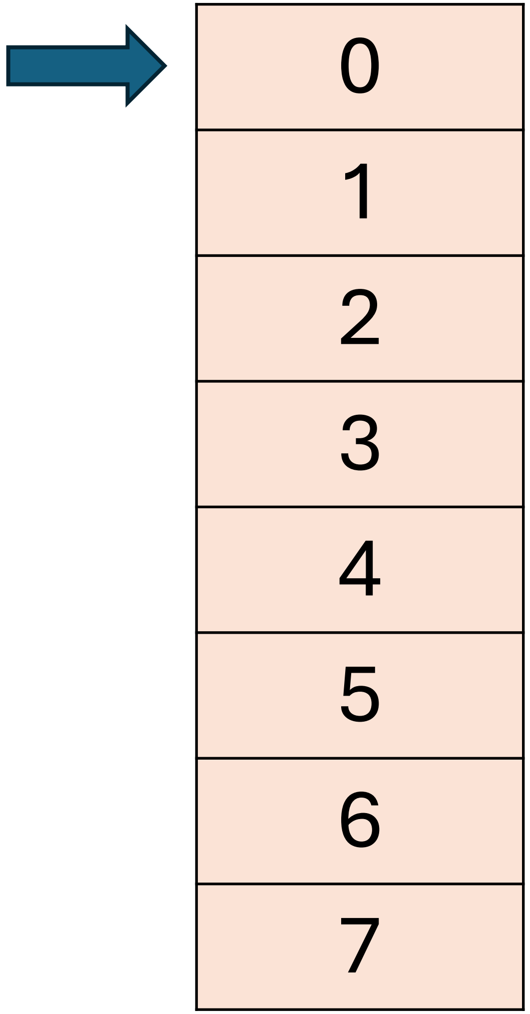
0
1
2
3
4
5
6
7

Ticket Locks

A lock() → Ticket 0

B lock() → Ticket 1

C lock() → Ticket 2



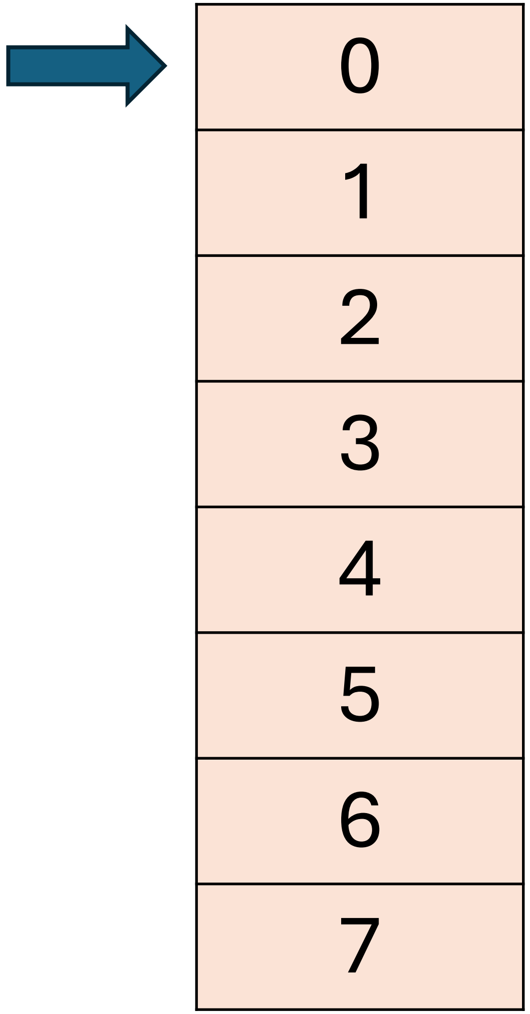
Ticket Locks

A lock() → Ticket 0

B lock() → Ticket 1

C lock() → Ticket 2

A runs



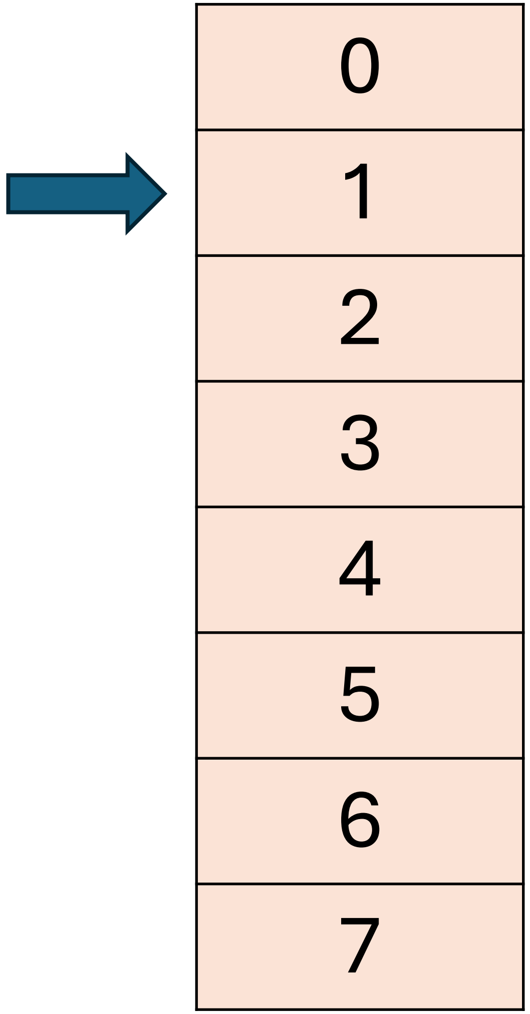
Ticket Locks

A lock() → Ticket 0

B lock() → Ticket 1

C lock() → Ticket 2

A runs → turn = 1



Ticket Locks

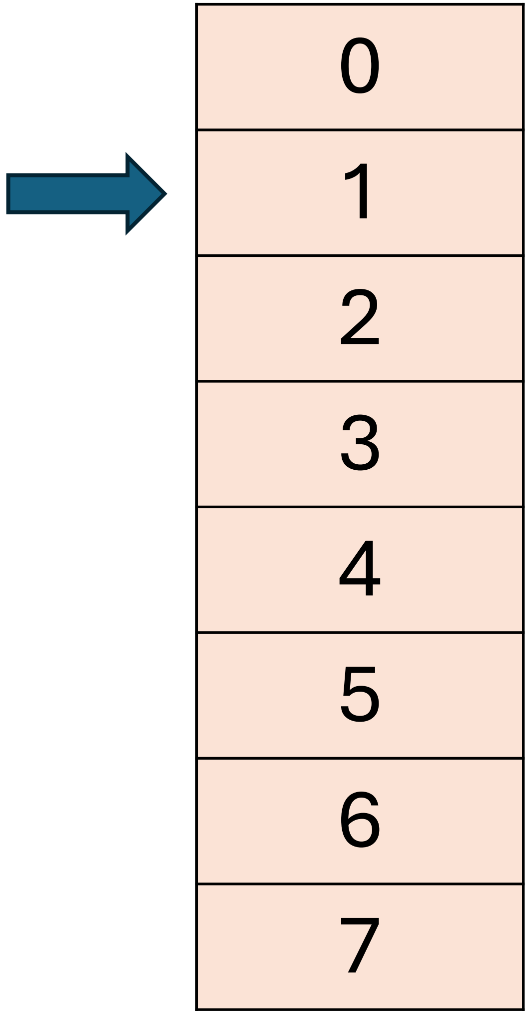
A lock() → Ticket 0

B lock() → Ticket 1

C lock() → Ticket 2

A runs → turn = 1

B runs



Ticket Locks

A lock() → Ticket 0

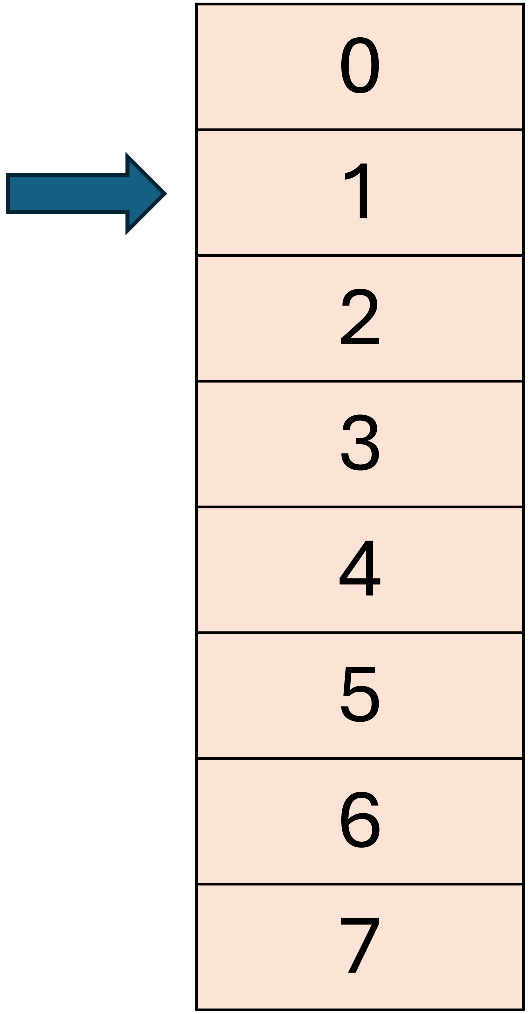
B lock() → Ticket 1

C lock() → Ticket 2

A runs → turn = 1

B runs

A lock() → Ticket 3



Ticket Locks

A lock() → Ticket 0

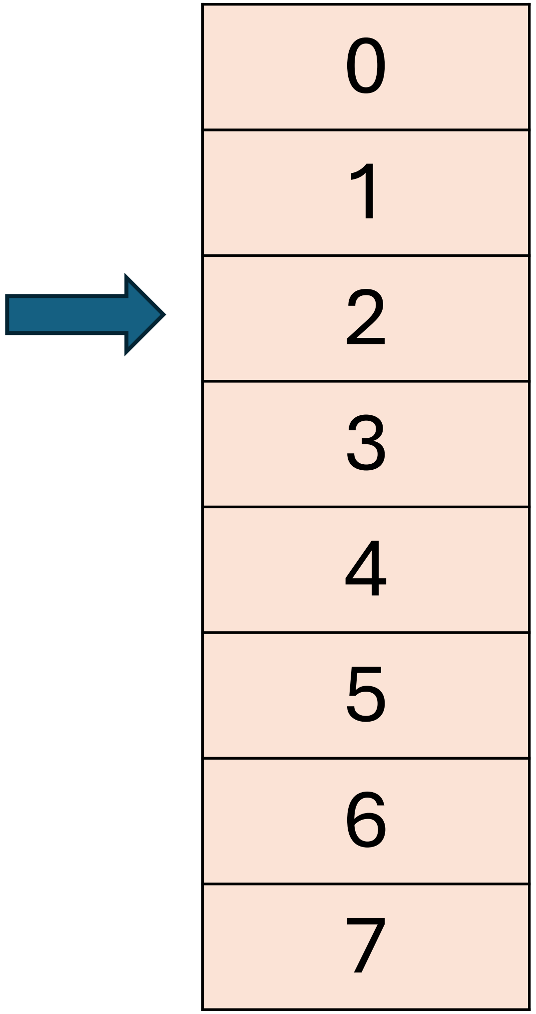
B lock() → Ticket 1

C lock() → Ticket 2

A runs → turn = 1

B runs → turn = 2

A lock() → Ticket 3



Ticket Locks

A lock() → Ticket 0

B lock() → Ticket 1

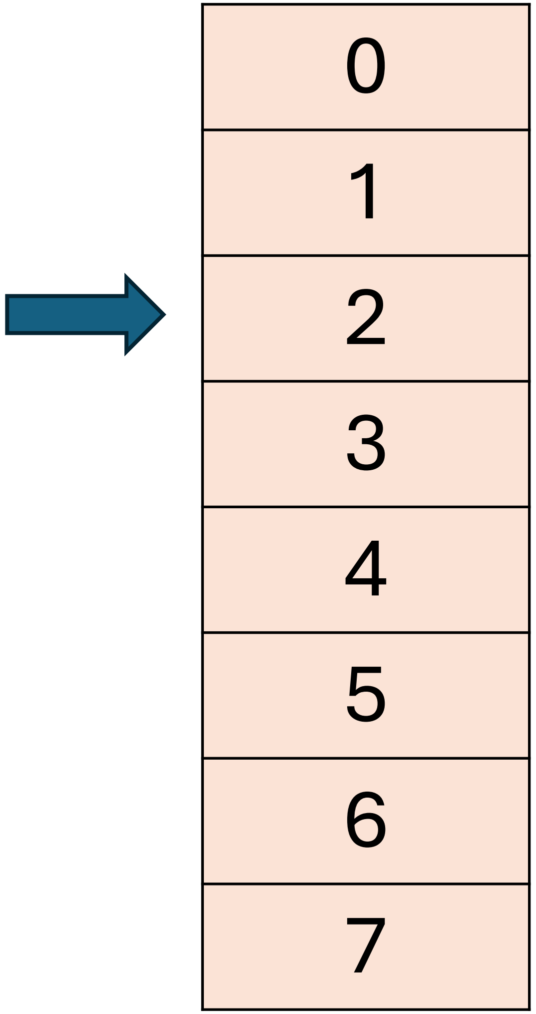
C lock() → Ticket 2

A runs → turn = 1

B runs → turn = 2

A lock() → Ticket 3

C runs



Ticket Locks

A lock() → Ticket 0

B lock() → Ticket 1

C lock() → Ticket 2

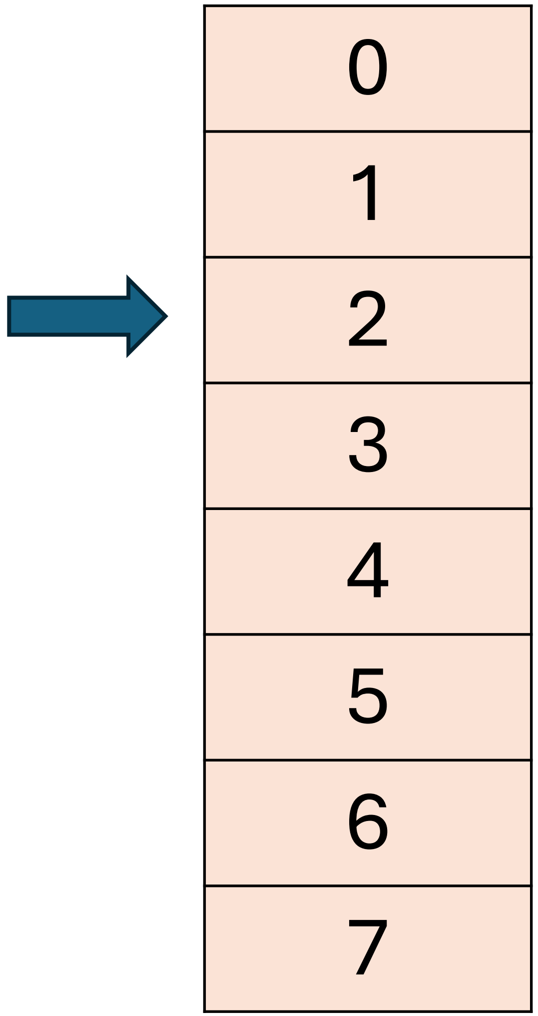
A runs → turn = 1

B runs → turn = 2

A lock() → Ticket 3

C runs

B lock() → Ticket 4



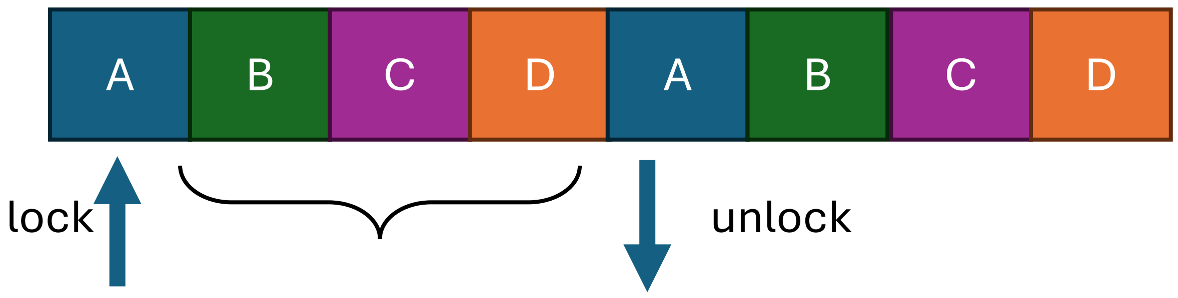
Spinlocks

Fast If:

- Many CPUs
- Locks held for short time
- (no context switch required)

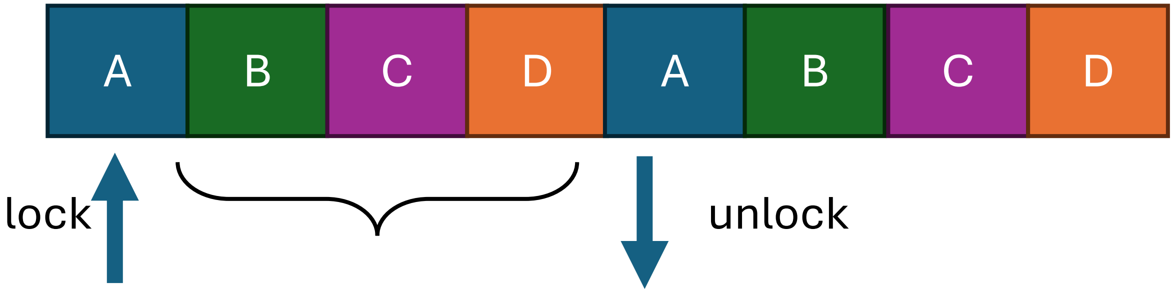
Slow If:

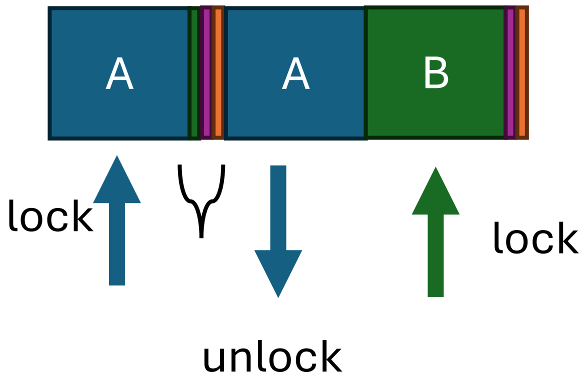
- One CPU
- Locks held for long time
- (spinning is just bad)



Why run B, C, D if they are waiting for A?
What does it achieve?

Yielding






```
typedef struct __lock_t
{
        int ticket;
        int turn;
} lock_t;

int FetchAndAdd(int *ptr)
{
        int old = *ptr;
        *ptr = old + 1;
        return old;
}


void acquire(lock_t *lock)
{
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        yield();
}

void release(lock_t *lock)
{
    FetchAndAdd(&lock->turn);
}
```

Evaluating Lock Designs

Fairness:

- Is request order the same as receipt order?

Performance:

- Low contention (lock mostly there)
- High contention (lock mostly in use)

This is effectively scheduling

Spinlock Performance Comparison

CPU Cost:

- Without yield: O(threads * time_slice)
- With yield: O(threads * context_switch)
- Yielding is still expensive

Implementation: Block when Waiting

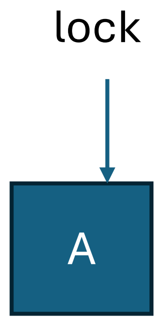
Concept:

- If a thread is ‘waiting’, put it on the blocked queue
- Scheduler only runs ‘ready’ processes

Here the OS scheduler doesn’t need to be involved (i.e., separation of concerns)

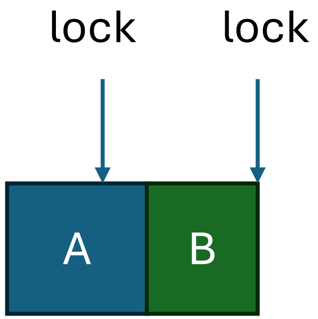
Example

Running: A
Ready: B, C, D
Blocked:



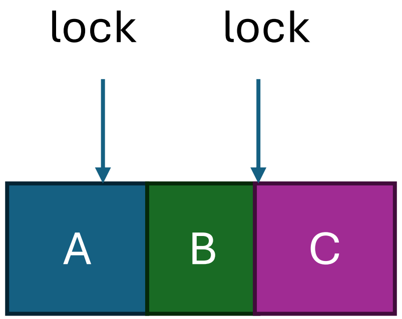
Example

Running: B
Ready: C, D, A
Blocked:



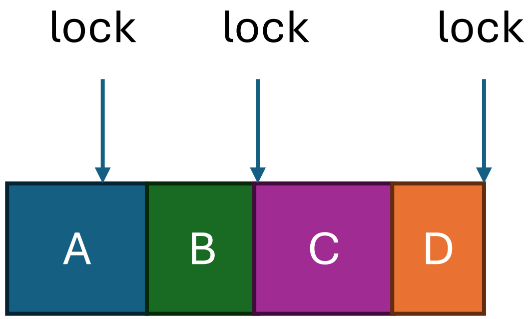
Example

Running: C
Ready: D, A
Blocked: B



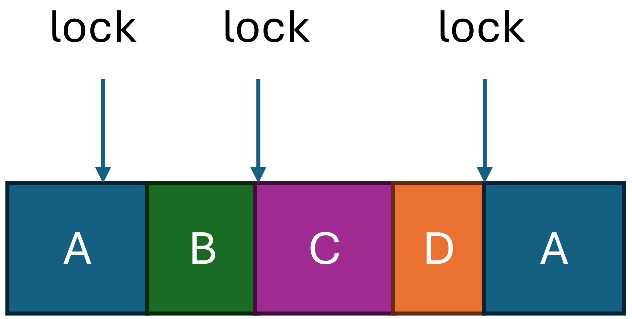
Example

Running: D
Ready: A, C
Blocked: B

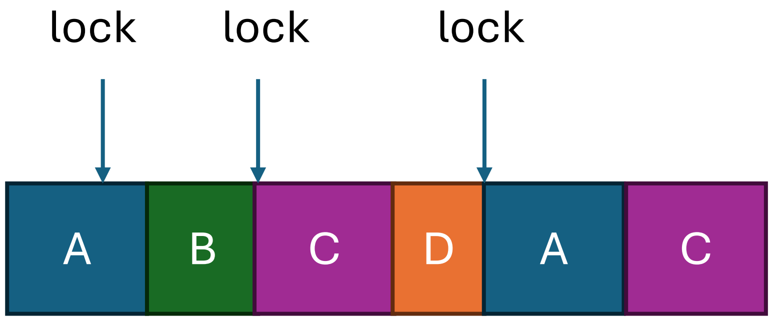


Example

Running: A
Ready: C
Blocked: B, D



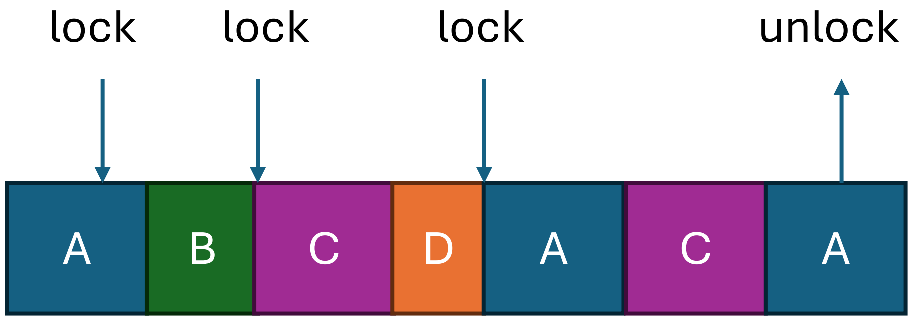
Example



Running: C
Ready: A
Blocked: B, D

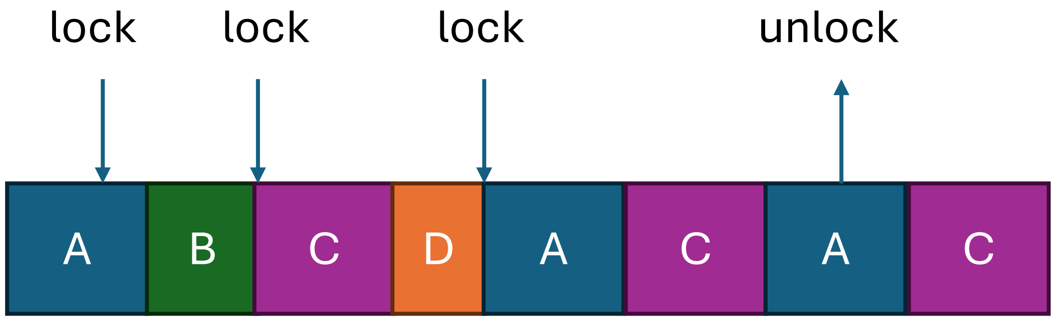
Example

Running: A
Ready: C, B
Blocked: D



Example

Running: C
Ready: B, A
Blocked: D



B is probably feeling a bit hard done by...

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

We get stuck by the guard...

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Flag should normally be zero to start, so we set the lock and reset the guard

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

The second process will get added to a queue, reset the guard and then get parked()

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Again, we're going to worry about the guard

Code: Block When Waiting

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Here we unpark the 2nd process (the head of the queue)

Code: Block When Waiting

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Some Questions:
The Guard?
Spinning Guard?
Unpark doesn't reset Flag?
Race condition?

Code: Block When Waiting

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

The guard acts to separate flag setting and queuing behaviours?

Some Questions:
The Guard?
Spinning Guard?
Unpark doesn't reset Flag?
Race condition?

Code: Block When Waiting

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Some Questions:
The Guard?
Spinning Guard?
Unpark doesn't reset Flag?
Race condition?

The Guard spins minimally...

Code: Block When Waiting

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

Some Questions:
The Guard?
Spinning Guard?
Unpark doesn't reset Flag?
Race condition?

Allows new process 'handover'

Race Condition

```
if (l->flag) {
        queue_add(l->q, gettid());
        l->guard = 0;
```

```
while (TestAndSet(&l->guard, 1) == 1);
if (queue_empty(l->q)) {
        l->flag = 0;
} else {
        unpark(queue_remove(l->q));
}
l->guard = 0;
```

```
        park();
```

I was told to wakeup just before
I went to sleep (instead of after)
Now I must sleep FOREVER

Code: Block When Waiting: Fix

```
typedef struct {
        bool flag;
        bool guard;
        queue_t *q;
} lock_t;
```

```
void lock_init(lock_t *m)
{
        m->flag = 0;
        m->guard = 0;
        queue_init(m->q);
}
```

```
void acquire(lock_t * l){
        while (TestAndSet(&l->guard, 1) == 1);
        if (l->flag) {
                queue_add(l->q, gettid());
                setpark(); // Notify of plan
                l->guard = 0;
                park();  // Blocked
        } else {
                l->flag = 1; // Lock acquired
                l->guard = 0;
        }
}
```

```
void release(lock_t * l) {
        while (TestAndSet(&l->guard, 1) == 1);
        if (queue_empty(l->q)) {
                l->flag = 0;
        } else {
                unpark(queue_remove(l->q));
        }
        l->guard = 0;
}
```

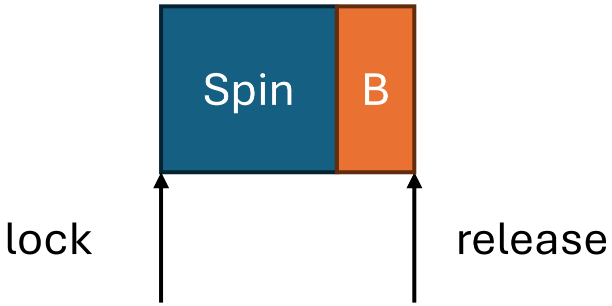
If I am about to sleep and someone ‘wakes’ me, I won’t go to sleep anymore.

Spin-Locking vs Blocking

How much does a context switch cost (C)?

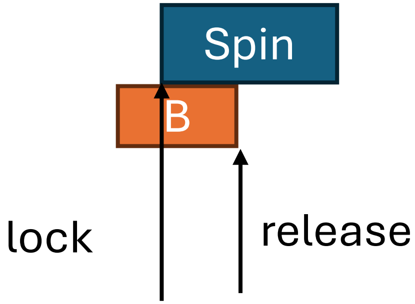
Uniprocessor:

- Waiting process is scheduled, process holding lock isn't
- Waiting process should always reliquinish processor
- Associate queue of waiters with each lock



Multiprocessor:

- Waiting process is scheduled, process holding lock could be?
- Spin or block depending on time before lock release
- Lock released **quickly**: spin
- Lock released **slowly**: block

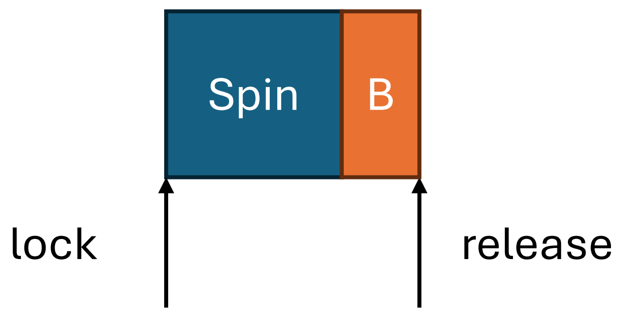


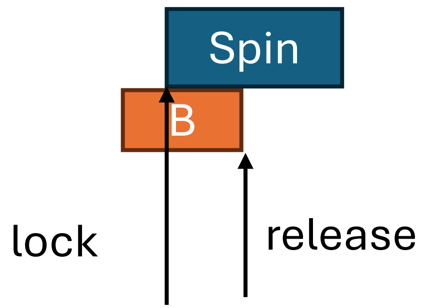
Spin-Locking vs Blocking

How much does a context switch cost (C)?

Question:

- We will wait **t** for the lock to be released
- If C > t:
  - Spin lock
- If C < t:
  - Block





Futex

Overview:

- futex is a linux-based implementation of mutex-style behaviour
    - Queue: Kernel
    - Atomic Integer: User space
- Main takeaway:
    - If the threads don't contend, the kernel system calls are never invoked ☺

futex_wait(address, expected)

- Puts the calling thread to sleep
- If the value at address is not expected, return call immediately

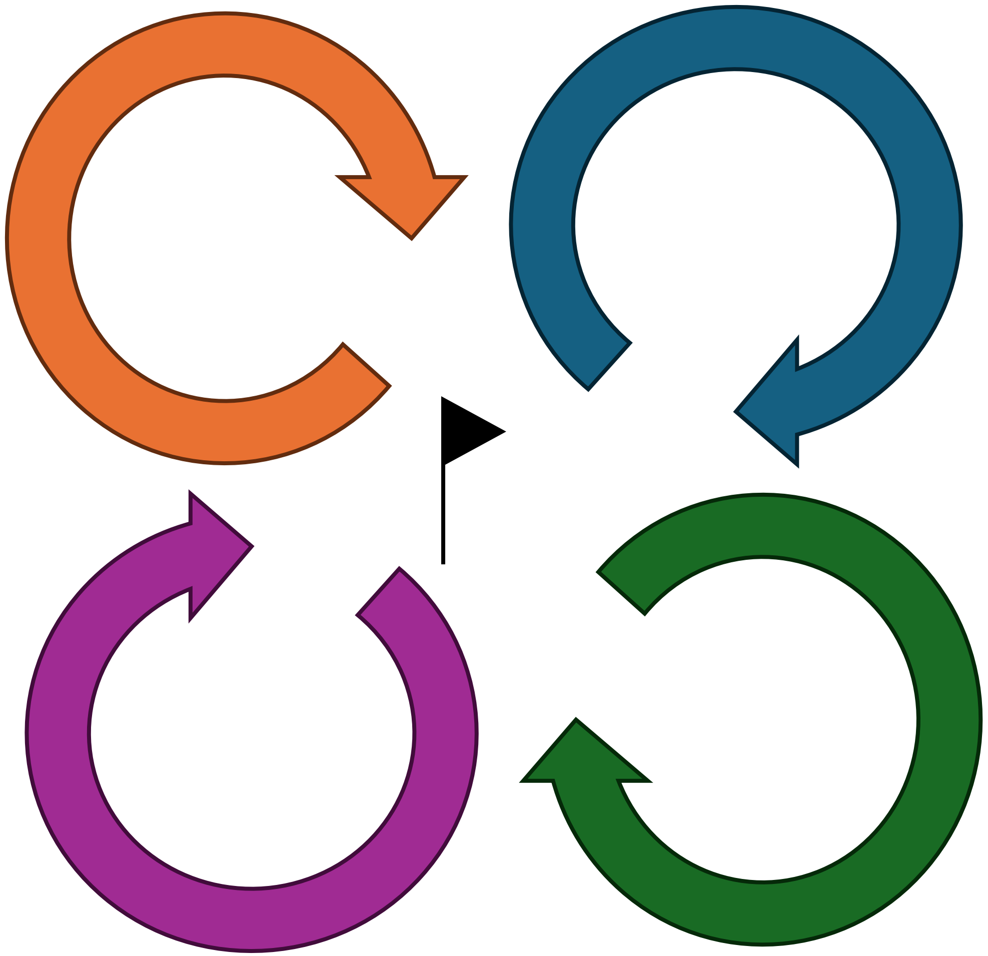
futex_wake(address)

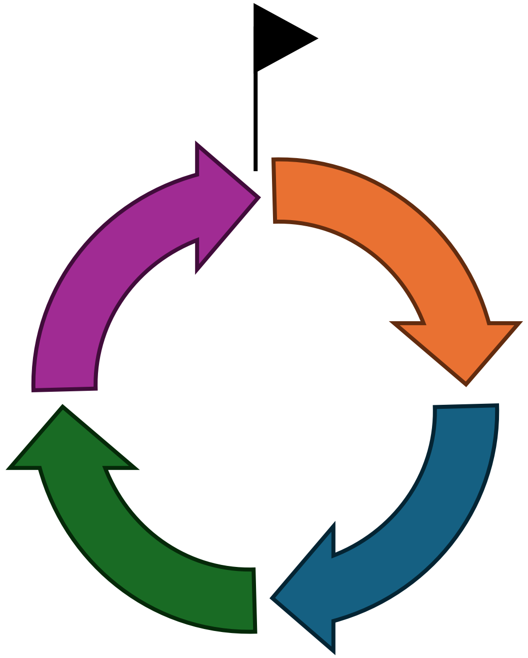
- Wake one thread waiting in the queue

Condition Variables

Threads aren’t independent

Relay Race

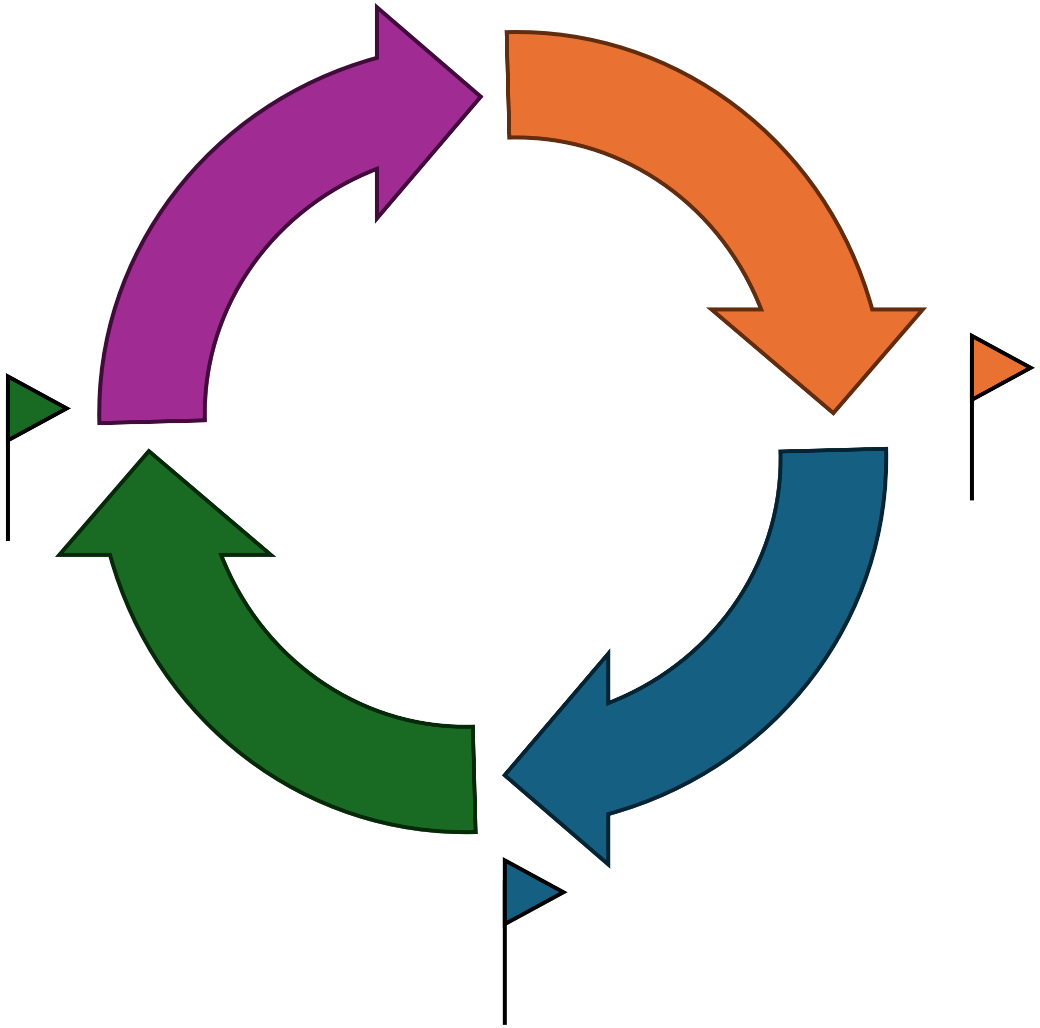




Logic

The Handover:

- **Wait** until they are done
- **Signal** when you are done



Wait!

Function Signature:

pthread_cond_wait(&cond, &lock)

cond:	pthread_cond_t
lock:	pthread_mutex_t
storage_t:	mailbox

Signal!

Function Signature:

pthread_cond_signal(&cond)

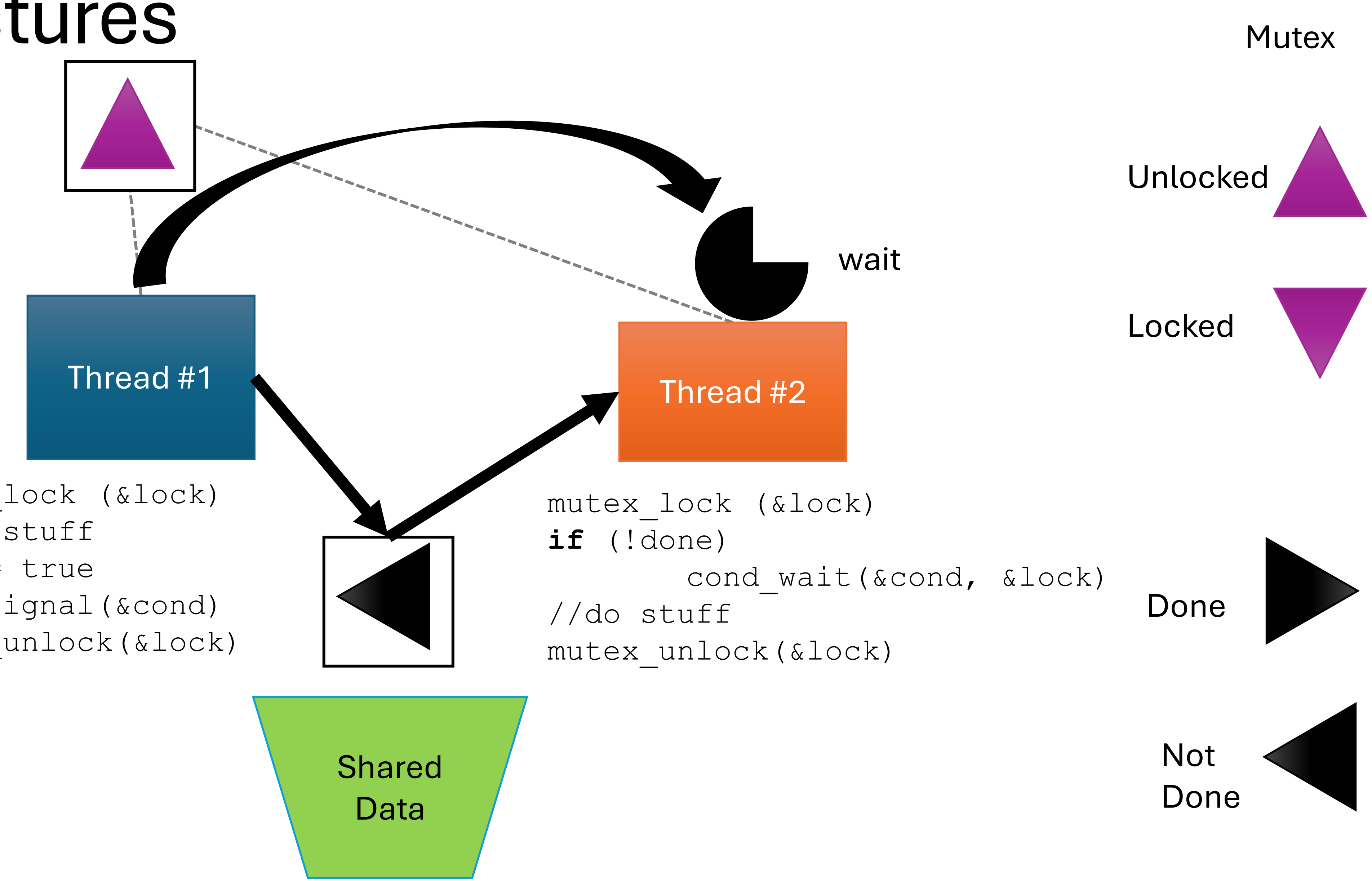
cond: pthread_cond_t

lock: pthread_mutex_t

storage_t: mailbox

Does nothing if no one is actually waiting

In Pictures




```
mutex_lock (&lock)
// do stuff
done = true
cond_signal(&cond)
mutex_unlock(&lock)
```

```
mutex_lock (&lock)
if (!done)
        cond_wait(&cond, &lock)
//do stuff
mutex_unlock(&lock)
```

In Code

```
mutex_lock (&lock)
// do stuff
done = true
cond_signal(&cond)
mutex_unlock(&lock)
```

```
mutex_lock (&lock)
if (!done)
      cond_wait(&cond, &lock)
//do stuff
mutex_unlock(&lock)
```

In Code

```
mutex_lock (&lock)
// do stuff
done = true
cond_signal(&cond)
mutex_unlock(&lock)
```

Sneak in here

```
mutex_lock (&lock)
if (!done)
    cond_wait(&cond, &lock)
//do stuff
mutex_unlock(&lock)
```

In Code

```
mutex_lock (&lock)
// do stuff
done = true
cond_signal(&cond)
mutex_unlock(&lock)
```

```
mutex_lock (&lock)
if (!done)
      cond_wait(&cond, &lock)
//do stuff
mutex_unlock(&lock)
```

What will we wait for?

Pipes

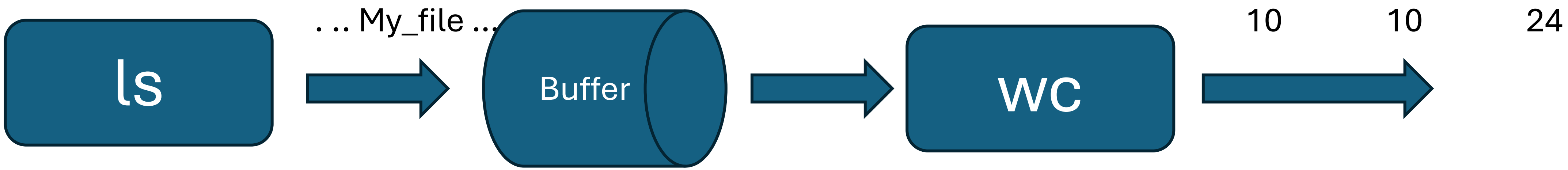
Ceci n’est pas une pipe

Example

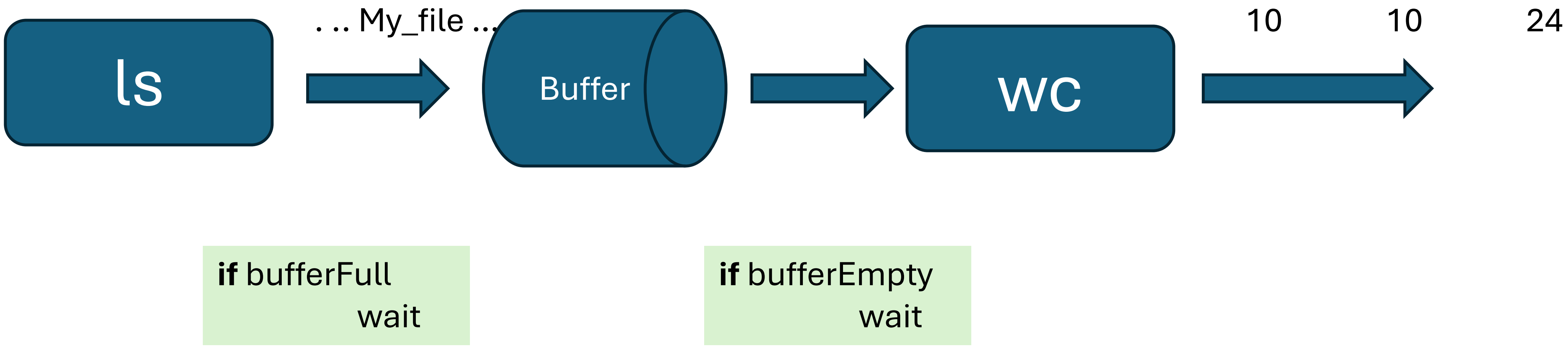
```
ls | wc
```

Pipes

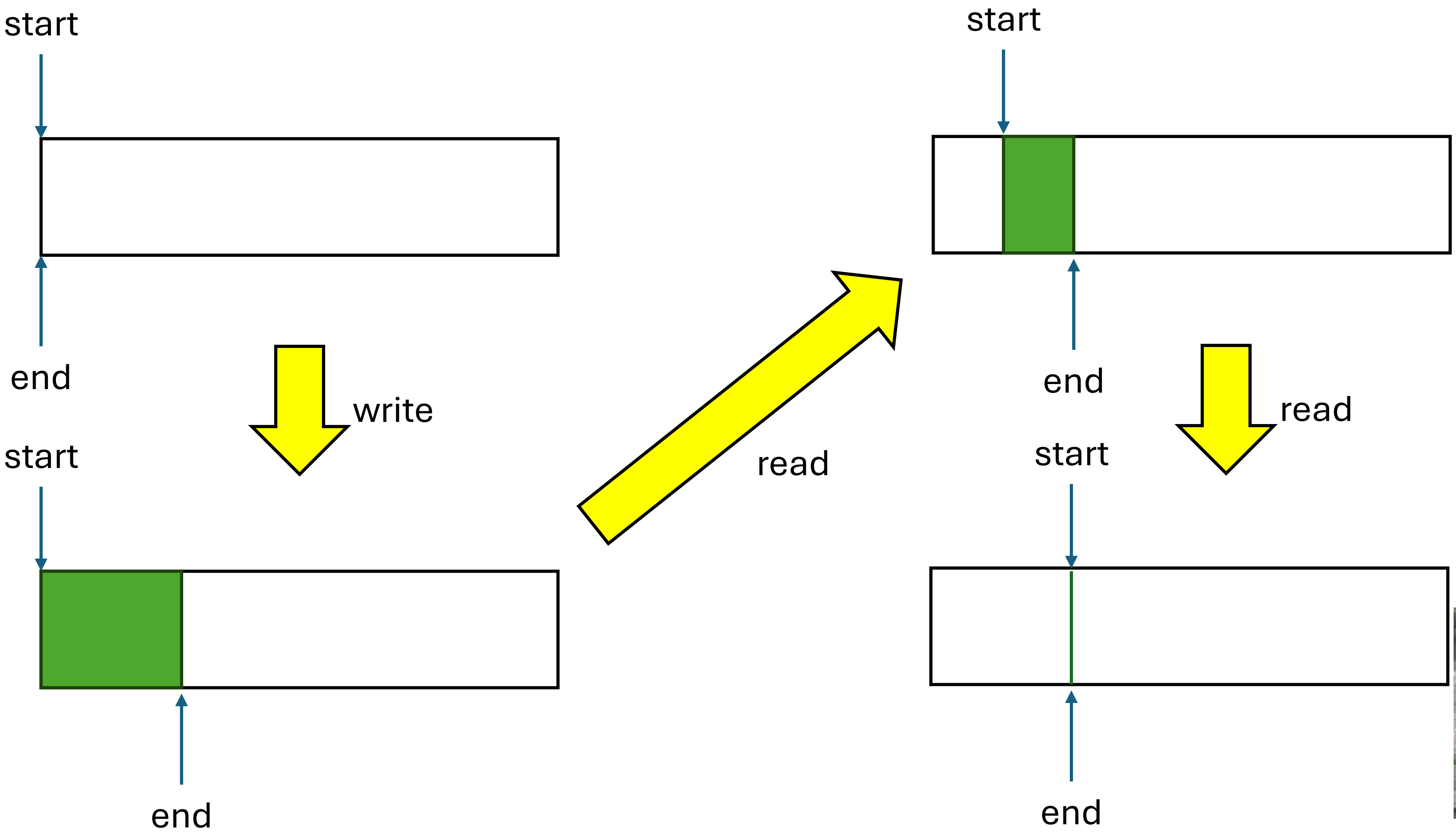
Pipes



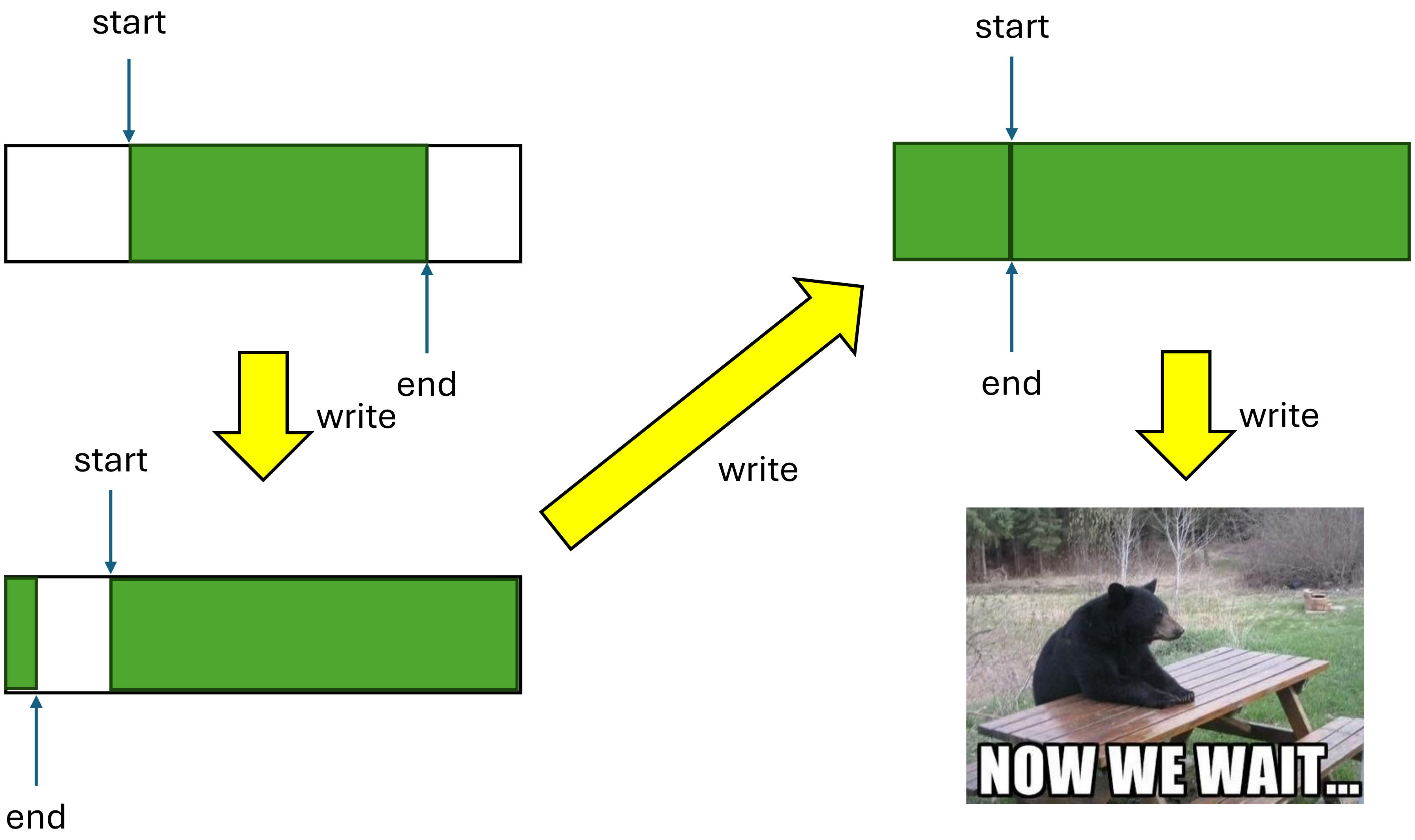
Pipes


ls
. .. My_file ...
Buffer
wc
10
10
24
if bufferFull
wait
if bufferEmpty
wait

Pipes: Buffer



Pipes: Buffer



Pipes: Producers and Consumers

Producer: Generate Data (pipe writer)

Consumer: Accept data and process it (pipe reader)

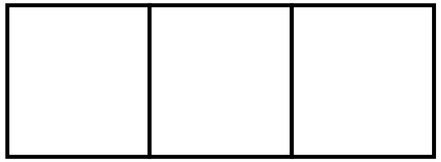
Examples:

- Server, Client (webservers)
- Pipes

Example: Code

```
void * producer(void * arg) {
        for (int i=0; i<loops; i++) {
                mutex_lock(&m);
                while (numfull == max)
                        cond_wait(&cond, &m);
                do_fill(i);
                cond_signal(&cond);
                mutex_unlock(&m);
        }
}
```

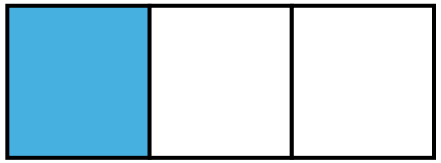
```
void * consumer(void * arg) {
        while (1) {
                mutex_lock(&m);
                while (numfull == 0)
                        cond_wait(&cond, &m);
                int tmp = do_get();
                cond_signal(&cond);
                mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```



Example: Code

```
void * producer(void * arg) {
        for (int i=0; i<loops; i++) {
                mutex_lock(&m);
                while (numfull == max)
                        cond_wait(&cond, &m);
                do_fill(i);
                cond_signal(&cond);
                mutex_unlock(&m);
        }
}
```

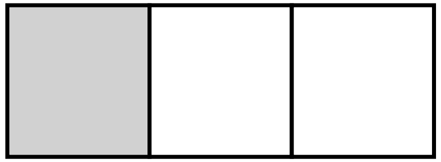
```
void * consumer(void * arg) {
        while (1) {
                mutex_lock(&m);
                while (numfull == 0)
                        cond_wait(&cond, &m);
                int tmp = do_get();
                cond_signal(&cond);
                mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```



Example: Code

```
void * producer(void * arg) {
        for (int i=0; i<loops; i++) {
                mutex_lock(&m);
                while (numfull == max)
                        cond_wait(&cond, &m);
                do_fill(i);
                cond_signal(&cond);
                mutex_unlock(&m);
        }
}
```

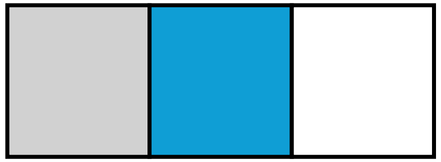
```
void * consumer(void * arg) {
        while (1) {
                mutex_lock(&m);
                while (numfull == 0)
                        cond_wait(&cond, &m);
                int tmp = do_get();
                cond_signal(&cond);
                mutex_unlock(&m);
                printf(“%d\n”, tmp);
        }
}
```



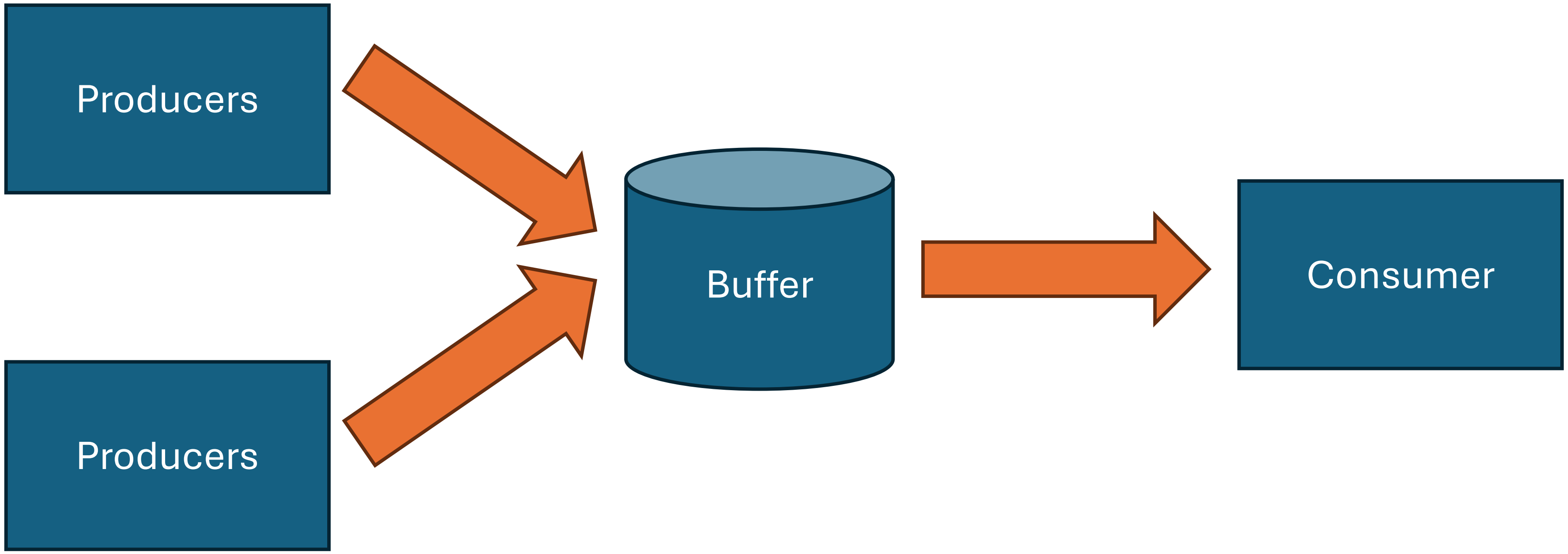
Example: Code

```
void * producer(void * arg) {
        for (int i=0; i<loops; i++) {
                mutex_lock(&m);
                while (numfull == max)
                        cond_wait(&cond, &m);
                do_fill(i);
                cond_signal(&cond);
                mutex_unlock(&m);
        }
}
```

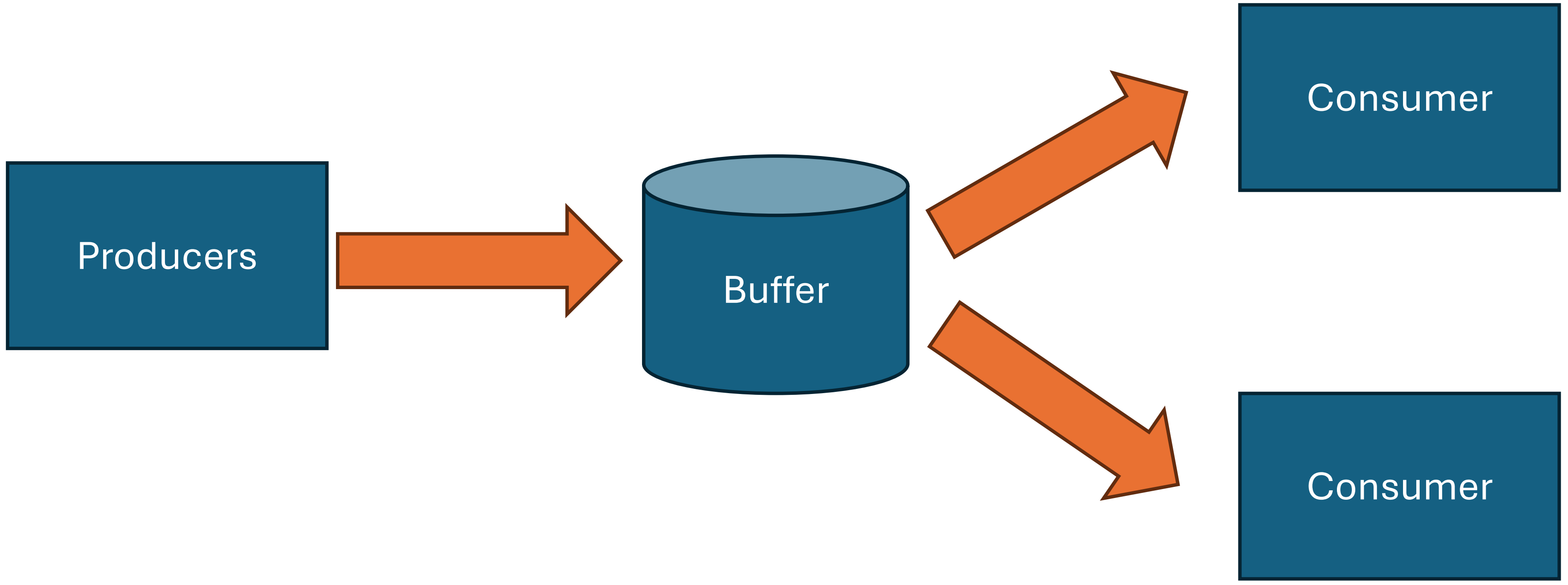
```
void * consumer(void * arg) {
        while (1) {
                mutex_lock(&m);
                while (numfull == 0)
                        cond_wait(&cond, &m);
                int tmp = do_get();
                cond_signal(&cond);
                mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```



Why is this useful?



Why is this useful?



Condition Variables: Fairness

Which thread to wake?

wait (cond_t, mutex_t)

- Assumes lock held prior to call
- Sleeps and releases locks
- Reacquires lock on waking

signal (cond_t)

- Wake one thread
- Else do nothing

broadcast (cond_t)

- Wake all threads
- Else do nothing

Example: Code

```
void * producer(void * arg) {
        for (int i=0; i<loops; i++) {
                mutex_lock(&m);
                while (numfull == max)
                        cond_wait(&full, &m);
                do_fill(i);
                cond_signal(&empty);
                mutex_unlock(&m);
        }
}
```

```
void * consumer(void * arg) {
        while (1) {
                mutex_lock(&m);
                while (numfull == 0)
                        cond_wait(&empty, &m);
                int tmp = do_get();
                cond_signal(&full);
                mutex_unlock(&m);
                printf("%d\n", tmp);
        }
}
```

Maybe have two condition variables?

Condition Variables Summary

Notes:

- Always do wait/signal while holding a mutex
- Always have a separate concept of ‘state’
- Whenever a thread wakes, recheck

Summary

Concurrency

- Locks
- Lock Implementation
  - Data Structures
- Condition Variables

Questions?