

# CS 318 Principles of Operating Systems

Fall 2022

## Lecture 6: Synchronization



JOHNS HOPKINS  
WHITING SCHOOL  
of ENGINEERING

**Prof. Ryan Huang**

# Before we start...: Too Much Milk

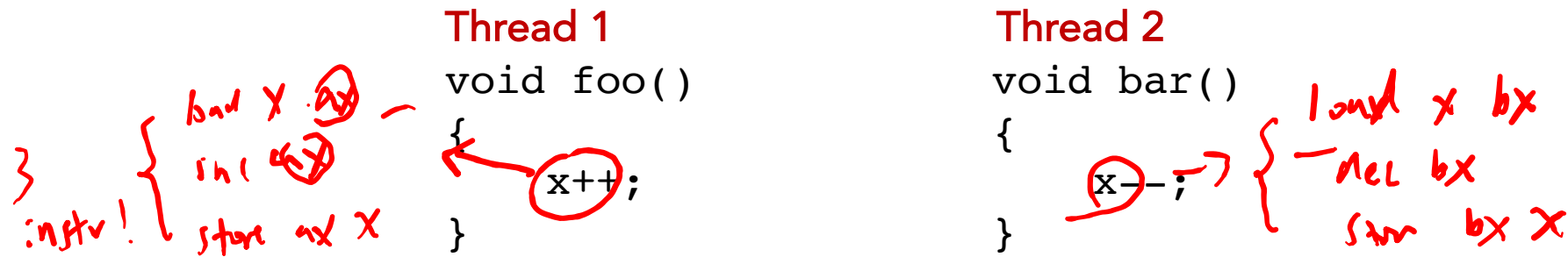
---

	Alice	Bob
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

---

# Before we start...: exercise #1

x is a global variable initialized to 0



After thread 1 and thread 2 finishes, what is the value of x?

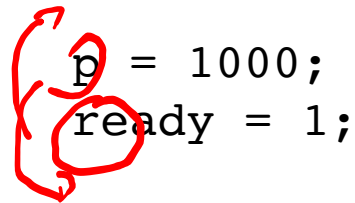
- could be 0, 1, -1
- Why?

# Before we start...: exercise #2

```
int p = 0, ready = 0;
```

Processor #1

```
p = 1000;  
ready = 1;
```



Processor #2

```
while (!ready);  
→ use(p);
```

**What value of `p` is passed to `use`?**

- could be 0, 1000
- Why?

**What if `p` holds an address?**

# Synchronization Motivation

## Threads cooperate in multithreaded programs

- To share resources, access shared data structures
- To coordinate their execution

## For correctness, we need to control this cooperation

- Thread schedule is **non-deterministic** (i.e., behavior changes when re-run program)
  - Scheduling is not under program control
  - Threads **interleave executions arbitrarily** and at different rates
- Multi-word operations are not atomic
- Compiler/hardware instruction reordering

# Shared Resources

**We initially focus on controlling access to shared resources**

## Basic problem

- If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior

**Over the next couple of lectures, we will look at**

- Mechanisms to control access to shared resources
  - Locks, mutexes, semaphores, monitors, condition variables, etc.
- Patterns for coordinating accesses to shared resources
  - Bounded buffer, producer-consumer, etc.

# Classic Example: Bank Account Balance

Implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {  
    balance = get_balance(account);  
    balance = balance - amount;  
    put_balance(account, balance);  
    return balance;  
}
```

Suppose that you and your significant other share a bank account with a balance of \$1000

Then you each go to separate ATM machines and simultaneously withdraw \$100 from the account

# Example Continued

We'll represent the situation by creating a separate thread for each person to do the withdrawals

These threads run on the same bank server:

```
withdraw (account, amount) {  
  ① balance = get_balance(account);  
  ④ balance = balance - amount;  
  put_balance(account, balance);  
  return balance;  
}
```

```
withdraw (account, amount) {  
  ② balance = get_balance(account);  
  ③ balance = balance - amount;  
  put_balance(account, balance);  
  return balance;  
}
```

**What's the problem with this implementation?**

- Think about potential schedules of these two threads



# Interleaved Schedules

The problem is that the execution of the two threads can be

**interleaved:**

**Execution  
sequence  
seen by CPU**

```
balance = get_balance(account);  
balance = balance - amount;
```

```
balance = get_balance(account);  
balance = balance - amount;  
put_balance(account, balance);
```

```
put_balance(account, balance);
```

**Context switch**

**What is the balance of the account now?**

**Is the bank happy with our implementation?**

# How Interleaved Can It Get?

How contorted can the interleavings be?

We'll assume that the only atomic operations are instructions

- e.g., reads and writes of words
- the hardware may not even give you that!

We'll assume that a **context switch can occur at any time**

We'll assume that **you can delay a thread as long as you like as long as it's not delayed forever**

```
..... get_balance(account);  
balance = get_balance(account);  
balance = .....  
balance = balance - amount;  
balance = balance - amount;  
put_balance(account, balance);  
put_balance(account, balance);
```

# Shared Resources

**Problem: concurrent threads accessed a **shared resource** without any **synchronization****

- Known as a **race condition**

**We need mechanisms to control access to these shared resources in the face of concurrency**

- So we can reason about how the program will operate

**Our example was updating a shared bank account**

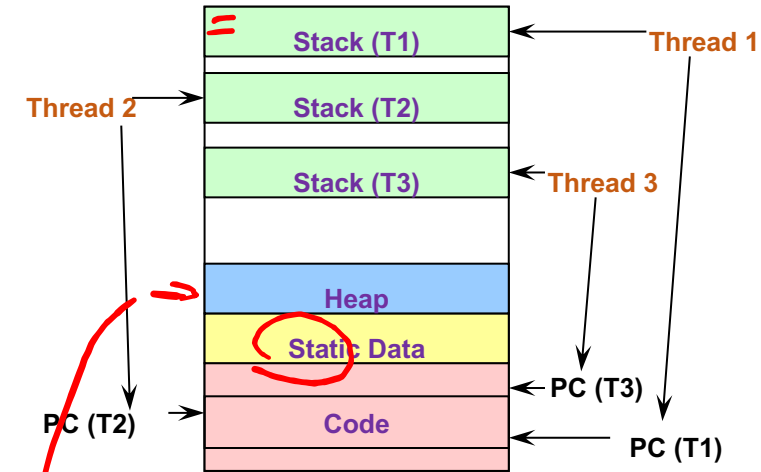
**Also apply to **any shared data structure****

- Buffers, queues, lists, hash tables, etc.

# When Are Resources Shared?

## Local variables are **not shared** (private)

- Refer to data on the stack
- Each thread has its own stack
- **Never** pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2



## Global variables and static objects are **shared**

- Stored in the static data segment, accessible by any thread

## Dynamic objects and other heap objects are **shared**

- Allocated from heap with **malloc**/free or **new**/delete

# Mutual Exclusion

We want to use **mutual exclusion** to synchronize access to shared resources

- This allows us to have larger atomic blocks

Code that uses mutual exclusion to synchronize its execution is called a **critical section**

- Only one thread at a time can execute in the critical section
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter
- Example: sharing your bathroom with housemates

**What requirements would you place on a critical section?**

# Critical Section Requirements

## 1) Mutual exclusion (mutex)

- If one thread is in the critical section, then no other is

## 2) Progress

- If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
- A thread in the critical section will eventually leave it

## 3) Bounded waiting (no starvation)

- If some thread T is waiting on the critical section, then T will eventually enter the critical section

## 4) Performance

- The overhead of entering and exiting the critical section is small with respect to the work being done within it

# About Requirements

There are three kinds of requirements that we'll use

**Safety property:** nothing bad happens

- Mutex

**Liveness property:** something good happens

- Progress, Bounded Waiting

**Performance requirement**

- Performance

**Properties hold for each run, while performance depends on all the runs**

- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!)

# Too Much Milk, Try #1

## Try #1: leave a note

```
if (milk == 0) {           // if no milk
    if (note == 0) {      // if no note
        note = 1;        // leave note
        milk++;          // buy milk
        note = 0;        // remove note
    }
}
```

What can go wrong?



# Too Much Milk, Try #2

## Try #1: leave a note

Alice

```
if (milk == 0) {
```

```
    if (note == 0) {  
        note = 1;  
        milk++;  
        note = 0;  
    }  
}
```

Bob

```
if (milk == 0) {  
    if (note == 0) {  
        note = 1;  
        milk++;  
        note = 0;  
    }  
}
```

# Too Much Milk, Try #2

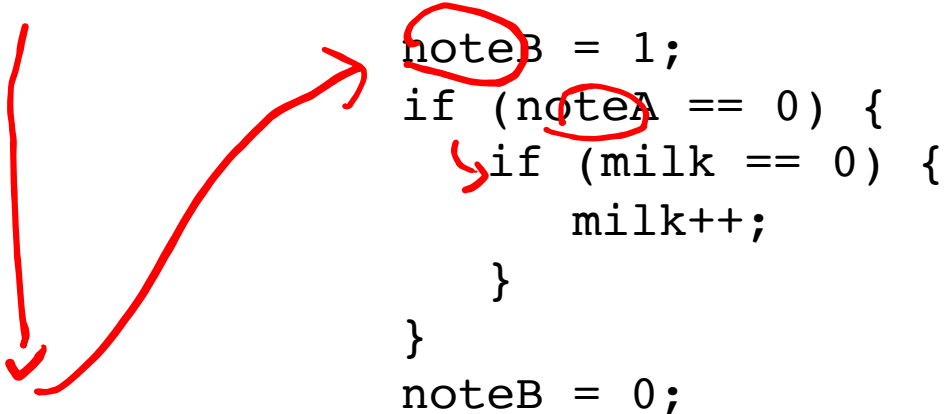
## Try #2: leave two notes

Alice

```
noteA = 1;
if (noteB == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteA = 0;
```

Bob

```
noteB = 1;
if (noteA == 0) {
    if (milk == 0) {
        milk++;
    }
}
noteB = 0;
```



Is this safe?

Does it ensure liveness?

# Too Much Milk, Try #3

## Try #3: monitoring note

Alice

```
noteA = 1;  
while (noteB == 1);  
if (milk == 0) {  
    milk++;  
}  
noteA = 0;
```

Bob

```
noteB = 1;  
if (noteA == 0) {  
    if (milk == 0) {  
        milk++;  
    }  
}  
noteB = 0;
```

Is this safe?

Does it ensure liveness?

# Mechanisms For Building Critical Sections

## Atomic read/write

- Can it be done?

## Locks

- Primitive, minimal semantics, used to build others

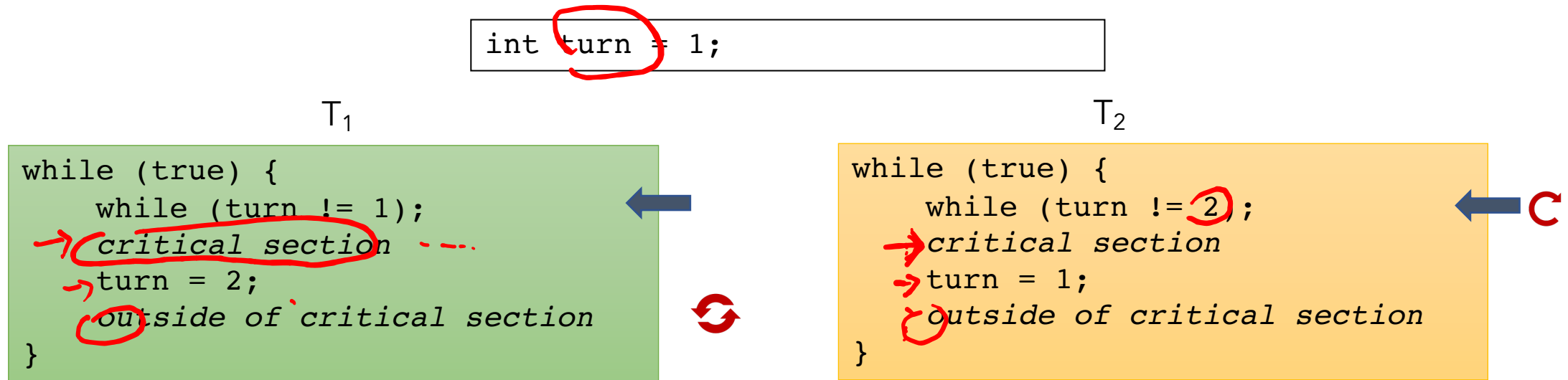
## Semaphores

- Basic, easy to get the hang of, but hard to program with

## Monitors

- High-level, requires language support, operations implicit

# Mutex with Atomic R/W: Try #1



This is called alternation

Does it satisfy the safety requirement?

- Yes

Does it satisfy the liveness requirement?

- No, T<sub>1</sub> can go into infinite loop outside of the critical section preventing T<sub>2</sub> from entering

# Mutex with Atomic R/W: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
  ↓ try1 = true;
  → turn = 2;
  → while (try2 && turn != 1) ;
  ↓ critical section
  → try1 = false;
  → outside of critical section ↻
}
```

```
while (true) {
  → try2 = true;
  → turn = 1;
  → while (try1 && turn != 2) ;
  → critical section
  → try2 = false;
  → outside of critical section
}
```

Does it satisfy the safety requirement?

Does it satisfy the liveness requirement?

# Mutex with Atomic R/W: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
  → {¬ try1 ∧ (turn == 1 ∨ turn == 2)}
  1   try1 = true;
  { try1 ∧ (turn == 1 ∨ turn == 2) }
  2   turn = 2;
  { try1 ∧ (turn == 1 ∨ turn == 2) }
  3   while (try2 && turn != 1);
  → { try1 ∧ (turn == 1 ∨ ¬ try2 ∨
  → (try2 ∧ (yellow at 6 or at 7))) }
  → critical section
  4   try1 = false;
  {¬ try1 ∧ (turn == 1 ∨ turn == 2)}
  outside of critical section
}
```

```
while (true) {
  {¬ try2 ∧ (turn == 1 ∨ turn == 2)}
  5   try2 = true;
  { try2 ∧ (turn == 1 ∨ turn == 2) }
  6   turn = 1;
  { try2 ∧ (turn == 1 ∨ turn == 2) }
  7   while (try1 && turn != 2);
  → { try2 ∧ (turn == 2 ∨ ¬ try1 ∨
  → (try1 ∧ (green at 2 or at 3))) }
  → critical section
  8   try2 = false;
  {¬ try2 ∧ (turn == 1 ∨ turn == 2)}
  outside of critical section
}
```

(green at 4) ∧ (yellow at 8) ⇒ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7)))  
∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (green at 2 or at 3)))  
... ⇒ (turn == 1 ∧ turn == 2)

# Locks

**A lock is an object in memory providing two operations**

- `acquire()`: wait until lock is free, then take it to enter a C.S
- `release()`: release lock to leave a C.S, waking up anyone waiting for it

**Threads pair calls to acquire and release**

- Between `acquire/release`, the thread **holds** the lock
- `acquire` does not return until any previous holder releases
- What can happen if the calls are not paired?

**Locks can spin (a spinlock) or block (a mutex)**

- Can break apart Peterson's to implement a spinlock



# Too Much Milk, Try #4

## Try #4: lock

Alice

```
lock.acquire();  
if (milk == 0) {  
    milk++;  
}  
lock.release();
```

Bob

```
lock.acquire();  
if (milk == 0) {  
    milk++;  
}  
lock.release();
```

# Using Locks

```
withdraw (account, amount) {  
    acquire(lock);  
    {  
        balance = get_balance(account);  
        balance = balance - amount;  
        put_balance(account, balance);  
    }  
    release(lock);  
    return balance;  
}
```

Critical  
Section

$T_1$

```
acquire(lock);  
balance = get_balance(account);  
balance = balance - amount;
```

**acquire(lock);**

put\_balance(account, balance);  
**release(lock);**

```
balance = get_balance(account);  
balance = balance - amount;  
put_balance(account, balance);  
release(lock);
```

- What happens when green tries to acquire the lock?
- Why is the "return" outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

# Implementing Locks (1)

How do we implement locks? Here is one attempt:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock->held);
    ↓ lock->held = 1;
}
void release (lock) {
    lock->held = 0;
}
```

busy-wait (spin-wait)  
for lock to be released

This is called a **spinlock** because a thread spins waiting for the lock to be released

Does this work?

# Implementing Locks (2)

No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}
void release(lock) {
    lock->held = 0;
}
```

A context switch can occur here, causing a race condition

$T_2$

# Implementing Locks (3)

The problem is that the implementation of locks has critical sections, too!

How do we stop the recursion?

The implementation of acquire/release must be **atomic**

- An atomic operation is one which executes as though it could not be interrupted
- Code that executes "all or nothing"

How do we make them atomic?

Need help from hardware

- Atomic instructions (e.g., test-and-set)
- Disable/enable interrupts (prevents context switches)

# Atomic Instructions: Test-And-Set

The semantics of test-and-set are:

- Record the old value
- Set the value to indicate available
- Return the old value

```
bool test_and_set(bool *flag) {  
    bool old = *flag;  
    *flag = True;  
    return old;  
}
```

Hardware executes it atomically!

When executing test-and-set on "flag"

- What is value of flag afterwards if it was initially False? True?
- What is the return result if flag was initially False? True?

Other similar flavor atomic instructions: xchg, CAS

# Using Test-And-Set

Here is our lock implementation with test-and-set:

```
struct lock {
    int held = 0;
}
void acquire(lock) {
    while (test-and-set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```

*Handwritten annotations:*  
- Red underline under `held` in the struct definition.  
- Red arrow pointing to the `test-and-set` function call in the `acquire` function.  
- Red arrow pointing to the `held` parameter in the `test-and-set` call.  
- Red circle around the `held` parameter in the `test-and-set` call.  
- Red text "return False" with an arrow pointing to the `test-and-set` call.

When will the while return? What is the value of held?

What about multiprocessors? *Yes*

Implement it with xchg, Compare-And-Swap

# Problems with Spinlocks

**The problem with spinlocks is that they are wasteful**

- If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor)

**How did the lock holder give up the CPU in the first place?**

- Lock holder calls yield or sleep
- Involuntary context switch

**Only want to use spinlocks as primitives to build higher-level synchronization constructs**



# Disabling Interrupts

Another implementation of acquire/release is to disable interrupts:

```
struct lock {  
    }  
void acquire(lock) {  
    disable interrupts;  
}  
void release(lock) {  
    enable interrupts;  
}
```

Note that there is no state associated with the lock

Can two threads disable interrupts simultaneously?

# On Disabling Interrupts

Disabling interrupts blocks notification of external events that could trigger a **context switch (e.g., timer)**

- This is what Pintos uses as its primitive

In a “real” system, this is only available to the kernel

- Why?

**Disabling interrupts is insufficient on a multiprocessor**

- Interrupts are only disabled on a per-core basis
- Back to atomic instructions

Like spinlocks, only want to disable interrupts to implement higher-level **synchronization primitives**

- Don't want interrupts disabled between acquire and release

# Summarize Where We Are

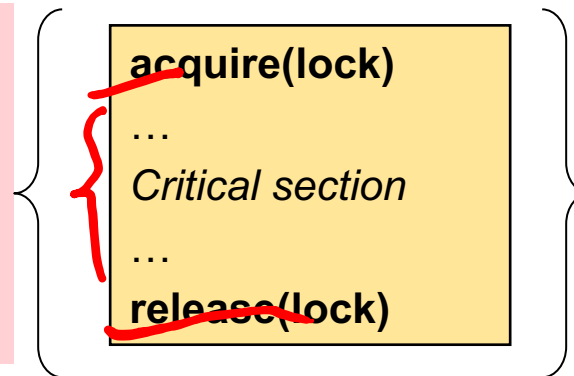
**Goal:** Use **mutual exclusion** to protect **critical sections** of code that access **shared resources**

**Method:** Use locks (either spinlocks or disable interrupts)

**Problem:** Critical sections (CS) can be long

## → Spinlocks

- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin, greater the chance for lock holder to be interrupted



## Disabling Interrupts:

- Disabling interrupts for long periods of time can miss or delay important events (e.g., timer, I/O)

# Higher-Level Synchronization

**Spinlocks and disabling interrupts are useful only for very short and simple critical sections**

- Wasteful otherwise
- These primitives are “primitive” – don’t do anything besides mutual exclusion

**Need higher-level synchronization primitives that:**

- Block waiters
- Leave interrupts enabled within the critical section

**All synchronization requires atomicity**

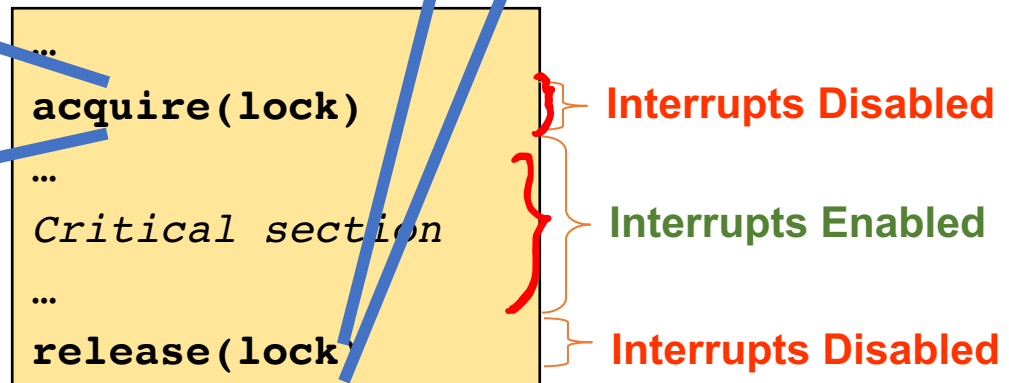
**So we’ll use our “atomic” locks as primitives to implement them**

# Implementing Locks (4)

## Block waiters, interrupts enabled in critical sections

```
struct lock {  
    int held = 0;  
    queue Q;  
}  
void acquire(lock) {  
    Disable interrupts;  
    while (lock->held) {  
        → put current thread on lock Q;  
        block current thread;  
    }  
    lock->held = 1; ↓  
    Enable interrupts; →  
}
```

```
void release(lock) {  
    Disable interrupts;  
    if (Q) remove waiting thread;  
    unblock waiting thread;  
    → lock->held = 0;  
    ↓ Enable interrupts;  
}
```



See Pintos [threads/synch.c](#): sema\_down/up

# Summary

**Why we need synchronizations**

**Critical sections**

**Simple algorithms to implement critical sections**

**Locks**

**Lock implementations**

# Next Time...

**Read Chapters 30, 31**