

CS162  
Operating Systems and  
Systems Programming  
Lecture 8

Synchronization 2:  
Lock Implementation, Atomic Instructions,  
Futex, Need for Higher-Level Locking

February 12<sup>th</sup>, 2026

Prof. John Kubiawicz

<http://cs162.eecs.Berkeley.edu>

# Recall: Multiple Threads on One CPU/core

Consider the following code blocks:

```
proc A() {  
    B();  
}  
proc B() {  
    while(TRUE) {  
        yield();  
    }  
}
```

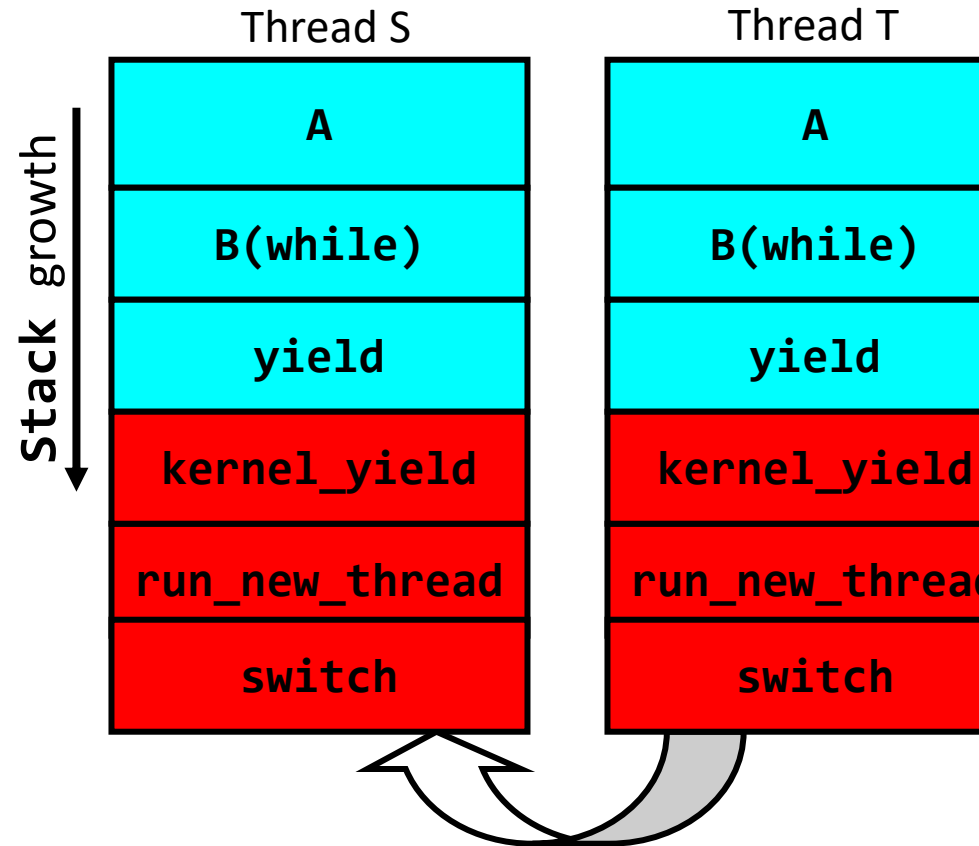
Suppose we have 2 threads:

Threads S and T

Kernel stack contains pointers to all state and can be placed on any queue:

**Ready queue** – available to run again

**Some wait queue** – won't run again until condition is resolved and back on ready queue



Thread T's switch returns to Thread S's switch

[ Thread T on Ready queue,  
Thread S is Running ]

## Recall: ATM bank server example

---

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}

ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}

Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

# Recall: Event Driven Version of ATM server

---

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style

- Example

```
BankServer() {
    while(TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest();
        else if (event == AcctAvail)
            ContinueRequest();
        else if (event == AcctStored)
            FinishRequest();
    }
}
```

- This technique is used for graphical programming

- Complication:
  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?

# Can Threads (in same Process) Make This Easier?

---

Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments

- One thread per request

Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {  
    acct = GetAccount(actId); /* May use disk I/O */  
    acct->balance += amount;  
    StoreAccount(acct);      /* Involves disk I/O */  
}
```

Unfortunately, shared state can get corrupted:

<u>Thread 1</u>	<u>Thread 2</u>
load r1, acct->balance	
	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

## Problem is at the Lowest Level

---

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, what about (Initially,  $y = 12$ ):

Thread A

$x = 1;$

$x = y + 1;$

Thread B

$y = 2;$

$y = y * 2;$

– What are the possible values of  $x$ ?

- Or, what are the possible values of  $x$  below?

Thread A

$x = 1;$

Thread B

$x = 2;$

–  $X$  could be 1 or 2 (non-deterministic!)

– Could even be 3 for serial processors:

» Thread A writes 0001, B writes 0010 → scheduling order ABABABBA yields 3!

# Atomic Operations

---

To understand a concurrent program, we need to know what the underlying indivisible operations are!

**Atomic Operation:** an operation that always runs to completion or not at all

- It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
- Fundamental building block – if no atomic operations, then have no way for threads to work together

On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

- Consequently – weird example that produces “3” on previous slide can’t happen

Many instructions are not atomic

- Double-precision floating point store often not atomic
- VAX and IBM 360 had an instruction to copy a whole array

# Another Concurrent Program Example

---

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

<u>Thread A</u>	<u>Thread B</u>
<pre>i = 0; while (i &lt; 10)     i = i + 1; printf("A wins!");</pre>	<pre>i = 0; while (i &gt; -10)     i = i - 1; printf("B wins!");</pre>

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

# Hand Simulation Multiprocessor Example

---

- Inner loop looks like this:

	<u>Thread A</u>		<u>Thread B</u>	
r1=0	load r1, M[i]		r1=0	load r1, M[i]
r1=1	add r1, r1, 1		r1=-1	sub r1, r1, 1
M[i]=1	store r1, M[i]		M[i]=-1	store r1, M[i]

- **Hand Simulation:**
  - And we're off. A gets off to an early start
  - B says "hmph, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH??? I could have sworn I put a 1 there"
- Could this happen on a uniprocessor? With Hyperthreads?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break...

# Definitions

---

**Synchronization:** using atomic operations to ensure cooperation between threads

- For now, only loads and stores are atomic
- We are going to show that its hard to build anything useful with only reads and writes

**Mutual Exclusion:** ensuring that only one thread does a particular thing at a time

- One thread *excludes* the other while doing its task

**Critical Section:** piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code

- Critical section is the result of mutual exclusion
- Critical section and mutual exclusion are two ways of describing the same thing

# 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

Locks need to be allocated and initialized:

- `structure Lock mylock`      or      `pthread_mutex_t mylock;`
- `lock_init(&mylock)`      or      `mylock = PTHREAD_MUTEX_INITIALIZER`

Locks provide two atomic operations:

- **acquire(&mylock)** – wait until lock is free; then mark it as busy
  - » After this returns, we say the calling thread *holds* the lock
- **release(&mylock)** – mark lock as free
  - » Should only be called by a thread that currently holds the lock
  - » After this returns, the calling thread no longer holds the lock

# Fix banking problem with Locks!

Identify critical sections (atomic instruction sequences) and add locking:

```
Deposit(acctId, amount) {
```

```
  acquire(&mylock)
```

```
  acct = GetAccount(actId);  
  acct->balance += amount;  
  StoreAccount(acct);
```

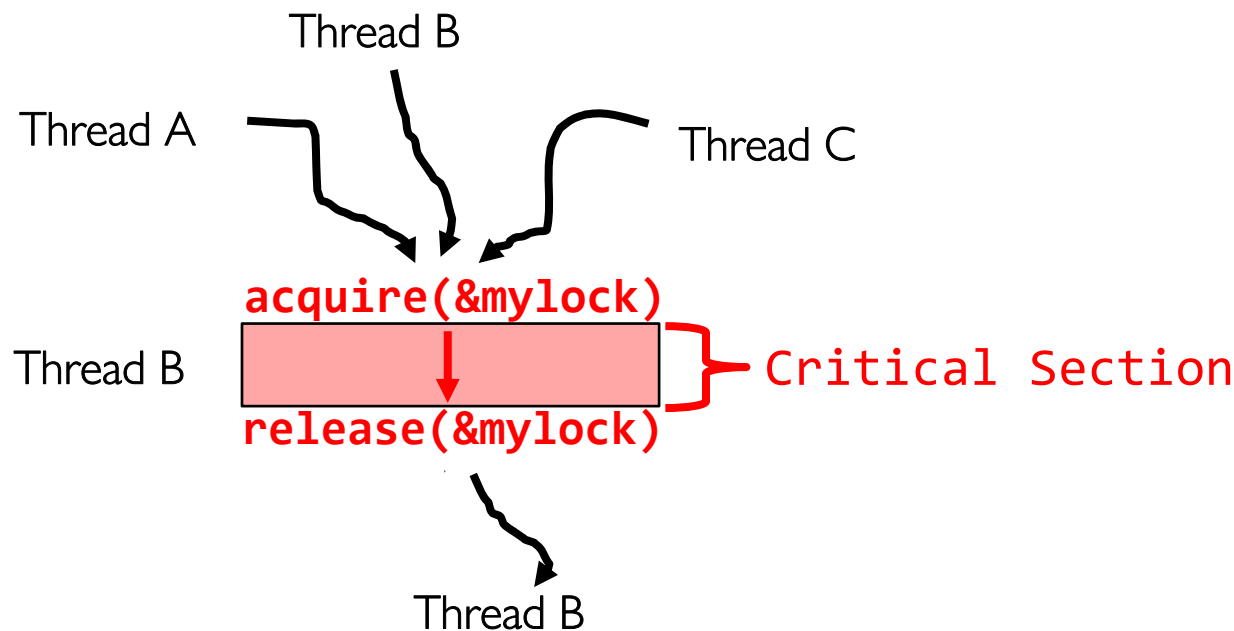
```
  release(&mylock)
```

```
}
```

```
// Wait if someone else in critical section
```

```
} Critical Section
```

```
// Release someone into critical section
```



Threads serialized by lock through critical section. Only one thread at a time

Must use SAME lock (**mylock**) with all of the methods (Withdraw, etc...)

– Shared with all threads!

# Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequence
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!

- Example: Therac-25

- Machine for radiation therapy
  - » Software control of electron accelerator and electron beam/Xray production
  - » Software control of dosage
- Software errors caused the death of several patients
  - » A series of race conditions on shared variables and poor software design

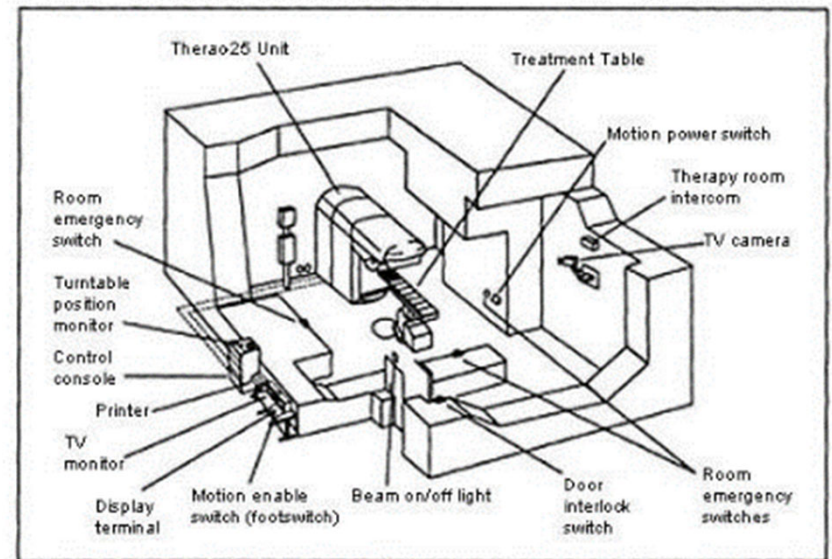


Figure 1. Typical Therac-25 facility

- » “They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred.”

# Administrivia

---

Midterm two weeks from last Tuesday (February 24, 7-10pm)!

- No class on day of midterm (extra office hours during class time)
- Topics, lectures, and assignments up to and including next Tuesday
- Closed book, one page of handwritten notes allowed

Project 1 Design Document due date Tuesday 2/17

Project 1 Design reviews upcoming

- High-level discussion of your approach
  - » What will you modify?
  - » What algorithm will you use?
  - » How will things be linked together, etc.
  - » Do not need final design (complete with all semicolons!)
- You will be asked about testing
  - » Understand testing framework
  - » Are there things you are doing that are not tested by tests we give you?

# Today's 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

# Solve with a lock?

**Recall:** Lock prevents someone from doing something

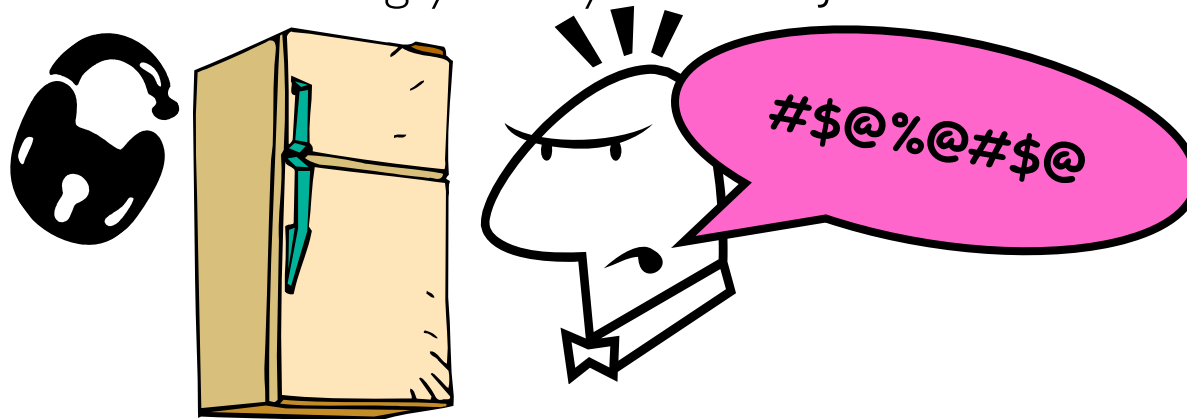
- Lock before entering critical section
- Unlock when leaving
- Wait if locked



» Important idea: all synchronization involves waiting

For example: fix the milk problem by putting a key on the refrigerator

- Lock it and take key if you are going to go buy milk
- Fixes too much: roommate angry if only wants OJ



Of Course – We don't know how to make a lock yet

- Let's see if we can answer this question!

# Too Much Milk: Correctness Properties

---

Need to be careful about correctness of concurrent programs, since non-deterministic

- Impulse is to start coding first, then when it doesn't work, pull hair out
- Instead, think first, then code
- Always write down behavior first

What are the correctness properties for the “Too much milk” problem???

- Never more than one person buys
- Someone buys if needed

First attempt: Restrict ourselves to use only atomic load and store operations as building blocks

# Too Much Milk: Solution #1

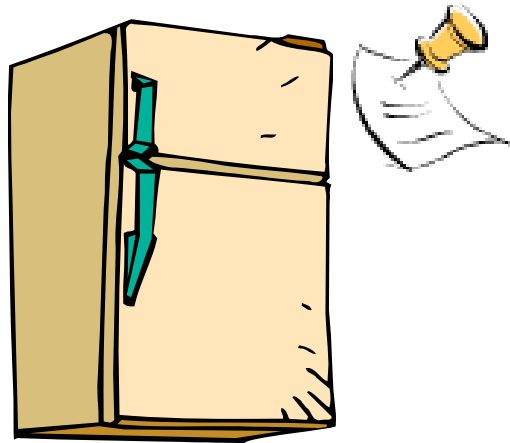
---

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;  
  }  
}
```



# 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):

Thread A

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

Thread B

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

# Too Much Milk: Solution #1

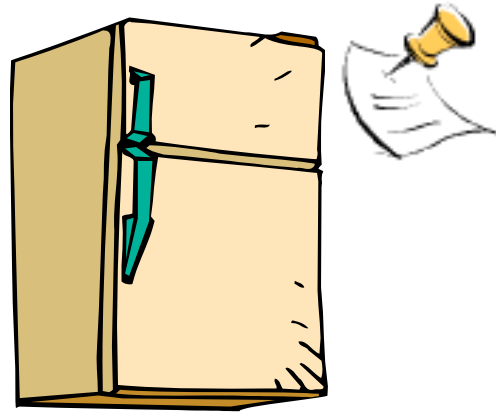
---

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

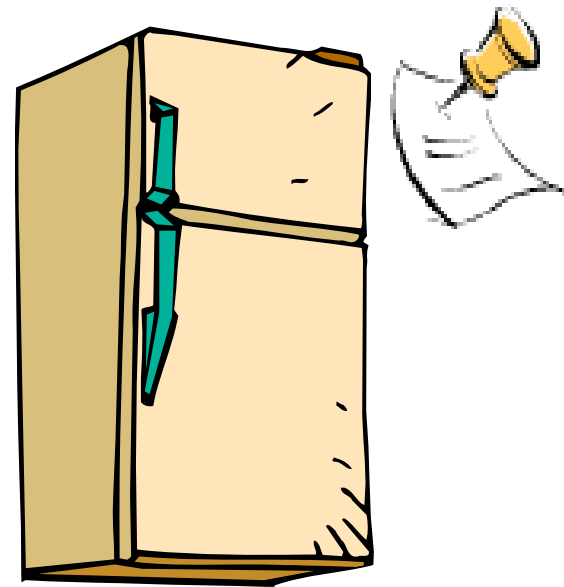
- Context switches at exactly the wrong times can lead each to think that the other is going to buy

Really insidious:

- **Extremely unlikely** this would happen, but will at worse possible time
- Probably something like this in UNIX

## Too Much Milk Solution #2: problem!

---



- *I'm not getting milk, You're getting milk*
- This kind of lockup is called "starvation!"

## 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)”

```
leave note A;  
while (note B) {\X  
    do nothing;  
};
```

*happened  
before*



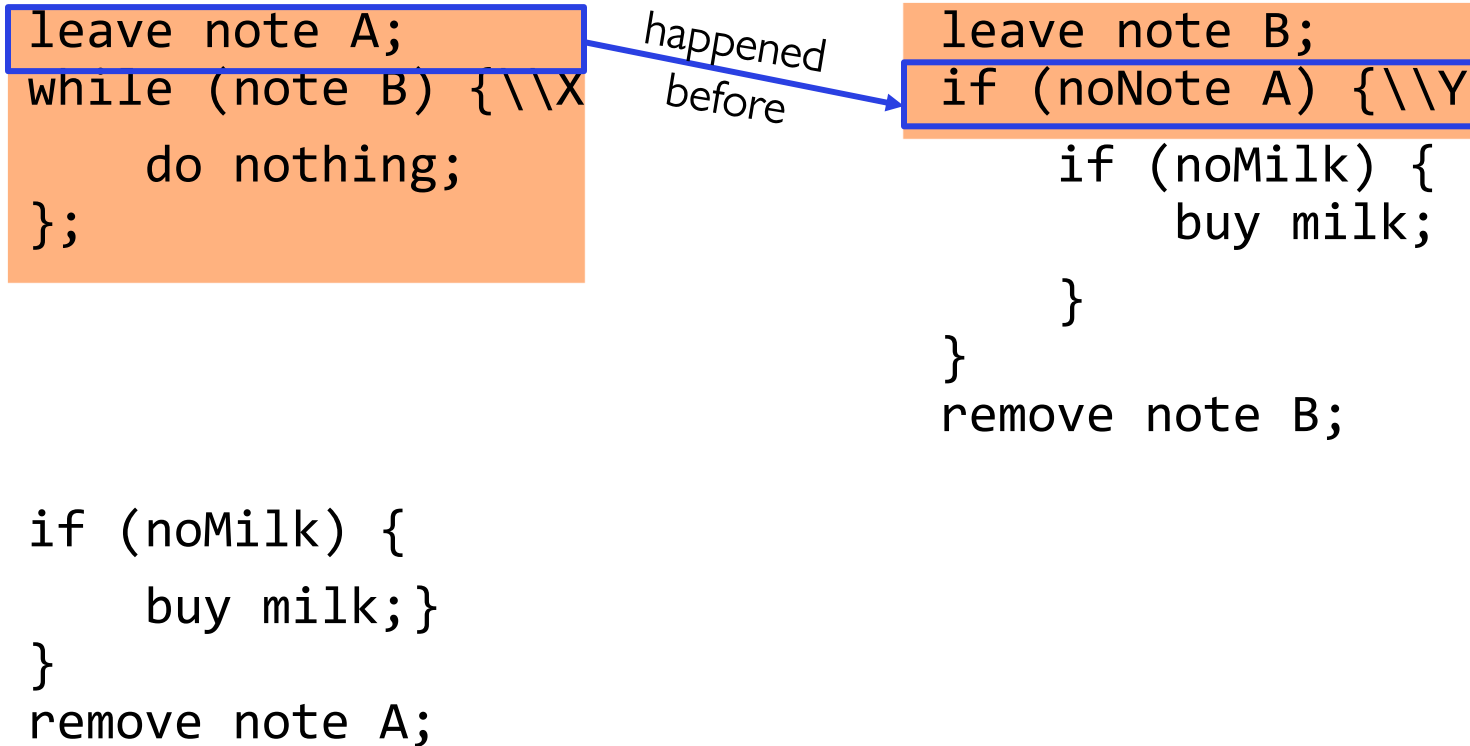
```
leave note B;  
if (noNote A) {\Y  
    if (noMilk) {  
        buy milk;  
    }  
}  
remove note B;
```

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

# Case 1

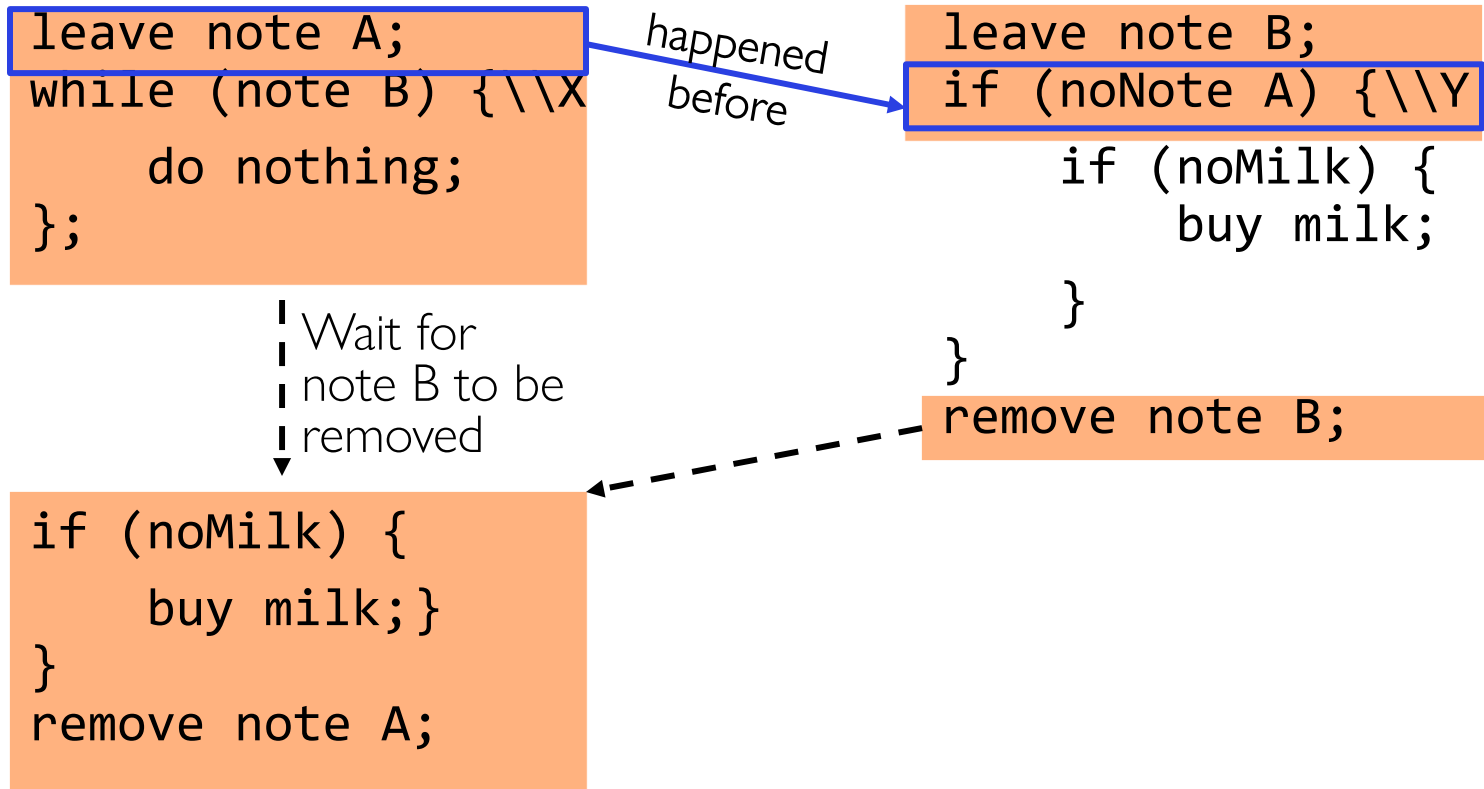
---

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



# Case 1

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



## Case 2

---

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

```
leave note A;  
while (note B) {\X  
    do nothing;  
};
```

happened  
before

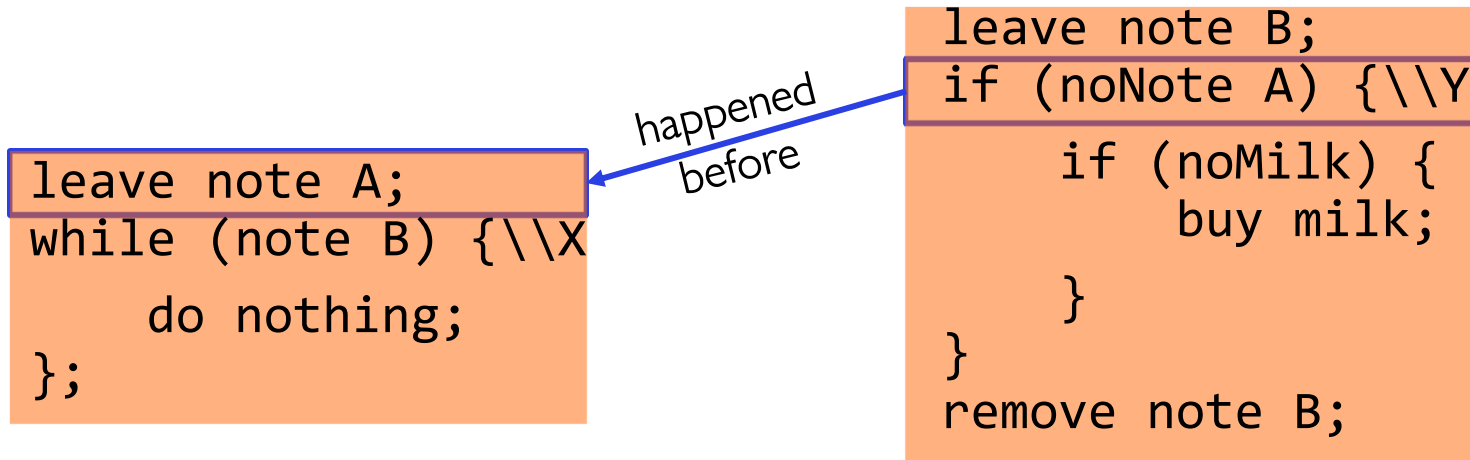
```
leave note B;  
if (noNote A) {\Y  
    if (noMilk) {  
        buy milk;  
    }  
}  
remove note B;
```

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

## Case 2

---

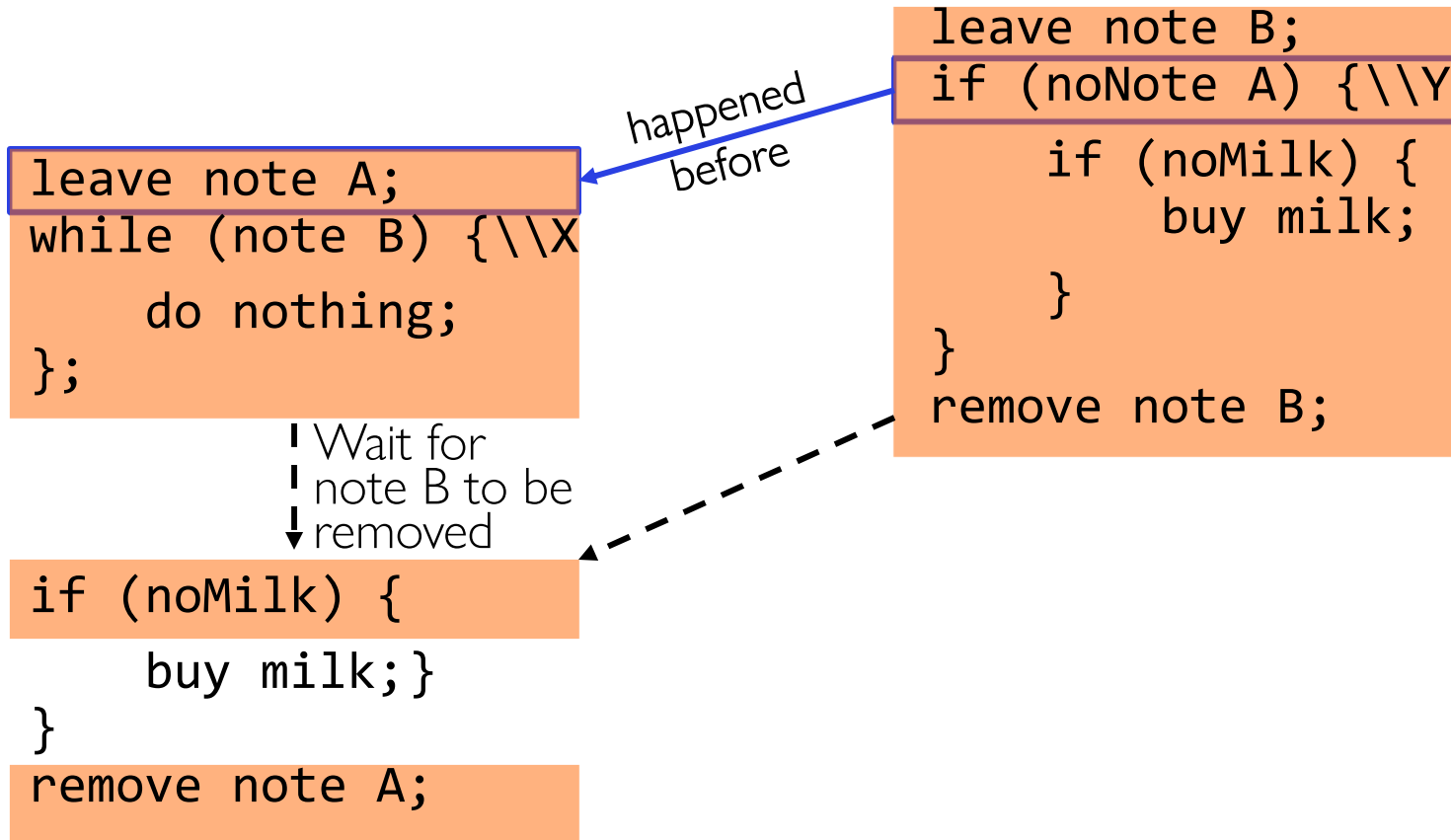
- “if (noNote A)” happens before “leave note A”



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

## Case 2

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



# This Generalizes to $n$ Threads...

---

Leslie Lamport's "Bakery Algorithm" (1974)

Computer  
Systems

G. Bell, D. Siewiorek,  
and S.H. Fuller, Editors

---

## A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport  
Massachusetts Computer Associates, Inc.

---

**A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate**

## 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:

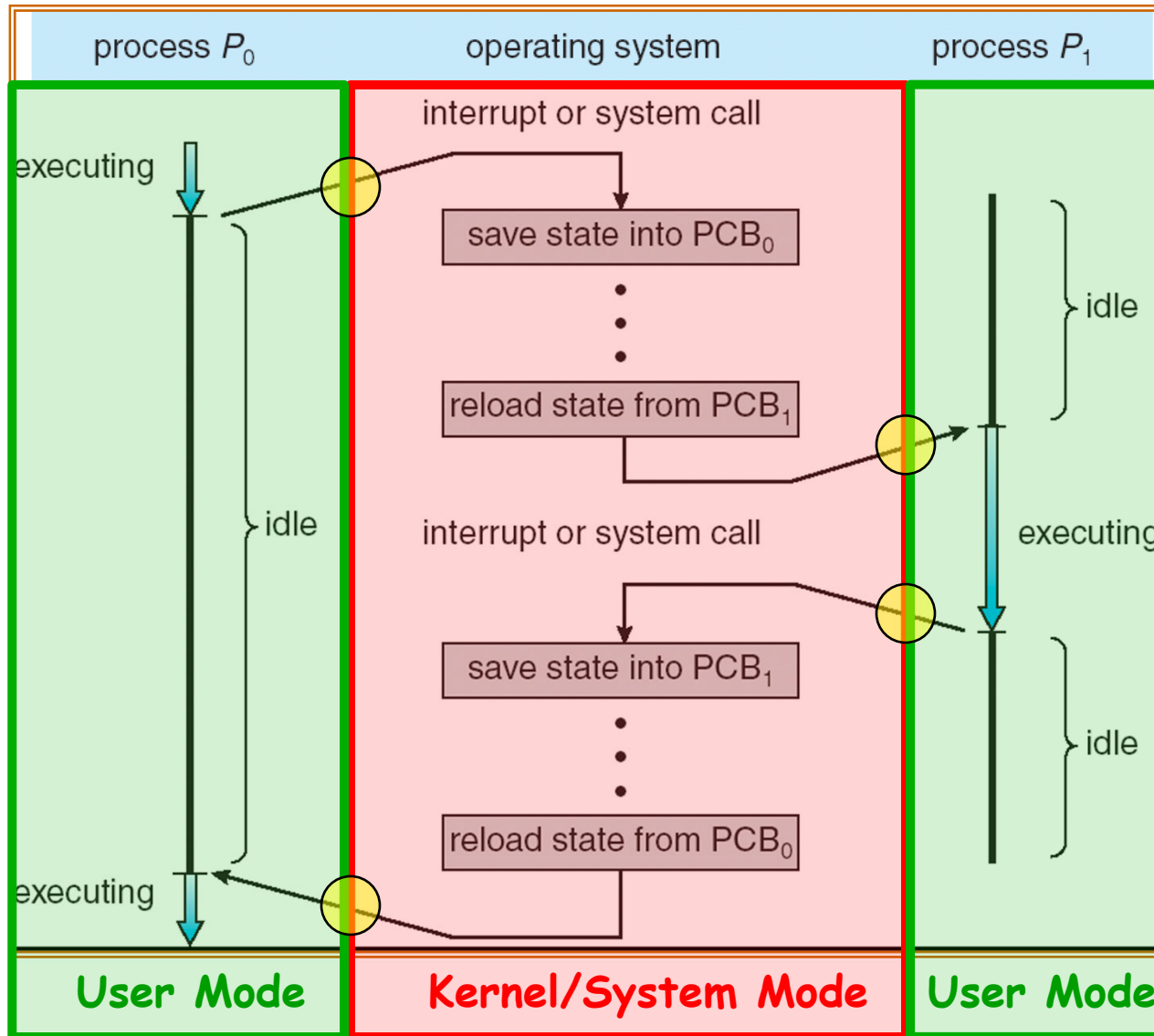
```
acquire(&milklock);  
if (nomilk)  
    buy milk;  
release(&milklock);
```

# Where are we going with synchronization?

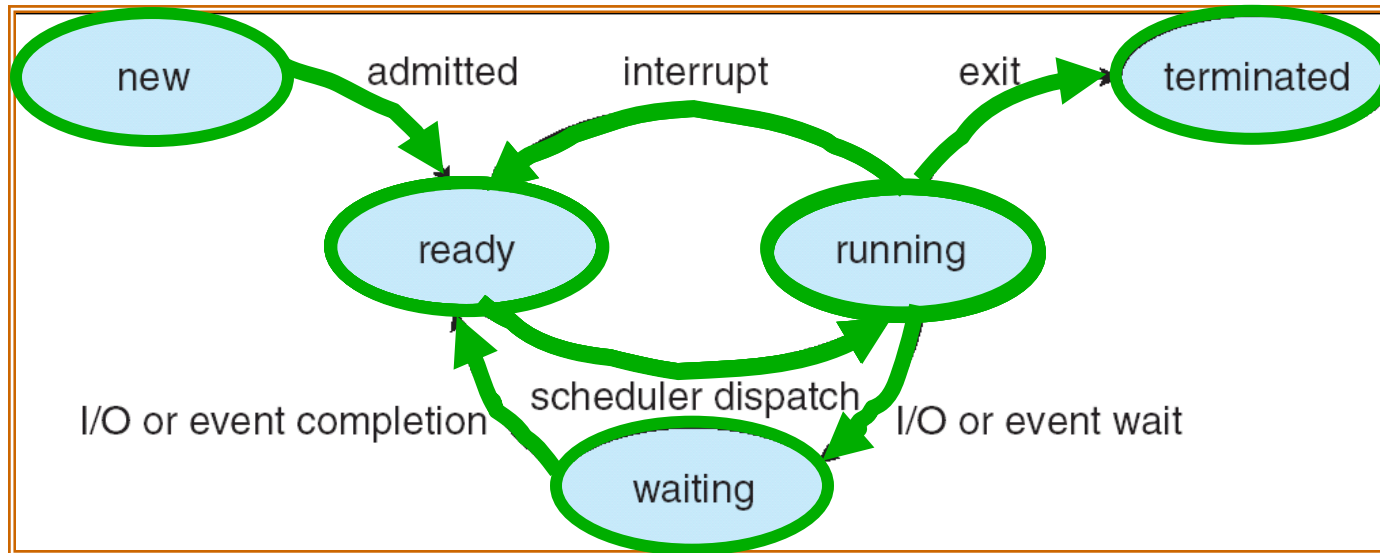
Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

- 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

# Recall: CPU Switch From Process A to Process B

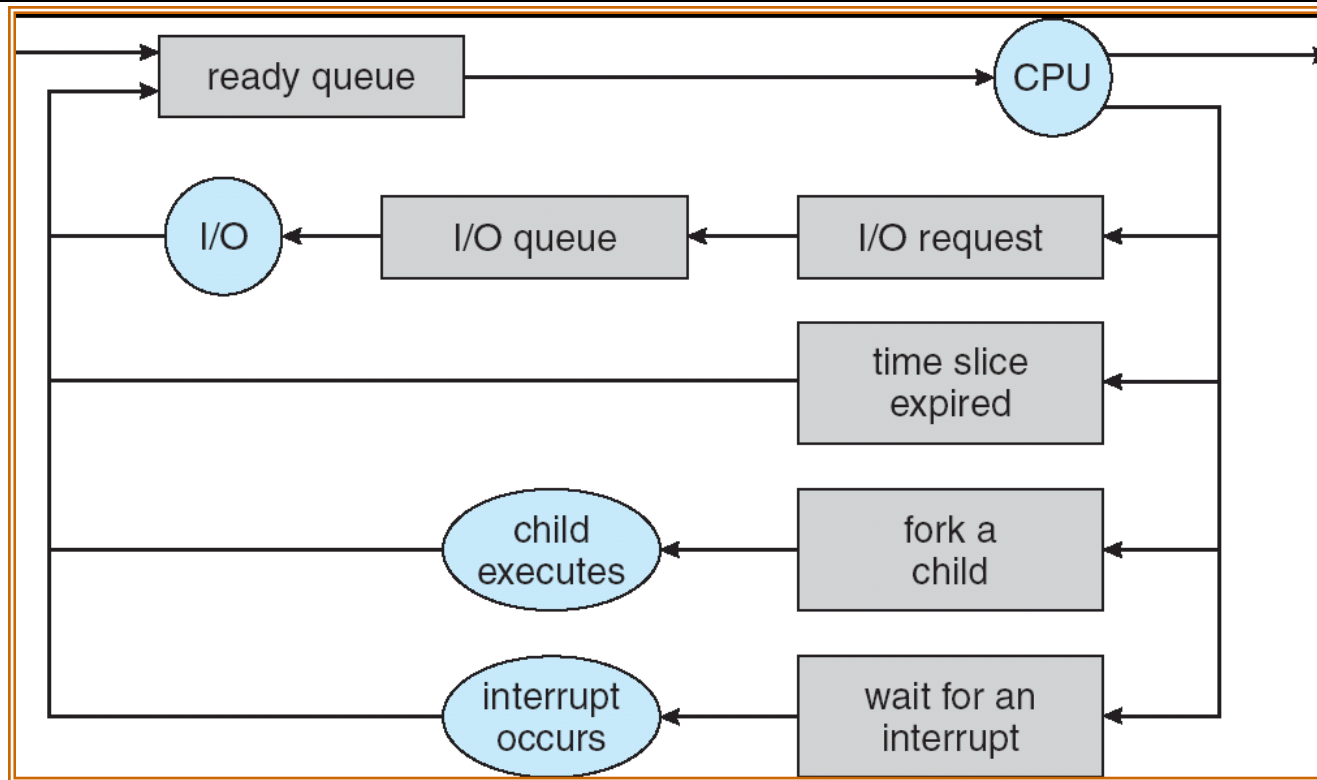


# Lifecycle of a Process (same for Thread)



- As a process executes, it changes state:
  - **new**: The process is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution (for process: zombie!)

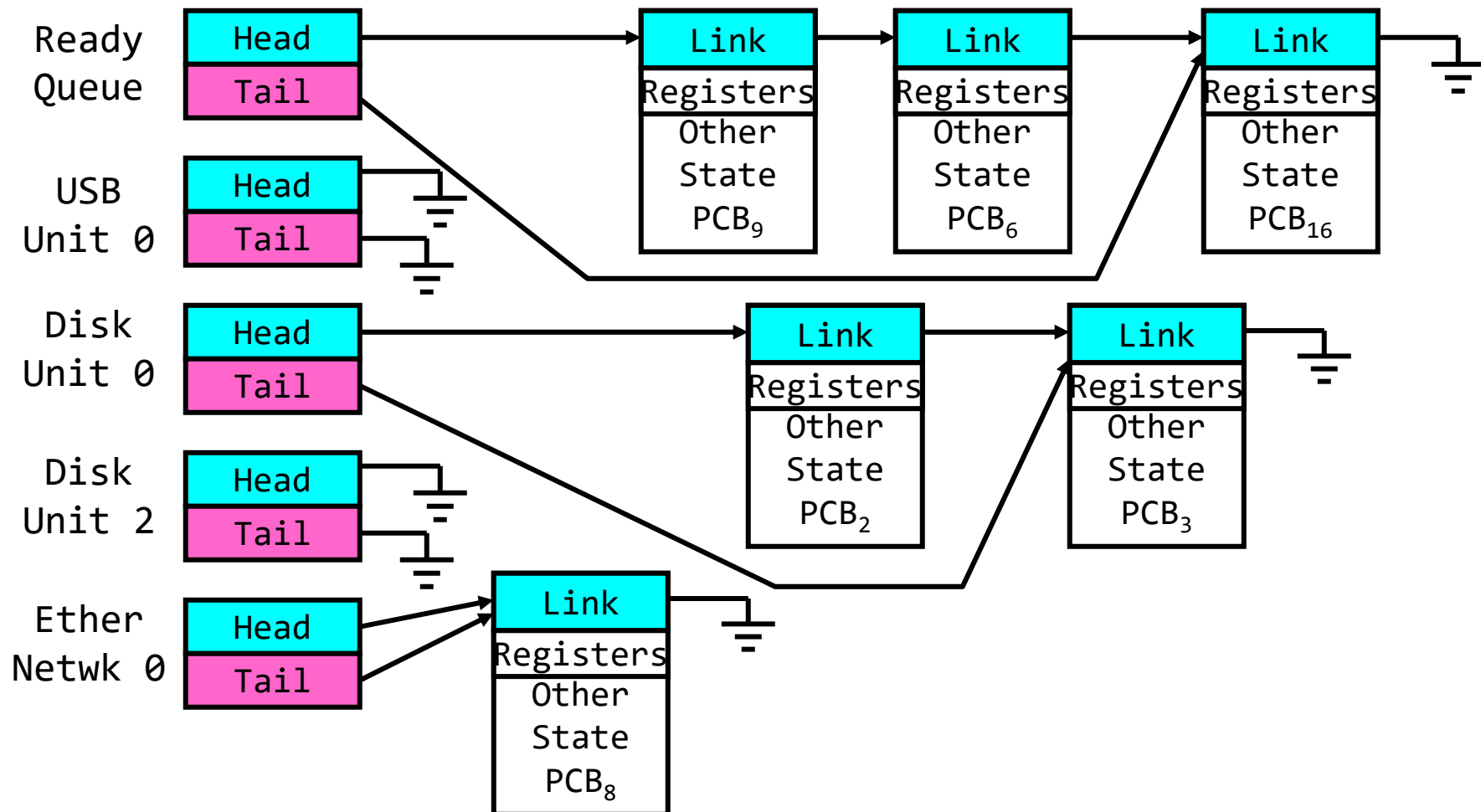
# Process/Thread Scheduling



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are **Scheduling** decisions
  - Many algorithms possible (few weeks from now)

# Ready Queue And Various I/O Device Queues

- Process not running  $\Rightarrow$  PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



# 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
  - Is this a good idea?
  - What about putting a task to sleep?
    - » What is the interface between the hardware and scheduler?
  - 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: scheduler 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 (although virtual memory tricky)
    - » 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.
  - *Prevent switching to other thread that might be trying to acquire lock!*
  - 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;
```

```
}
```

“Meta-”  
Critical  
Section

- Note: unlike previous solution, this “meta-”critical section 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!

# What about 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;
}
```

# What about Interrupt Re-enable in Going to Sleep?

---

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

Enable Position? 


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

- Before Putting thread on the wait queue?

# What about Interrupt Re-enable in Going to Sleep?

---

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

Enable Position? 

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


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

# What about 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

# What about 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!)

# What about Interrupt Re-enable in Going to Sleep?

---

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

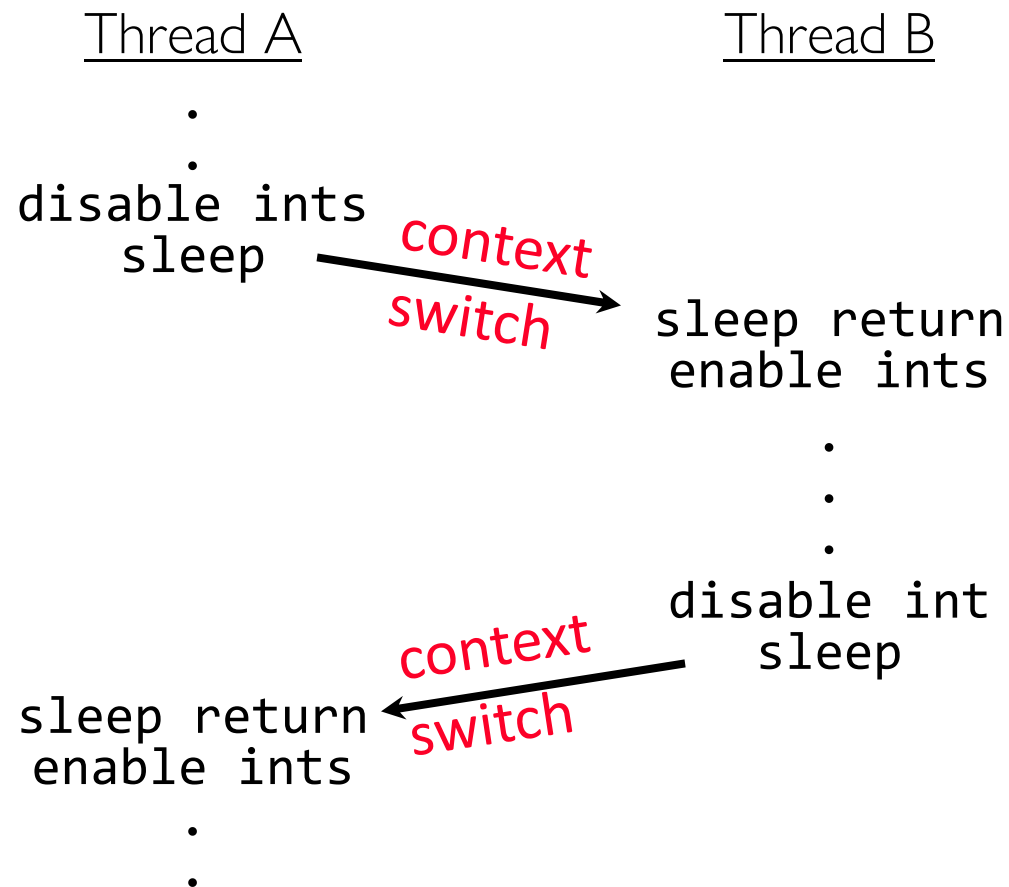
```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Enable Position? 

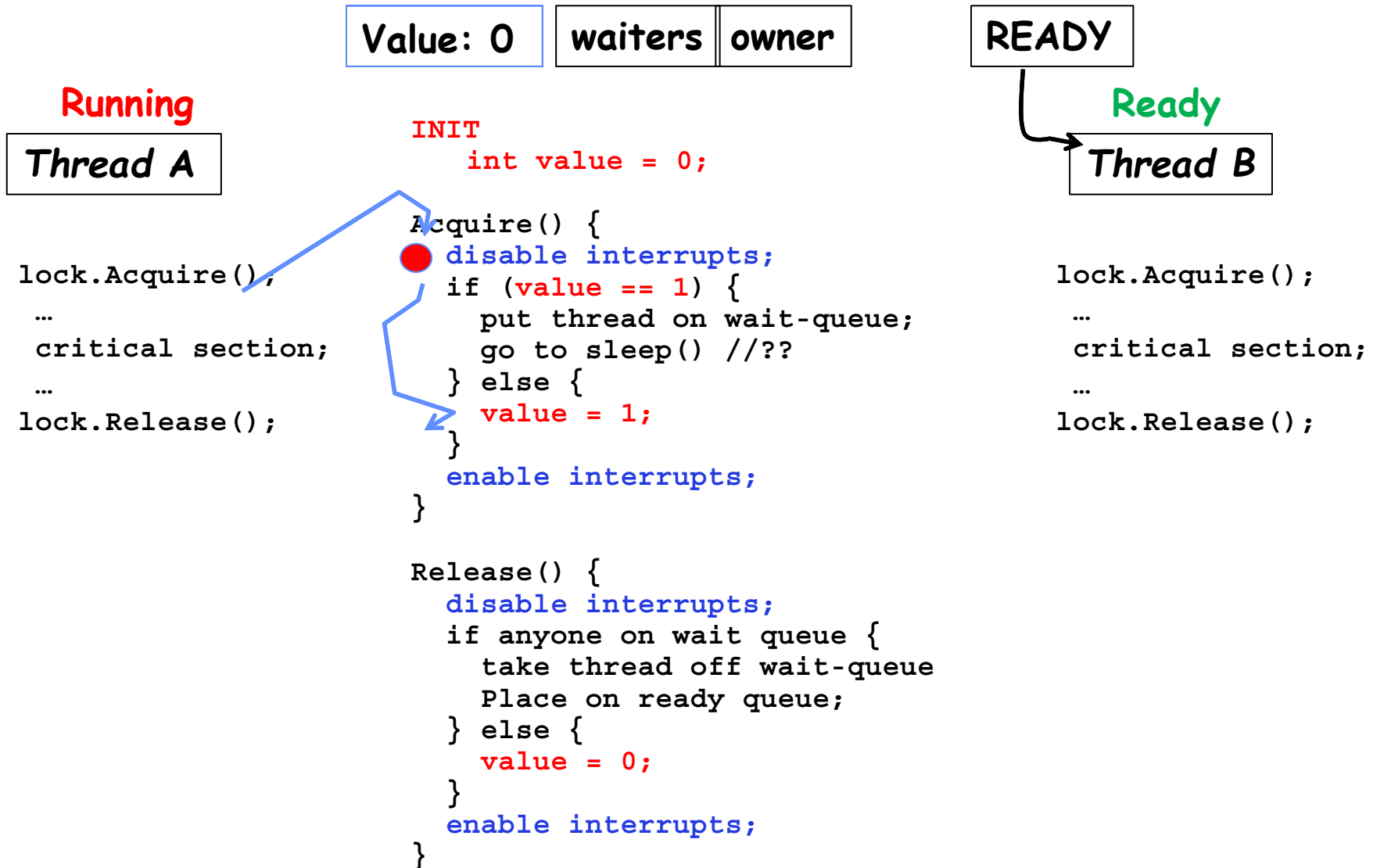
- Before Putting thread on the wait queue
  - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
  - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  - Misses wakeup and still holds lock (deadlock!)
- Want to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

# How to Re-enable After Sleep()?

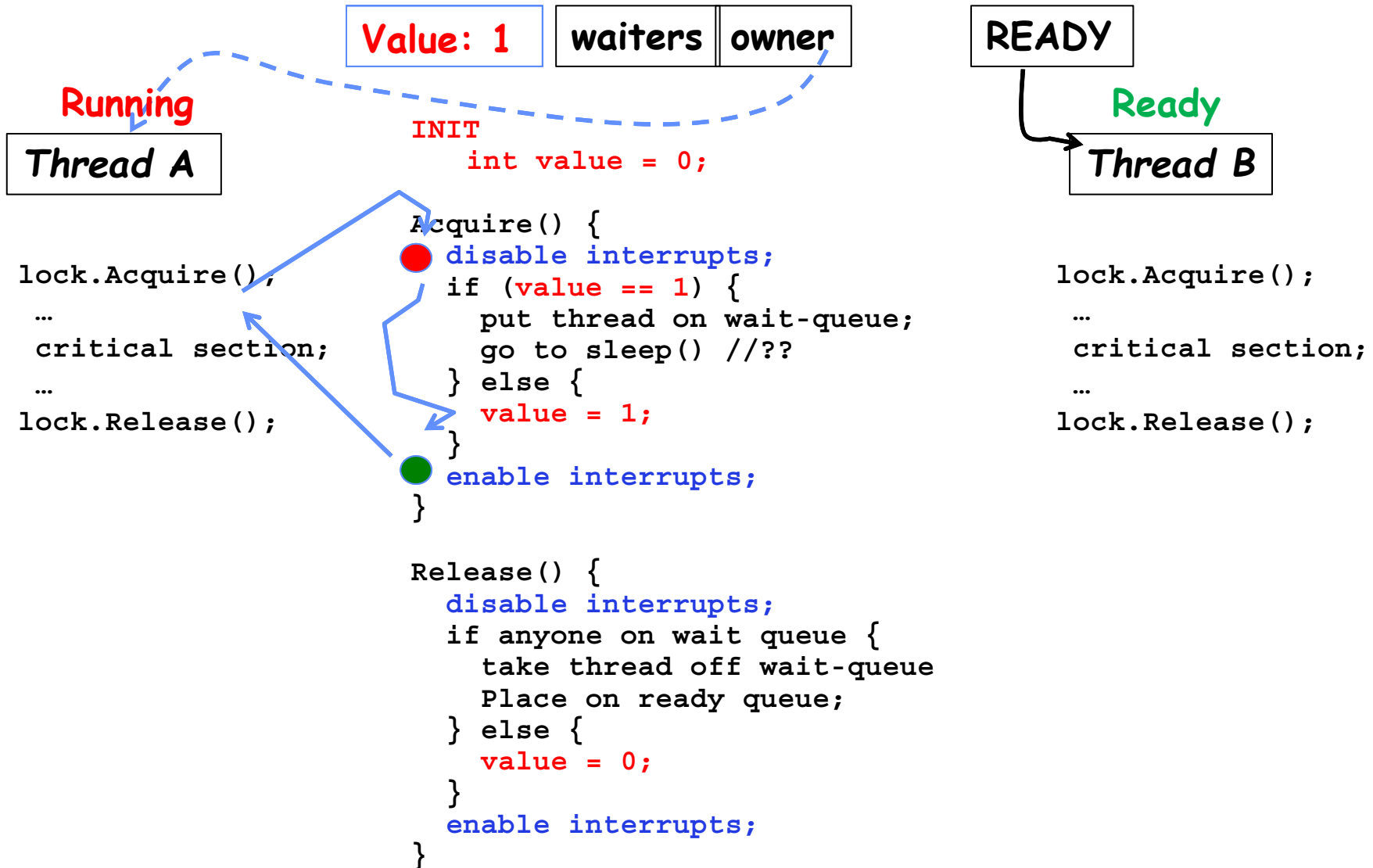
- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



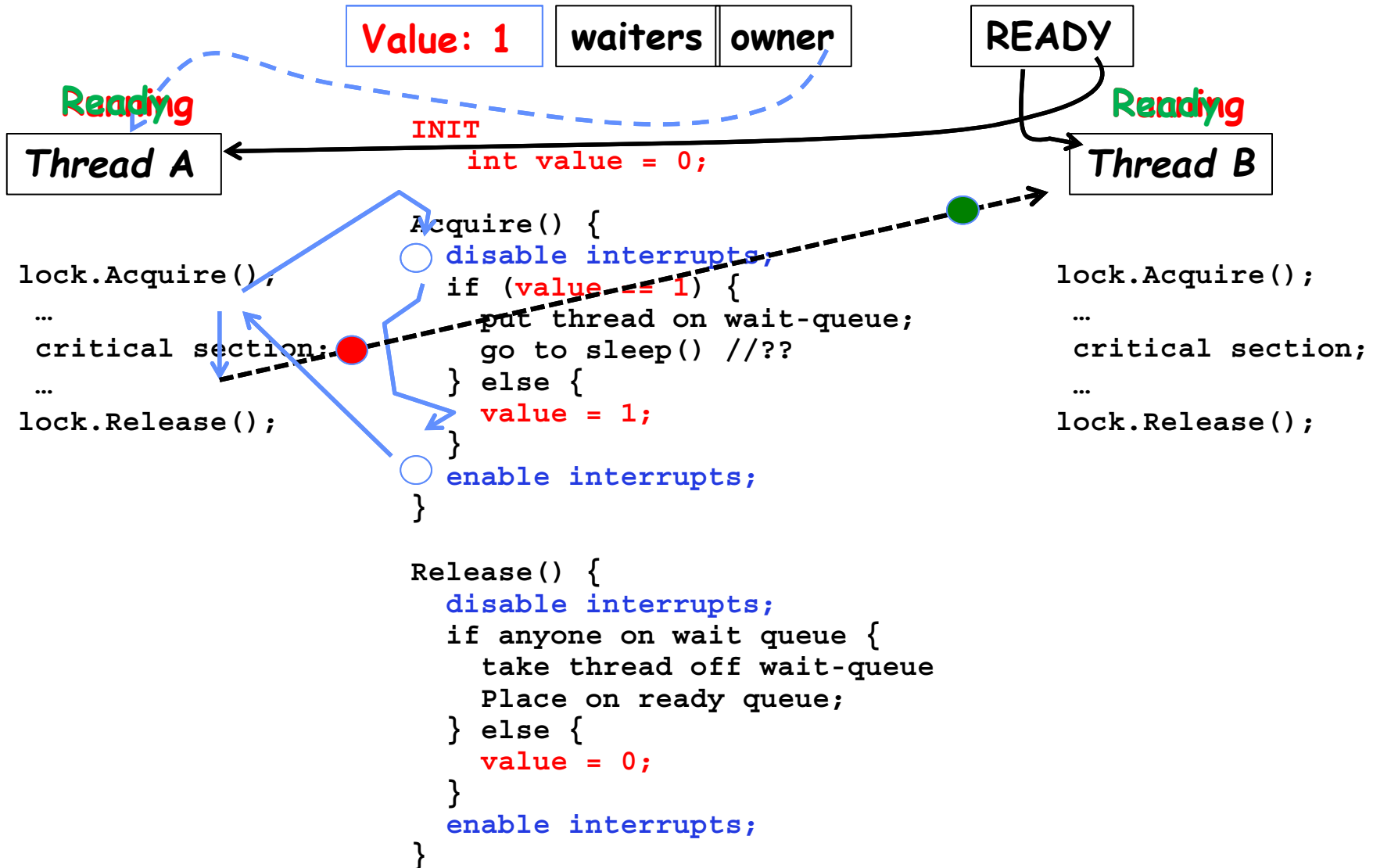
# In-Kernel Lock: Simulation



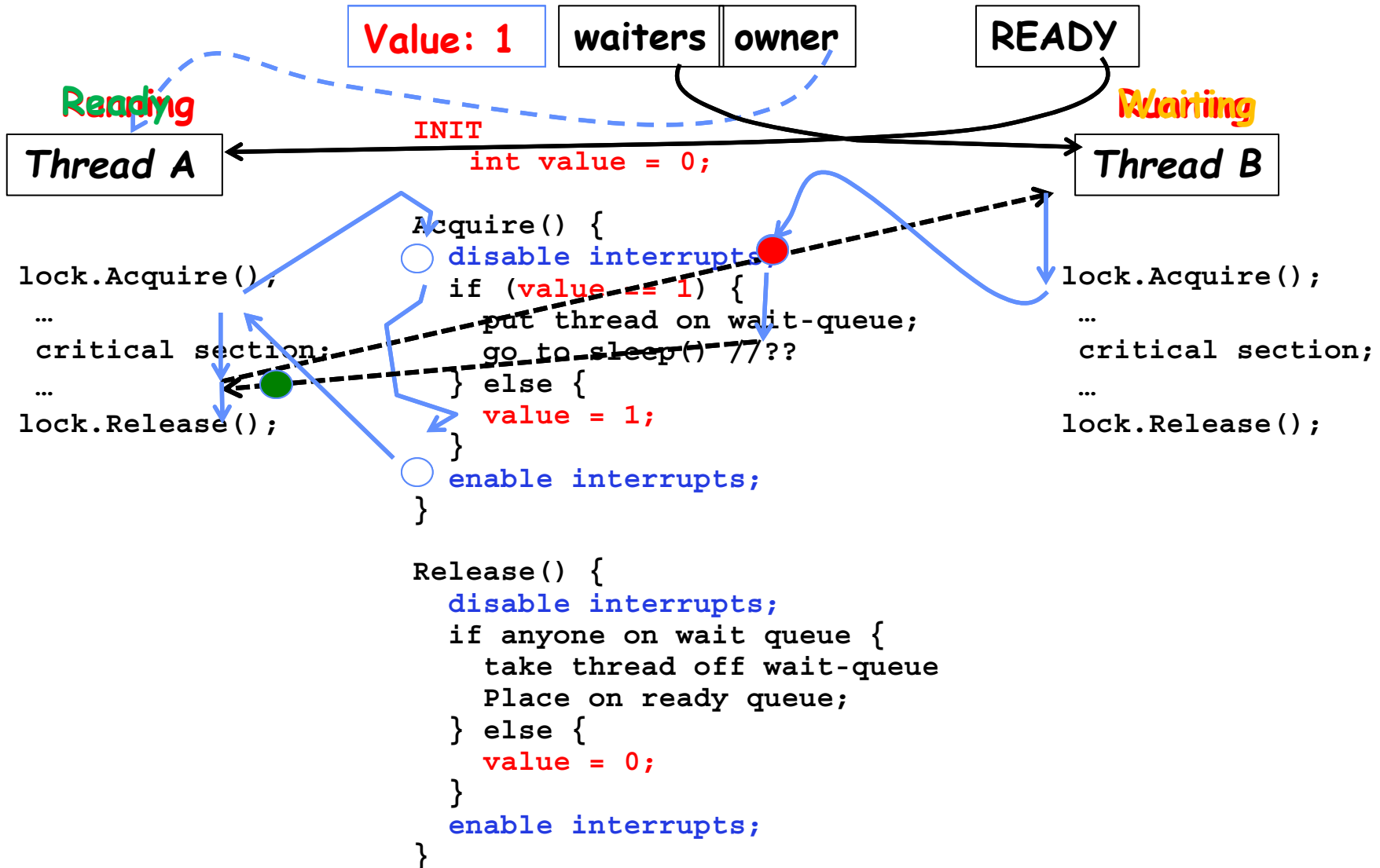
# In-Kernel Lock: Simulation



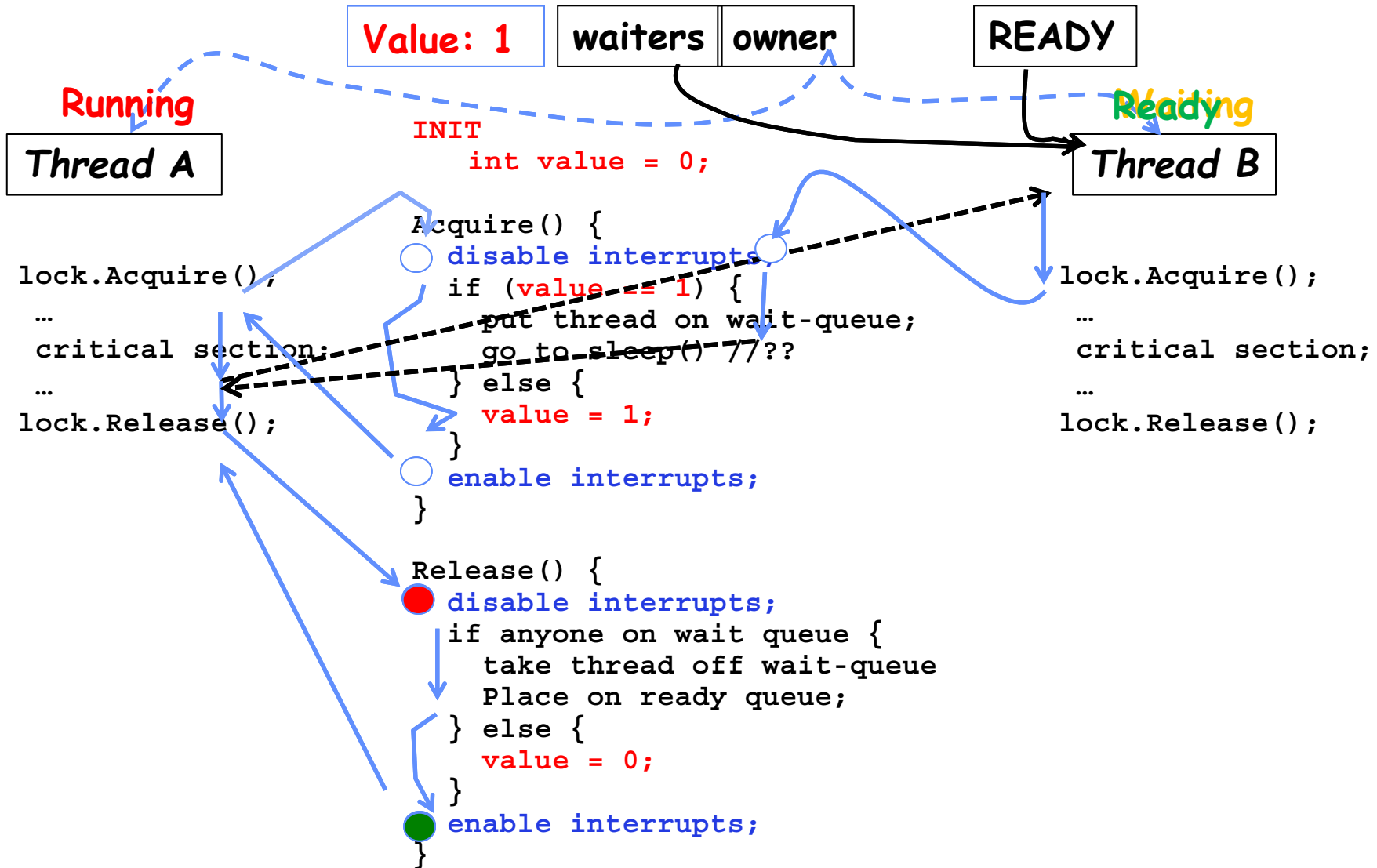
# In-Kernel Lock: Simulation



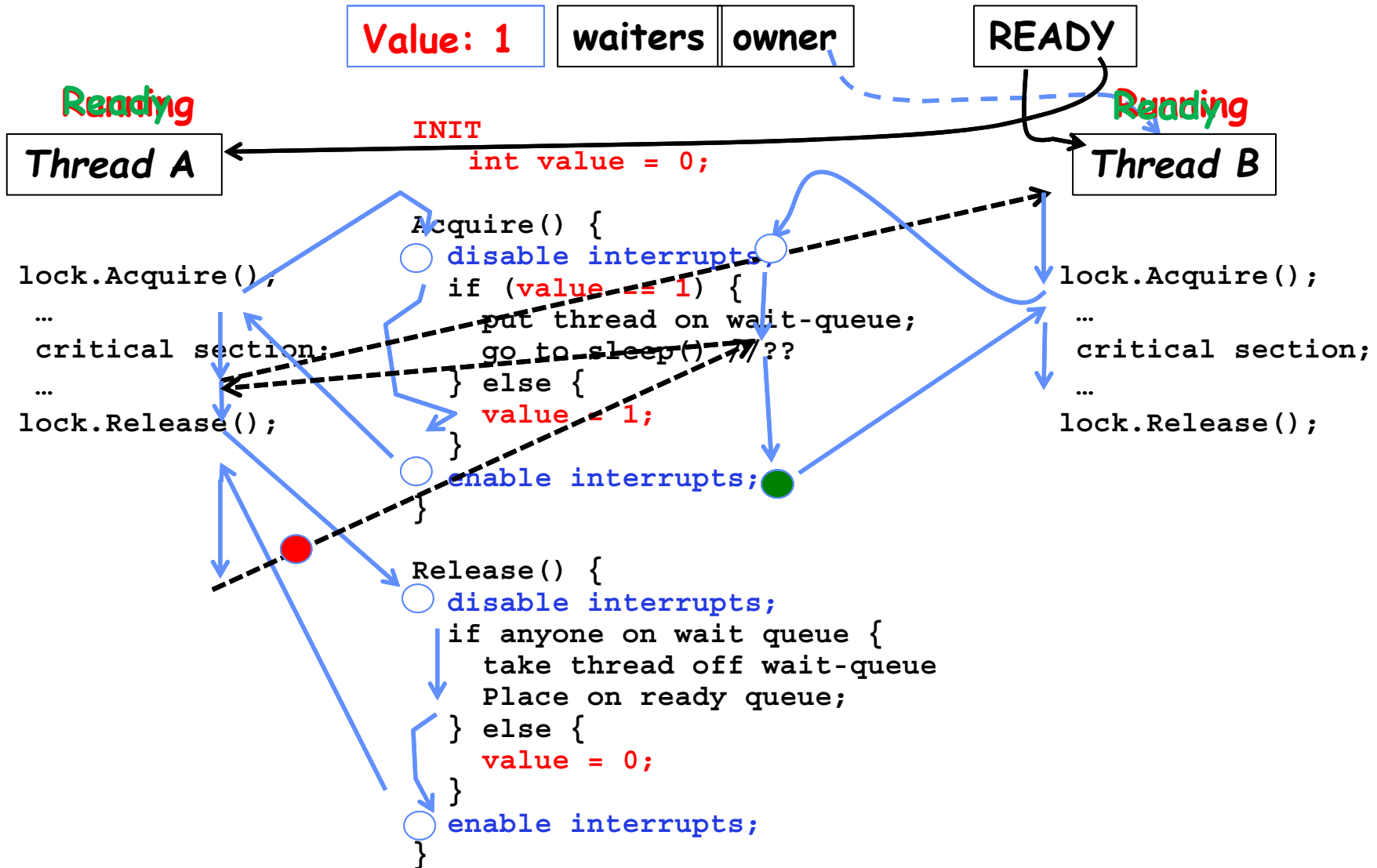
# In-Kernel Lock: Simulation



# 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;` `// value from "address" put back to register`  
    `return temp;` `// value from "address" considered return from swap`  
}
- `compare&swap (&address, reg1, reg2) {` `/* x86 (returns old value), 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;`  
    `}`  
}
- `load-linked&store-conditional(&address) {` `/* R4000, alpha */`  
    `loop:`  
        `ll r1, M[address];`  
        `movi r2, 1;` `// Can do arbitrary computation`  
        `sc r2, M[address];`  
        `beqz r2, loop;`  
    `}`

# Implementing Locks with test&set

---

Simple lock that doesn't require entry into the kernel:

```
// (Free) Can access this memory location from user space!  
int thelock = 0; // Interface: acquire(&thelock);  
                //                release(&thelock);  
  
acquire(int *thelock) {  
    while (test&set(thelock)); // Atomic operation!  
}  
  
release(int *thelock) {  
    *thelock = 0; // Atomic operation!  
}
```

Simple explanation:

- If lock is free, test&set reads 0 and sets lock=1, so lock is now busy. It returns 0 so while exits.
- If lock is busy, test&set reads 1 and sets lock=1 (no change) It returns 1, so while loop continues.
- When we set thelock = 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

For higher-level synchronization primitives (e.g. semaphores or monitors), waiting threads may wait for an arbitrary long time!

- Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
- Homework/exam solutions should avoid busy-waiting!

# Multiprocessor Spin Locks: test&test&set

---

A better solution for multiprocessors:

```
// (Free) Can access this memory location from user space!  
int thelock = 0; // Interface: acquire(&thelock);  
                //                release(&thelock);  
  
acquire(int *thelock) {  
    do {  
        while(*thelock); // Wait until might be free (quick check/test)  
    } while(test&set(thelock)); // Atomic grab of lock (exit if succeeded)  
}  
  
release(int *thelock) {  
    *thelock = 0; // Atomic release of lock  
}
```

Simple explanation:

- Wait until lock might be free (only reading – stays in cache)
- Then, try to grab lock with test&set
- Repeat if fail to actually get lock

Issues with this solution:

- **Busy-Waiting**: thread still consumes cycles while waiting
  - » However, it does not impact other processors!

# Better Locks using test&set

Can we build test&set locks without busy-waiting?

– Mostly. Idea: only busy-wait to atomically check lock value

– `int guard = 0; // Global Variable!`

`int thelock = FREE; // Interface: acquire(&thelock);  
// release(&thelock);`

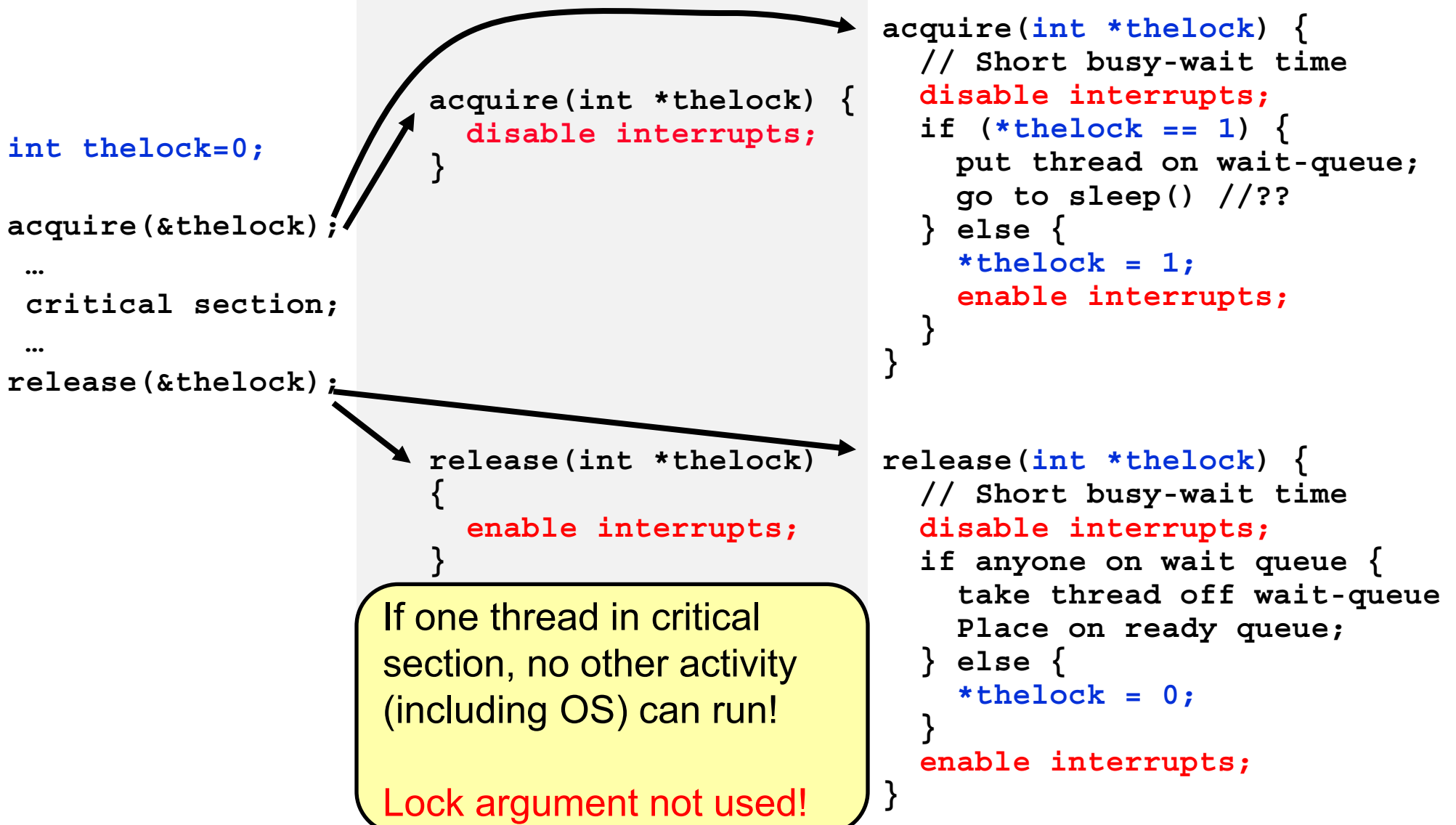
```
acquire(int *thelock) {  
    // Short busy-wait time  
    while (test&set(guard));  
    if (*thelock == BUSY) {  
        put thread on wait queue;  
        go to sleep() & guard = 0;  
        // guard == 0 on wakeup!  
    } else {  
        *thelock = BUSY;  
        guard = 0;  
    }  
}
```

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

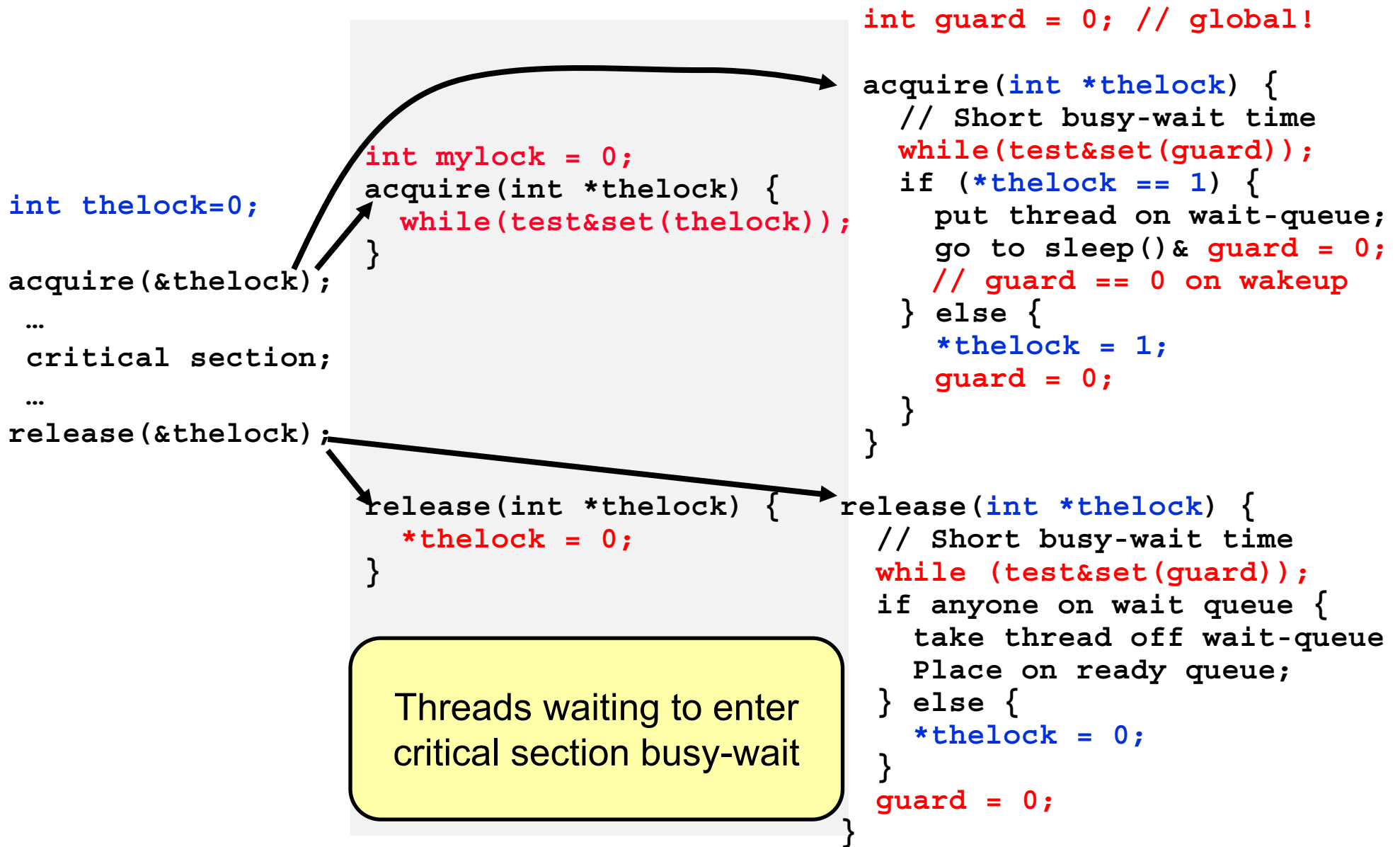
Note: sleep has to be sure to reset the guard variable

– Why can't we do it just before or just after the sleep?

# Recap: Locks using interrupts



# Recap: Locks using test & set



# Linux futex: Fast Userspace Mutex

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout );
```

`uaddr` points to a 32-bit value in user space

`futex_op`

- FUTEX\_WAIT – if `val == *uaddr` sleep till FUTEX\_WAIT
  - » *Atomic* check that condition still holds after we disable interrupts (in kernel!)
- FUTEX\_WAKE – wake up at most `val` waiting threads
- FUTEX\_FD, FUTEX\_WAKE\_OP, FUTEX\_CMP\_REQUEUE: More interesting operations!

`timeout`

- ptr to a *timespec* structure that specifies a timeout for the op

Interface to the kernel `sleep()` functionality!

- Let thread put themselves to sleep – conditionally!

**futex** is not exposed in `libc`; it is used within the implementation of `pthread`s

- Can be used to implement locks, semaphores, monitors, etc...

## Example: First try: T&S and futex

---

```
int mylock = 0; // Interface: acquire(&mylock);
                //                release(&mylock);

acquire(int *thelock) {
    while (test&set(thelock)) {
        futex(thelock, FUTEX_WAIT, 1);
    }
}

release(int *thelock) {
    *thelock = 0; // unlock
    futex(thelock, FUTEX_WAKE, 1);
}
```

Properties:

- Sleep interface by using futex – no busywaiting

No overhead to acquire lock

- Good!

Every unlock has to call kernel to potentially wake someone up – even if none

- Doesn't quite give us no-kernel crossings when uncontended...!

## Example: Try #2: T&S and futex

---

```
1 maybe_waiters = false;
  mylock = 0; // Interface: acquire(&mylock,&maybe_waiters);
                //                release(&mylock,&maybe_waiters);

acquire(int *thelock, bool *maybe) {
  while (test&set(thelock)) {
    // Sleep, since lock busy!
    *maybe = true;
    futex(thelock, FUTEX_WAIT, 1);

    // Make sure other sleepers not stuck
    *maybe = true;
  }
}

release(int *thelock, bool *maybe) {
  *thelock = 0;
  if (*maybe) {
    *maybe = false;
    // Try to wake up someone
    futex(thelock, FUTEX_WAKE, 1);
  }
}
```

This is syscall-free in the uncontended case

– Temporarily falls back to syscalls if multiple waiters, or concurrent acquire/release

– but it can be considerably optimized!

– See “[Futexes are Tricky](#)” by Ulrich Drepper

## Try #3: Better, using more atomics

better: Three (3) states:

**UNLOCKED**: No one has lock

**LOCKED**: One thread has lock

**CONTESTED**: Possibly more than one  
(with someone sleeping)

an interface!

grabbed cleanly by either

**compare&swap()**

or **swap()**

with overhead if uncontested!

and build semaphores in a similar way!

```
typedef enum { UNLOCKED, LOCKED, CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface: acquire(&mylock)
                          //              release(&mylock)

acquire(Lock *thelock) {
    // If unlocked, grab lock!
    if (compare&swap(thelock, UNLOCKED, LOCKED))
        return;

    // Keep trying to grab lock, sleep in futex
    while (swap(thelock, CONTESTED) != UNLOCKED)
        // Sleep unless someone releases here!
        futex(thelock, FUTEX_WAIT, CONTESTED);
}

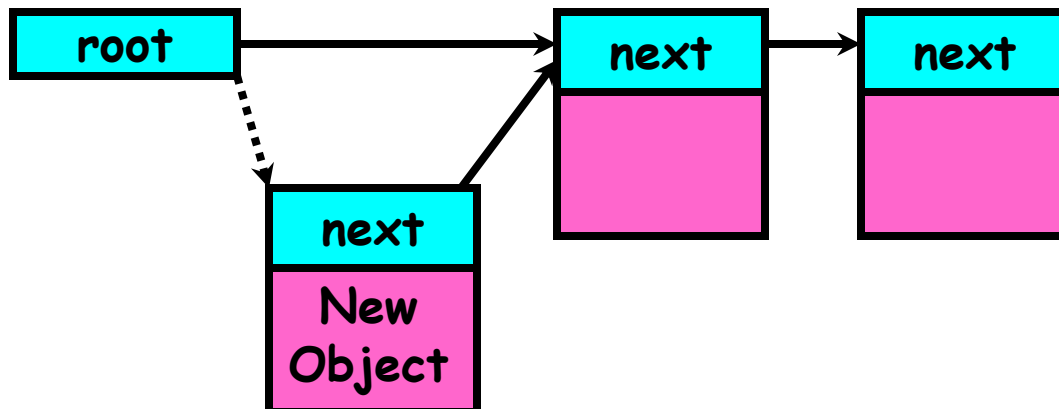
release(Lock *thelock) {
    // If someone sleeping,
    if (swap(thelock, UNLOCKED) == CONTESTED)
        futex(thelock, FUTEX_WAKE, 1);
}
```

# Using of Compare&Swap for lock-free queues

```
• compare&swap (&address, reg1, reg2) { /* x86, 68000 */  
  if (reg1 == M[address]) {  
    M[address] = reg2;  
    return success;  
  } else {  
    return failure;  
  }  
}
```

Here is an atomic add to linkedlist function:

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



# 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

# Summary

---

- 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, load-locked & store-conditional
- 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 primitive for constructing user-level locks
  - Packages up functionality of sleeping