

Operating Systems (Honor Track)

Synchronization 3: Atomic Instructions, Monitors

Xin Jin

Spring 2022

Motivating Example: “Too Much Milk”

- Great thing about OS’s – analogy between problems in OS and problems in real life
 - Help you understand real life problems better
 - But, computers are much stupider than people
- Example: People need to coordinate:

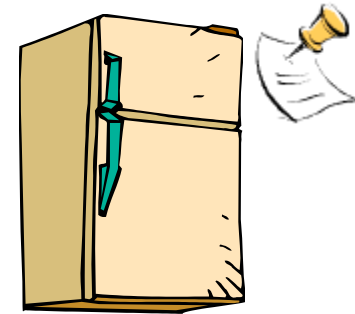


Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
 - Leave a note before buying (kind of “lock”)
 - Remove note after buying (kind of “unlock”)
 - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy milk;  
        remove note;  
    }  
}
```



- Result?
 - Still too much milk **but only occasionally!**
 - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails **intermittently**
 - Makes it really hard to debug...
 - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
 - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;  
if (noMilk) {  
    if (noNote) {  
        buy milk;  
    }  
}  
remove Note;
```

- What happens here?
 - Well, with human, probably nothing bad
 - With computer: no one ever buys milk



Too Much Milk Solution #2

- How about labeled notes?
 - Now we can leave note before checking
- Algorithm looks like this:

```
Thread A  
leave note A;  
if (noNote B) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note A;
```

```
Thread B  
leave note B;  
if (noNoteA) {  
    if (noMilk) {  
        buy Milk;  
    }  
}  
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
 - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
 - **Extremely unlikely** this would happen, but will at worse possible time
 - Probably something like this in UNIX

Too Much Milk Solution #3

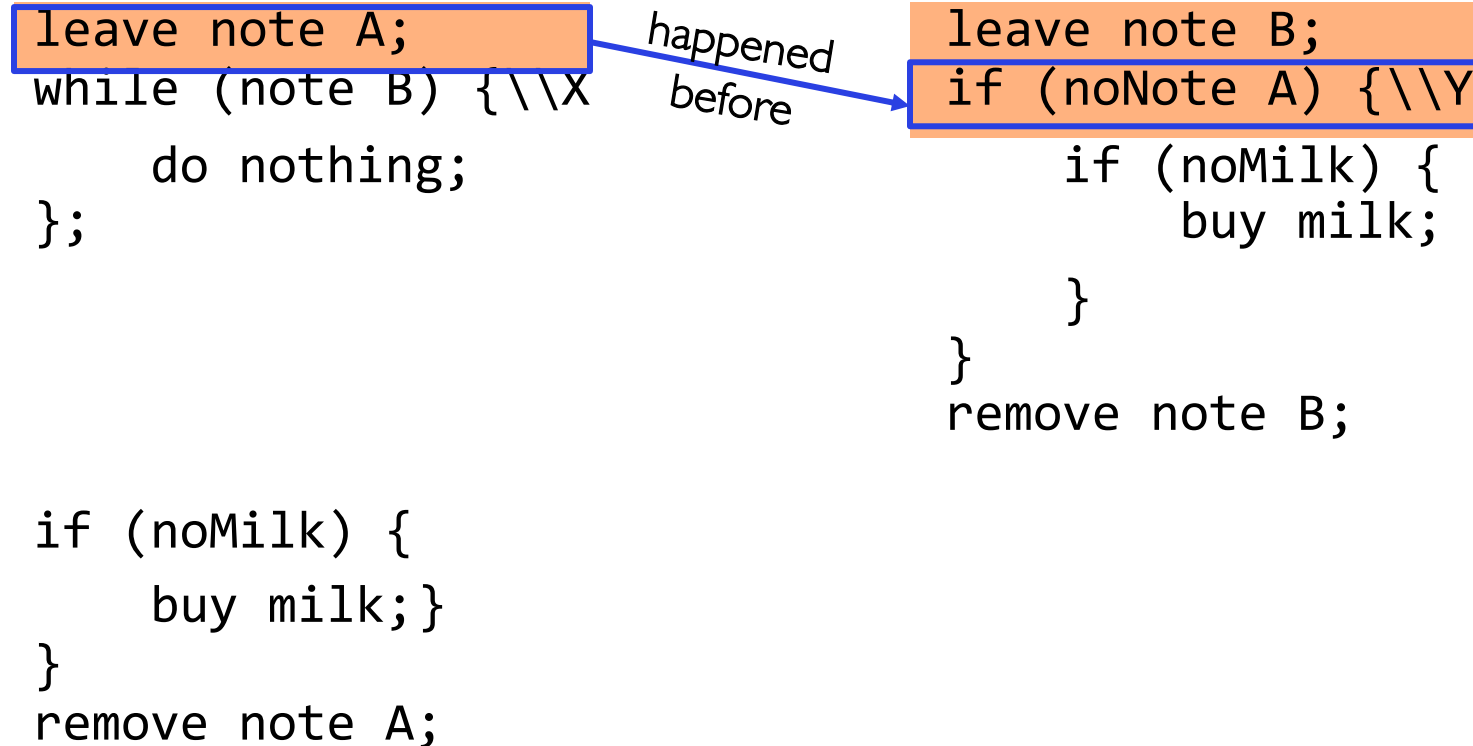
- Here is a possible two-note solution:

<u>Thread A</u>	<u>Thread B</u>
leave note A;	leave note B;
while (note B) { \\X	if (noNote A) { \\Y
do nothing;	if (noMilk) {
}	buy milk;
if (noMilk) {	}
buy milk;	}
}	remove note B;
remove note A;	

- Does this work? **Yes**. Both can guarantee that:
 - It is safe to buy, or
 - Other will buy, ok to quit
- At X:
 - If no note B, safe for A to buy,
 - Otherwise wait to find out what will happen
- At Y:
 - If no note A, safe for B to buy
 - Otherwise, A is either buying or waiting for B to quit

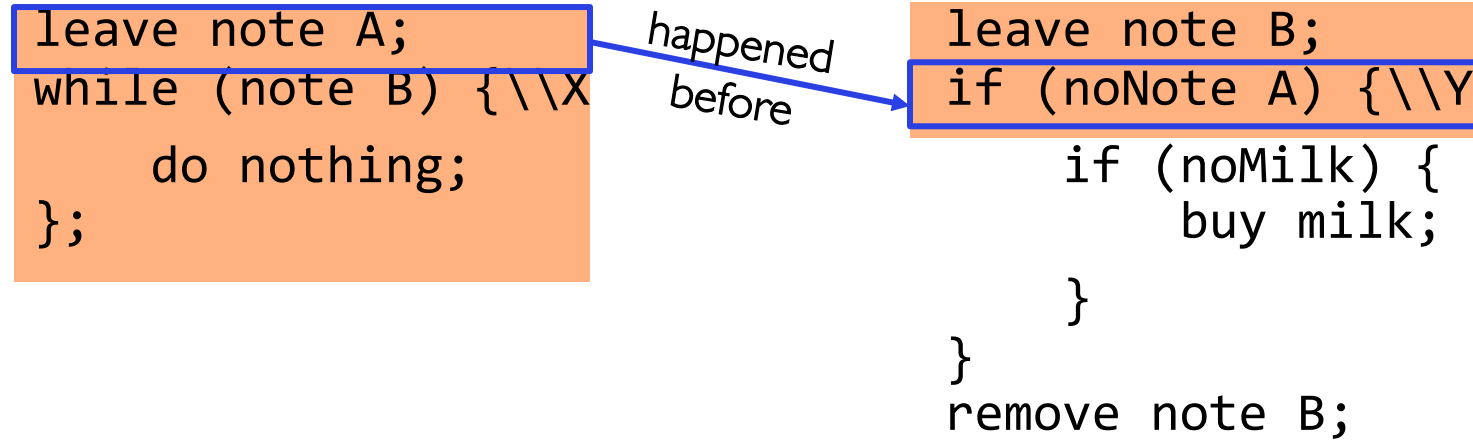
Case 1

- “leave note A” happens before “if (noNote A)”



Case 1

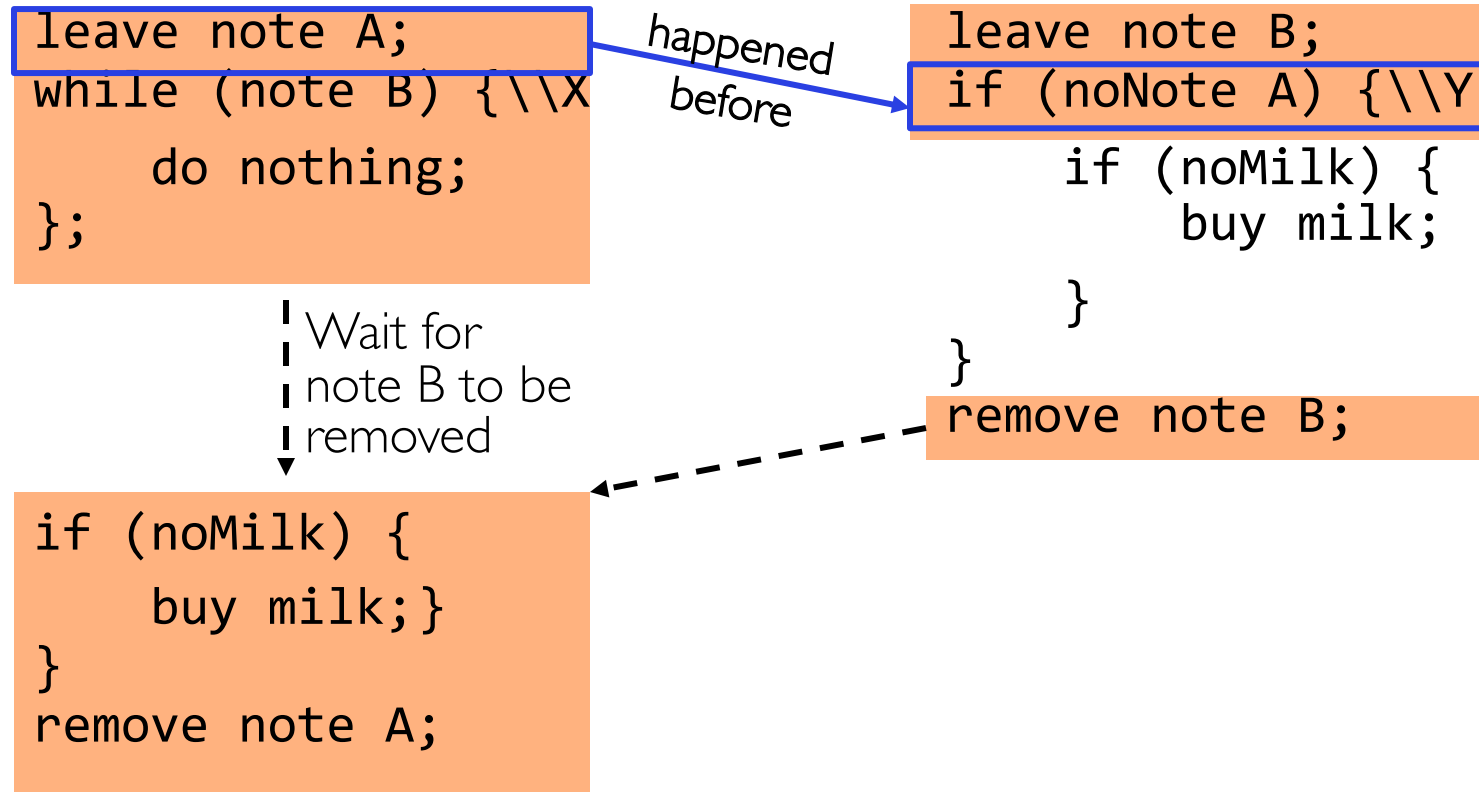
- “leave note A” happens before “if (noNote A)”



```
if (noMilk) {  
    buy milk;  
}  
remove note A;
```

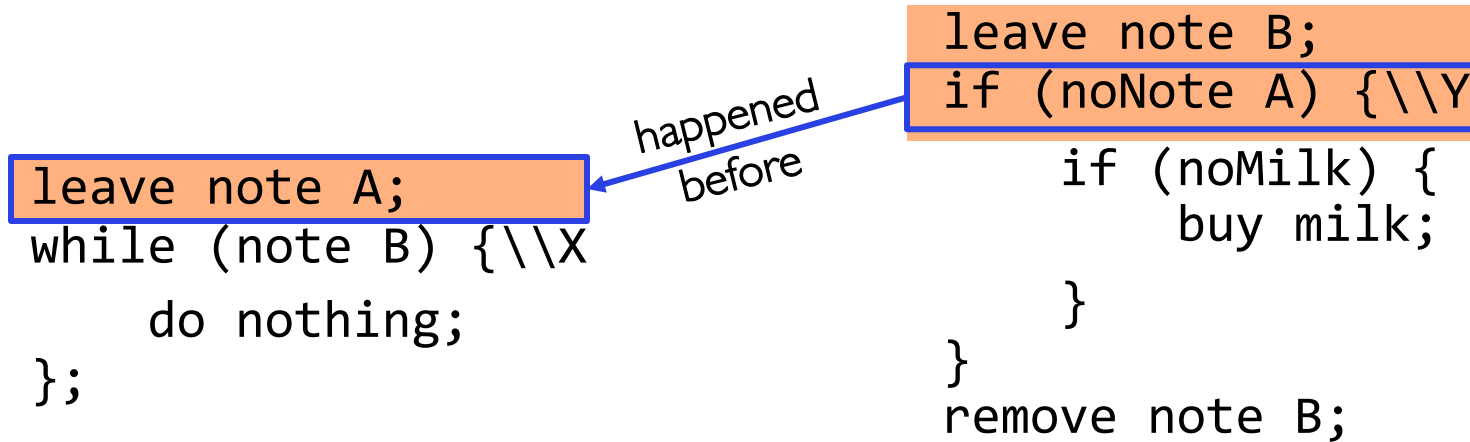

Case 1

- “leave note A” happens before “if (noNote A)”



Case 2

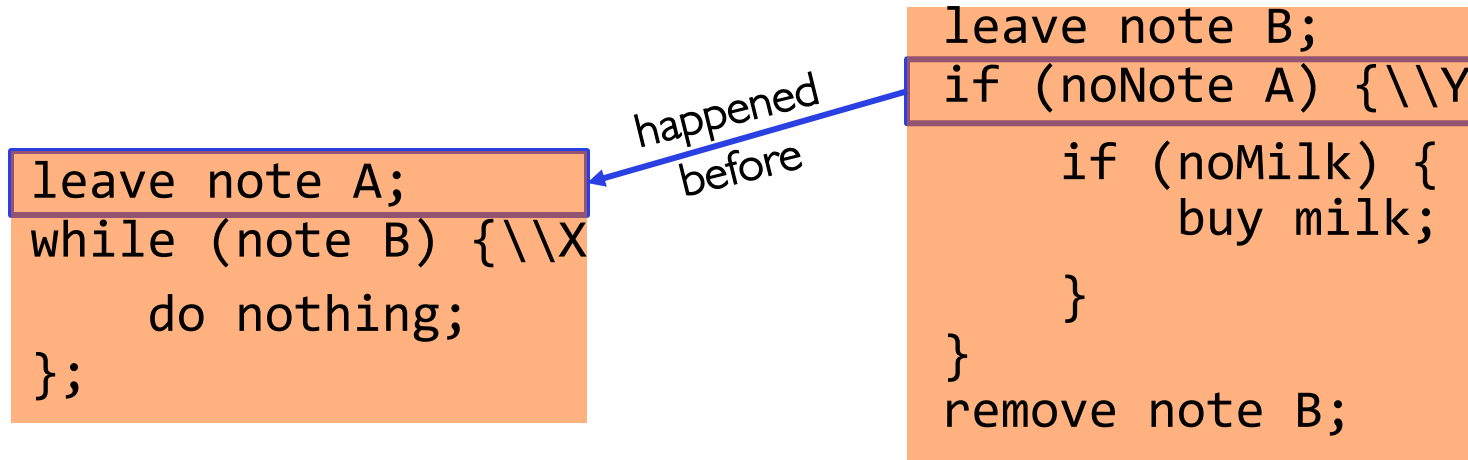
- “if (noNote A)” happens before “leave note A”



```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

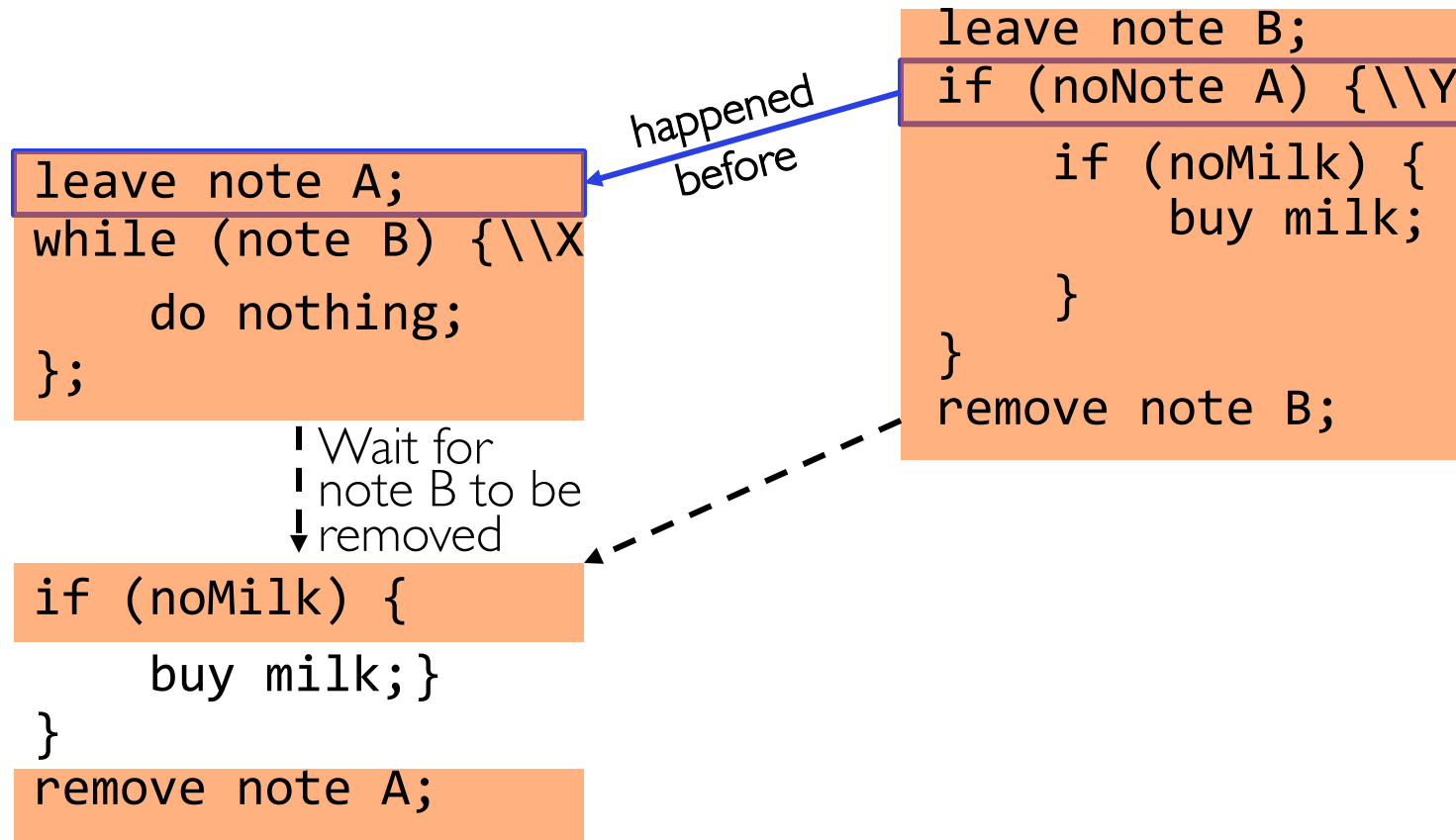
- “if (noNote A)” happens before “leave note A”



```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- “if (noNote A)” happens before “leave note A”



Solution #3 discussion

- Our solution protects a single “Critical-Section” piece of code for each thread:

```
if (noMilk) {  
    buy milk;  
}
```

- Solution #3 works, but it’s really unsatisfactory
 - Really complex – even for this simple an example
 - » Hard to convince yourself that this really works
 - A’s code is different from B’s – what if lots of threads?
 - » Code would have to be slightly different for each thread
 - While A is waiting, it is consuming CPU time
 - » This is called “busy-waiting”
- There’s got to be a better way!
 - Have hardware provide higher-level primitives than atomic load & store
 - Build even higher-level programming abstractions on this hardware support

Too Much Milk: Solution #4?

- Recall our target lock interface:
 - `acquire(&milklock)` – wait until lock is free, then grab
 - `release(&milklock)` – Unlock, waking up anyone waiting
 - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
acquire(&milklock);  
if (nomilk)  
    buy milk;  
release(&milklock);
```

Back to: How to Implement Locks?

- **Lock**: prevents someone from doing something
 - Lock before entering critical section and before accessing shared data
 - Unlock when leaving, after accessing shared data
 - Wait if locked
 - » Important idea: all synchronization involves waiting
 - » Should *sleep* if waiting for a long time
- Atomic Load/Store: get solution like Milk #3
 - Pretty complex and error prone
- Hardware Lock instruction
 - Complexity?
 - » Done in the Intel 432
 - » Each feature makes HW more complex and slow



Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
 - Recall: dispatcher gets control in two ways.
 - » Internal: Thread does something to relinquish the CPU
 - » External: Interrupts cause dispatcher to take CPU
 - On a uniprocessor, can avoid context-switching by:
 - » Avoiding internal events
 - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }  
LockRelease { enable Ints; }
```
- Problems with this approach:
 - **Can't let user do this!** Consider following:

```
LockAcquire();  
While(TRUE) {;
```
 - Real-Time system—no guarantees on timing!
 - » Critical Sections might be arbitrarily long
 - What happens with I/O or other important events?
 - » “Reactor about to meltdown. Help?”



Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

```
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue;  
        place on ready queue;  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
 - Avoid interruption between checking and setting lock value
 - Otherwise two threads could think that they both have lock

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```



**Critical
Section**

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short
 - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
 - Critical interrupts taken in time!

Group Discussion

- Topic: Interrupt Re-enable in Going to Sleep
 - What about re-enabling ints when going to sleep?
 - Do we need to do it?
 - If so, where? If not, why?
- Discuss in groups of two to three students
 - Each group chooses a leader to summarize the discussion
 - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

Enable Position →

- Before Putting thread on the wait queue?

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

Enable Position →

- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```


Enable Position →

- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```


Enable Position 

- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!)

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

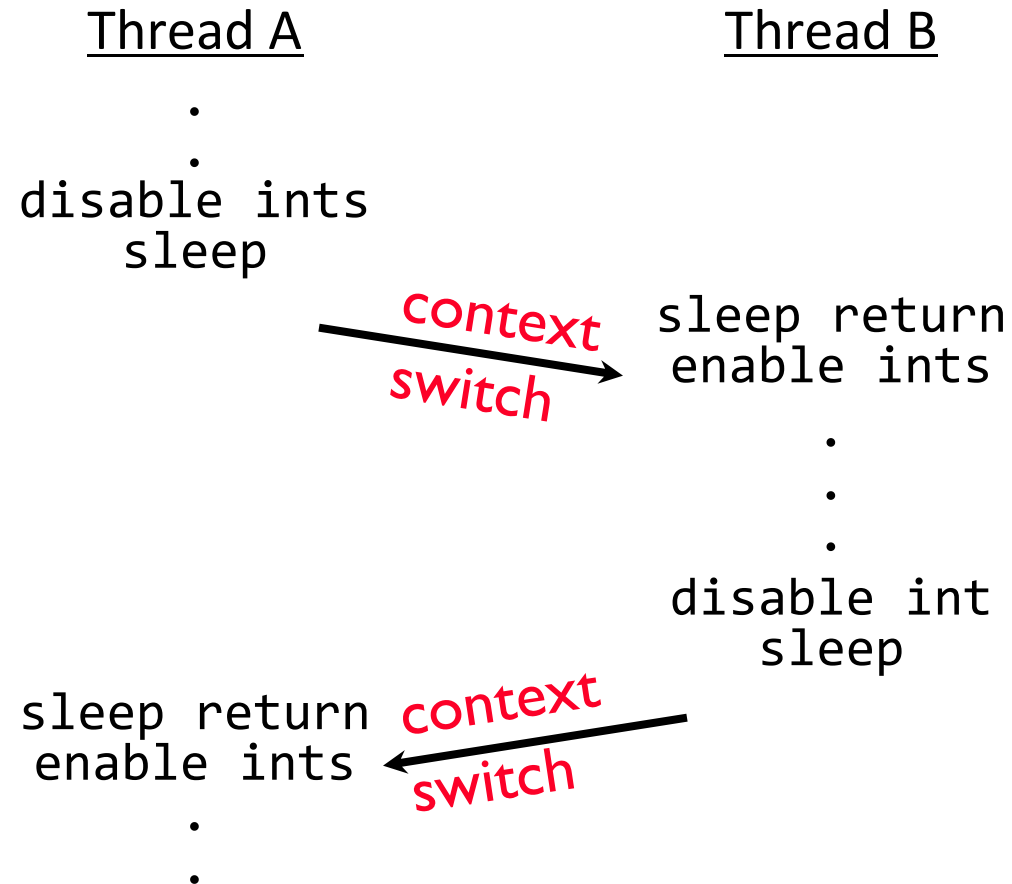
```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

Enable Position 

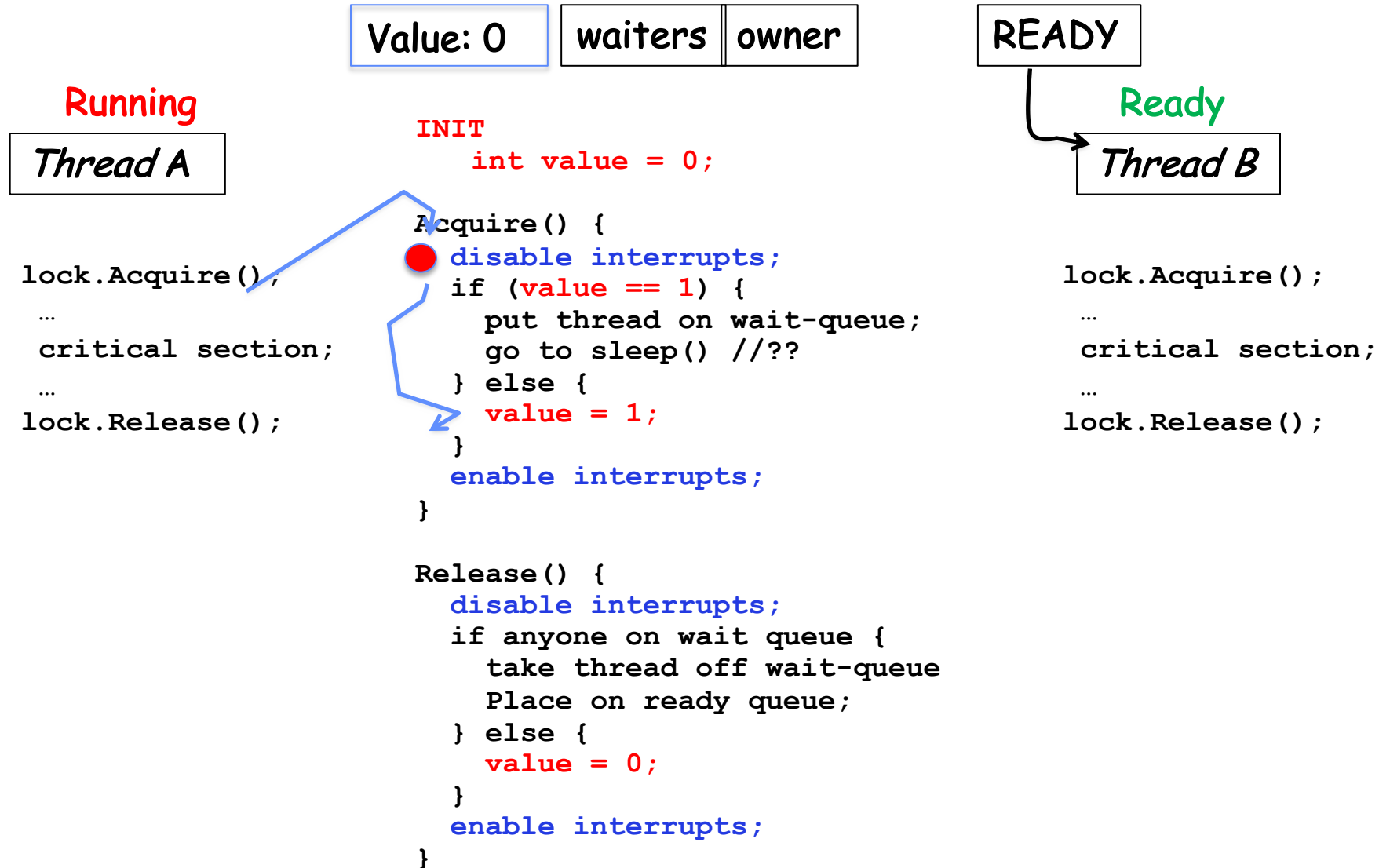
- Before Putting thread on the wait queue
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!)
- Want to put it after `sleep()`. But – how?

How to Re-enable After Sleep()?

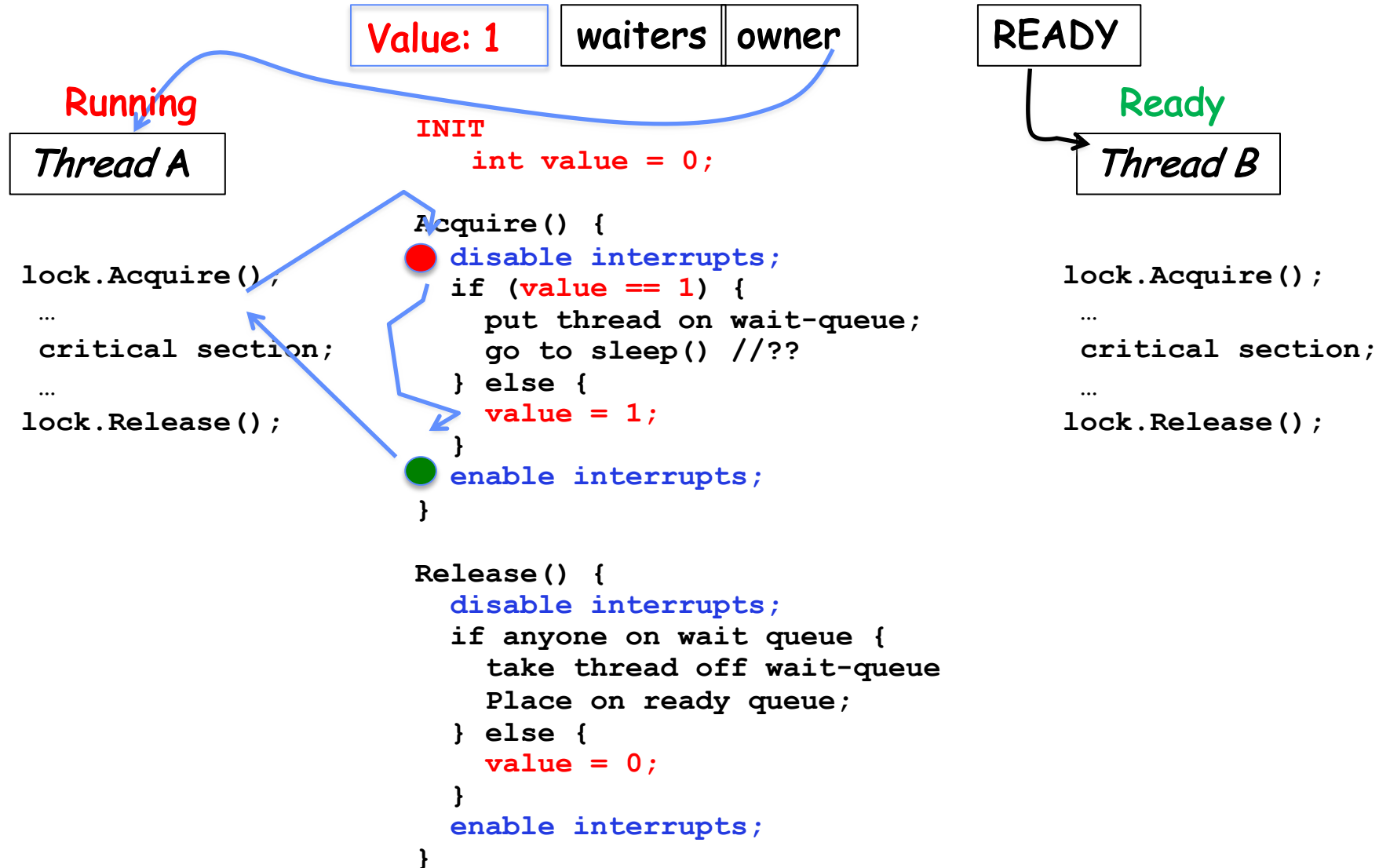
- In scheduler, since interrupts are disabled when you call sleep:
 - Responsibility of the next thread to re-enable ints
 - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



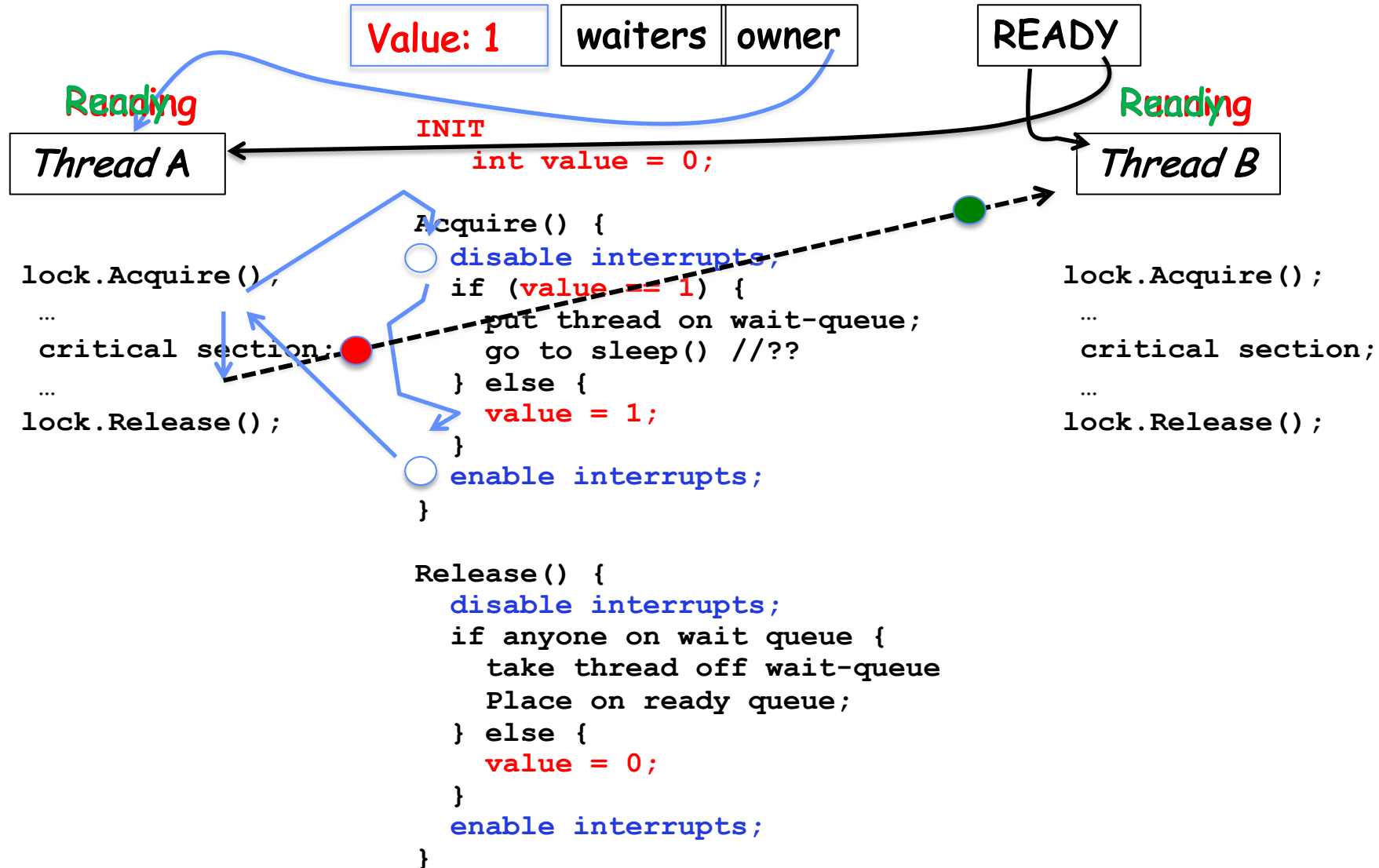
In-Kernel Lock: Simulation



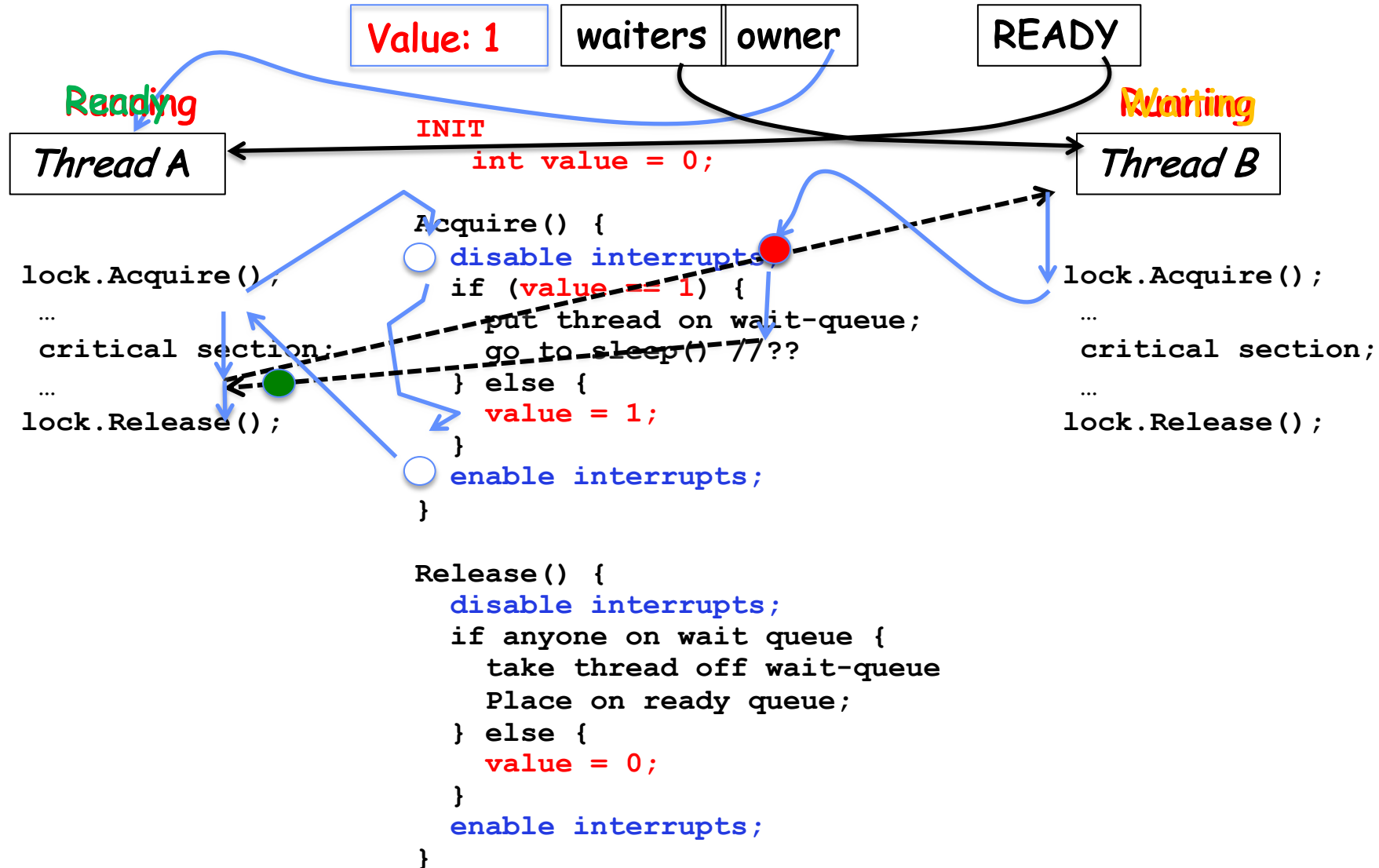
In-Kernel Lock: Simulation



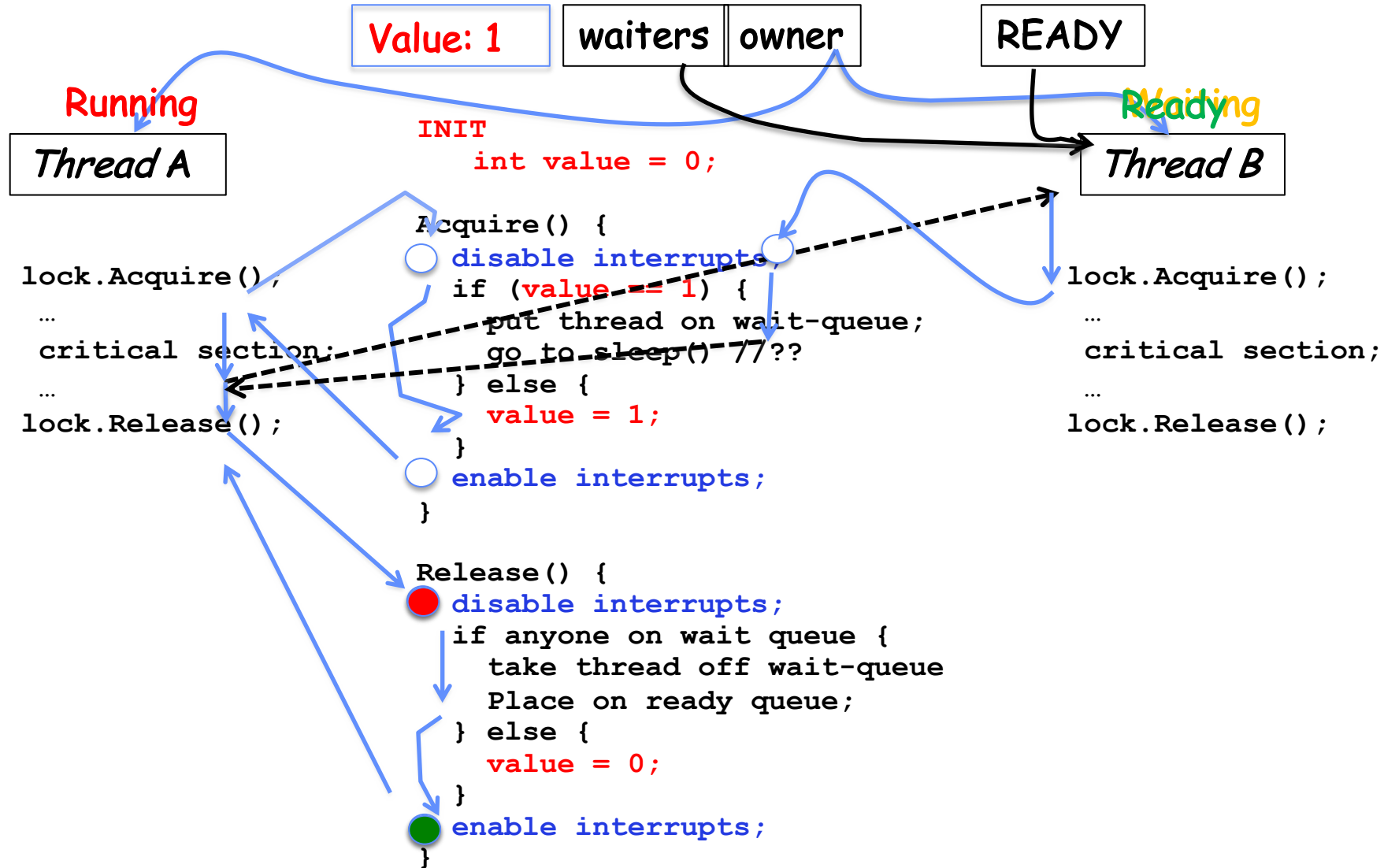
In-Kernel Lock: Simulation



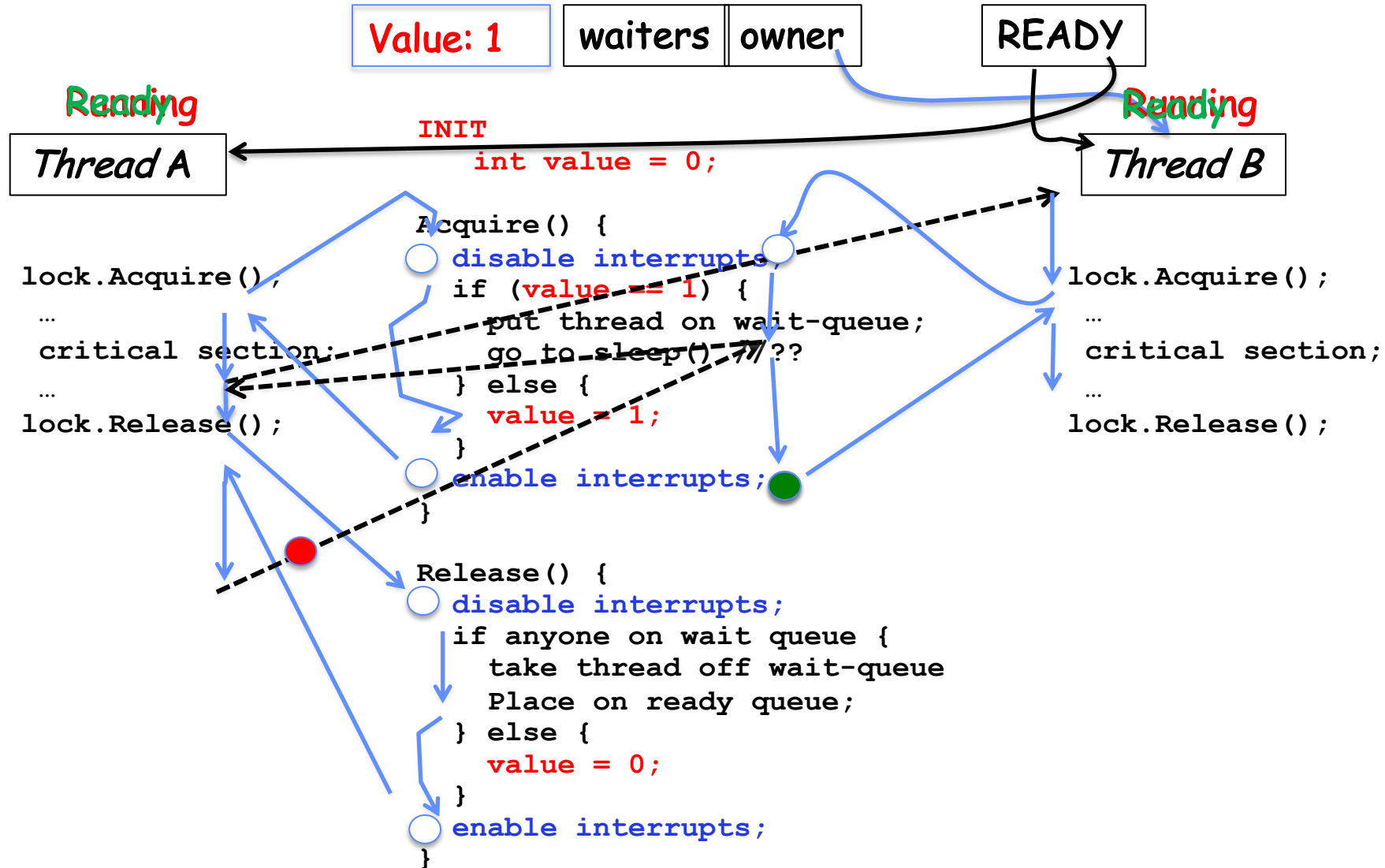
In-Kernel Lock: Simulation



In-Kernel Lock: Simulation



In-Kernel Lock: Simulation



Atomic Read-Modify-Write Instructions

- Problems with previous solution:
 - Can't give lock implementation to users
 - Doesn't work well on multiprocessor
 - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: **atomic instruction sequences**
 - These instructions read a value and write a new value atomically
 - Hardware is responsible for implementing this correctly
 - » on both uniprocessors (not too hard)
 - » and multiprocessors (requires help from cache coherence protocol)
 - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

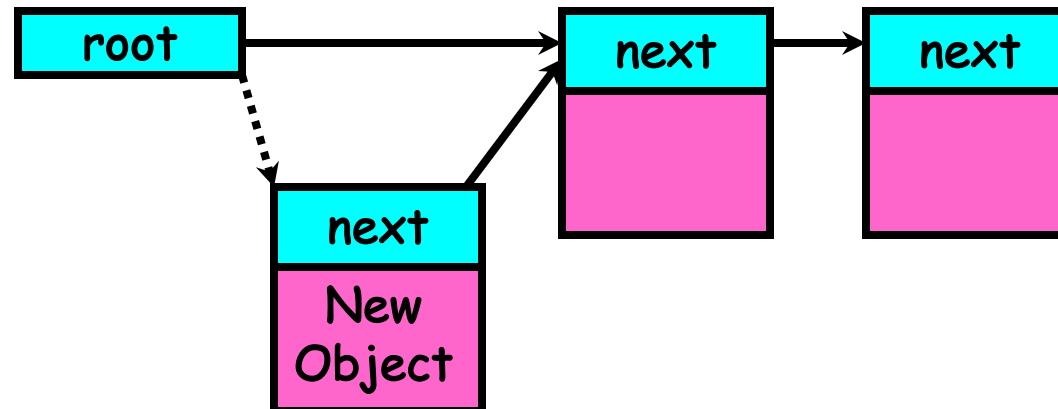
- `test&set (&address) {` `/* most architectures */`
 `result = M[address];` `// return result from "address" and`
 `M[address] = 1;` `// set value at "address" to 1`
 `return result;`
}
- `swap (&address, register) {` `/* x86 */`
 `temp = M[address];` `// swap register's value to`
 `M[address] = register;` `// value at "address"`
 `register = temp;`
}
- `compare&swap (&address, reg1, reg2) {` `/* 68000 */`
 `if (reg1 == M[address]) {` `// If memory still == reg1,`
 `M[address] = reg2;` `// then put reg2 => memory`
 `return success;`
 `} else {` `// Otherwise do not change memory`
 `return failure;`
 `}`
}

Using of Compare&Swap for queues

```
• compare&swap (&address, reg1, reg2) { /* 68000 */  
  if (reg1 == M[address]) {  
    M[address] = reg2;  
    return success;  
  } else {  
    return failure;  
  }  
}
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {  
  do {  
    ld r1, M[root] // repeat until no conflict  
    st r1, M[object] // Get ptr to current head  
  } until (compare&swap(&root, r1, object));  
}
```



Implementing Locks with test&set

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
    while (test&set(value)); // while busy
}
Release() {
    value = 0;
}
```

- Simple explanation:
 - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
 - If lock is busy, test&set reads 1 and sets value=1 (no change) It returns 1, so while loop continues.
 - When we set value = 0, someone else can get lock.
- **Busy-Waiting**: thread consumes cycles while waiting
 - For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)

Problem: Busy-Waiting for Lock

- Positives for this solution
 - Machine can receive interrupts
 - User code can use this lock
 - Works on a multiprocessor
- Negatives
 - This is very inefficient as thread will consume cycles waiting
 - Waiting thread may take cycles away from thread holding lock (no one wins!)
 - **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!
 - Priority Inversion problem with original Martian rover



Group Discussion

- Topic: Better Locks using test&set
 - Can you come up with a better solution that avoids or minimizes busy-awaiting
- Discuss in groups of two to three students
 - Each group chooses a leader to summarize the discussion
 - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

```
int value = 0; // Free
Acquire() {
    // while busy
    while (test&set(value));
}
Release() {
    value = 0;
}
```

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
 - Can't entirely, but can minimize!
 - Idea: only busy-wait to atomically check lock value

```
int guard = 0;  
int value = FREE;
```



```
Acquire() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep() & guard = 0;  
    } else {  
        value = BUSY;  
        guard = 0;  
    }  
}
```

```
Release() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if anyone on wait queue {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    guard = 0;  
}
```

- Note: sleep has to be sure to reset the guard variable
 - Why can't we do it just before or just after the sleep?

Recall: Locks using Interrupts vs. test&set

Compare to “disable interrupt” solution

```
int value = FREE;
```

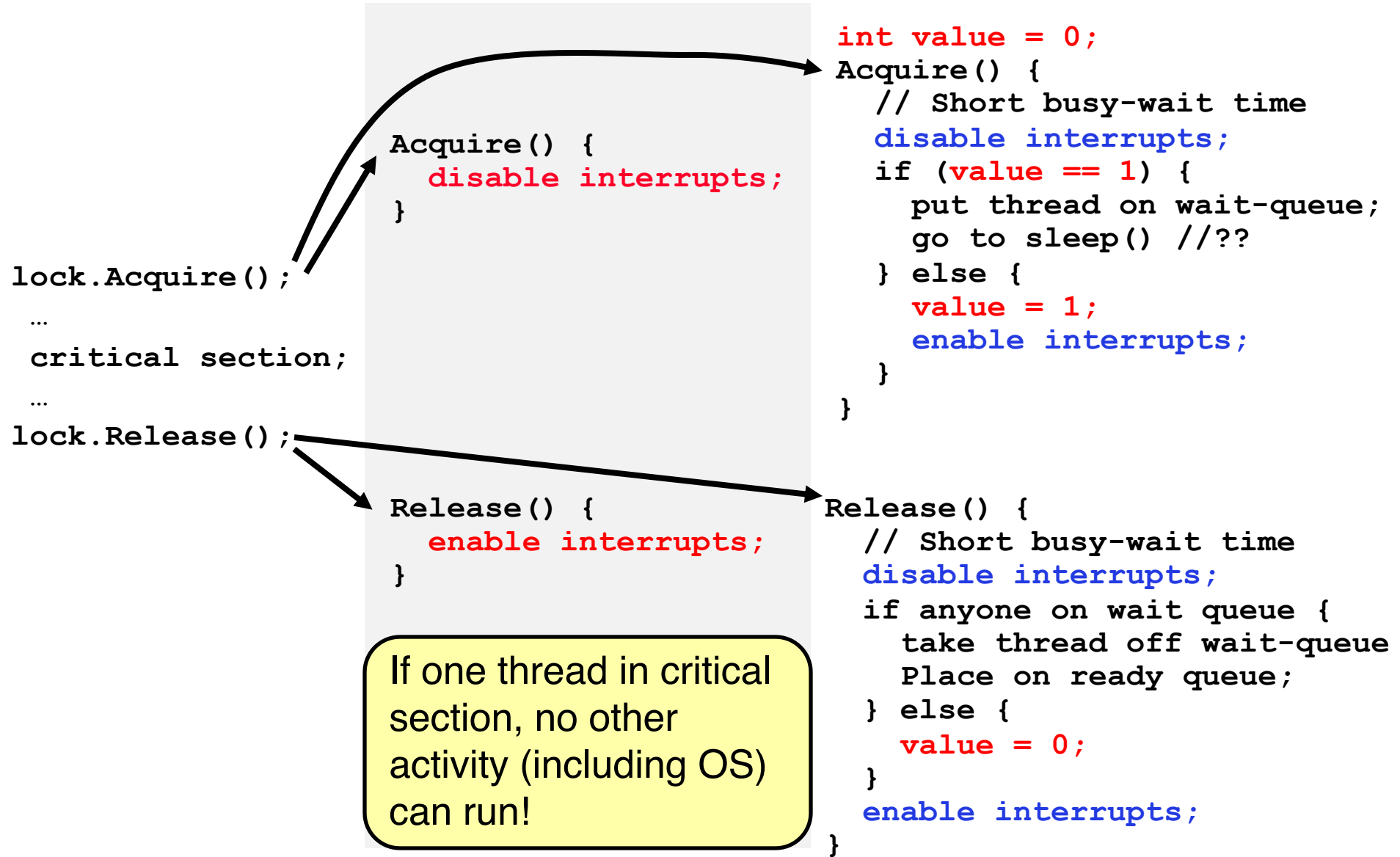


```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}  
  
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

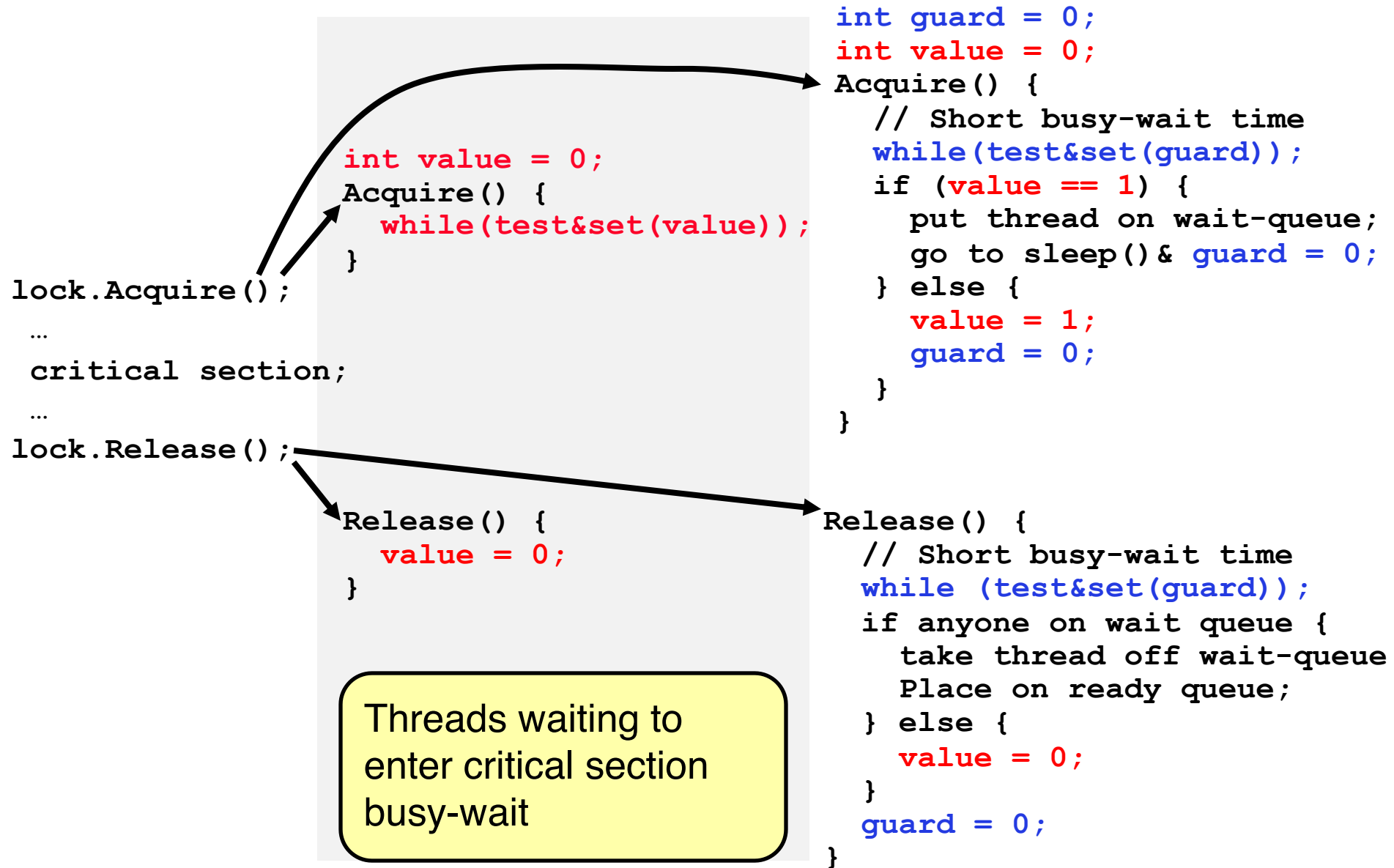
Basically we replaced:

- `disable interrupts` → `while (test&set(guard));`
- `enable interrupts` → `guard = 0;`

Recap: Locks using interrupts



Recap: Locks using test & set



Recall: Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
 - Everything is pretty painful if only atomic primitives are load and store
 - Need to provide primitives useful at user-level

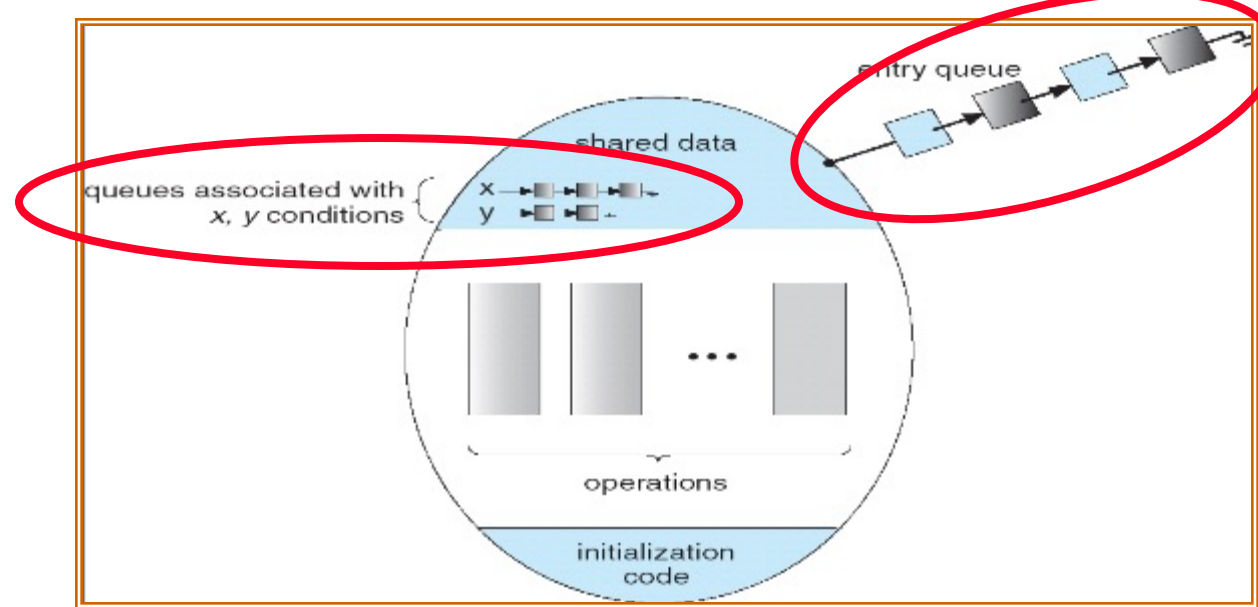
Semaphores are good but...Monitors are better!

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores or even with locks!
- Problem is that semaphores are dual purpose:
 - They are used for both mutex and scheduling constraints
 - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: **Monitor**: a **lock** and zero or more **condition variables** for managing concurrent access to shared data
 - Some languages like Java provide this natively
 - Most others use actual locks and condition variables
- A “Monitor” is a paradigm for concurrent programming!
 - Some languages support monitors explicitly

Condition Variables

- How do we change the consumer() routine to wait until something is on the queue?
 - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section
- Operations:
 - **Wait(&lock)**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
 - **Signal()**: Wake up one waiter, if any
 - **Broadcast()**: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!

Monitor with Condition Variables



- **Lock**: the lock provides mutual exclusion to shared data
 - Always acquire before accessing shared data structure
 - Always release after finishing with shared data
 - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
 - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section

Synchronized Buffer (with condition variable)

- Here is an (infinite) synchronized queue:

```
lock buf_lock; // Initially unlocked
condition buf_CV; // Initially empty
queue queue;

Producer(item) {
    acquire(&buf_lock); // Get Lock
    enqueue(&queue, item); // Add item
    cond_signal(&buf_CV); // Signal any waiters
    release(&buf_lock); // Release Lock
}

Consumer() {
    acquire(&buf_lock); // Get Lock
    while (isEmpty(&queue)) {
        cond_wait(&buf_CV, &buf_lock); // If empty, sleep
    }
    item = dequeue(&queue); // Get next item
    release(&buf_lock); // Release Lock
    return(item);
}
```

Summary (1/2)

- Important concept: **Atomic Operations**
 - An operation that runs to completion or not at all
 - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
 - Disabling of Interrupts, test&set, swap, compare&swap,
- Showed several constructions of Locks
 - Must be very careful not to waste/tie up machine resources
 - » Shouldn't disable interrupts for long
 - » Shouldn't spin wait for long
 - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Showed primitives for constructing user-level locks
 - Packages up functionality of sleeping

Summary (2/2)

- **Semaphores**: Like integers with restricted interface
 - Two operations:
 - » **P()**: Wait if zero; decrement when becomes non-zero
 - » **V()**: Increment and wake a sleeping task (if exists)
 - » Can initialize value to any non-negative value
 - Use separate semaphore for each constraint
- **Monitors**: A lock plus one or more condition variables
 - Always acquire lock before accessing shared data
 - Use condition variables to wait inside critical section
 - » Three Operations: **Wait()**, **Signal()**, and **Broadcast()**
- Monitors represent the logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed