

CS162
Operating Systems and
Systems Programming
Lecture 8

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

February 8th, 2024

Prof. John Kubiatowicz

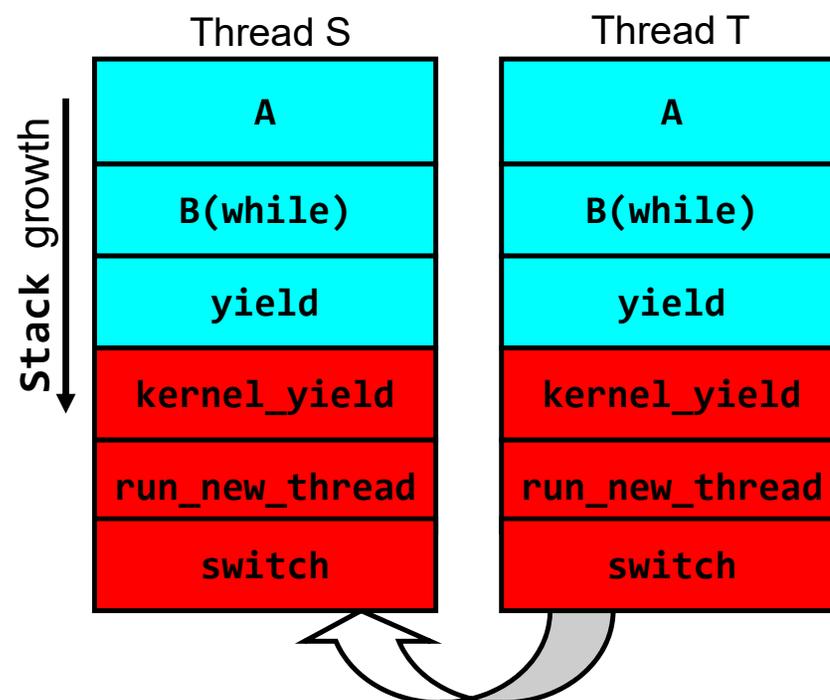
<http://cs162.eecs.Berkeley.edu>

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 resolved and back on ready queue

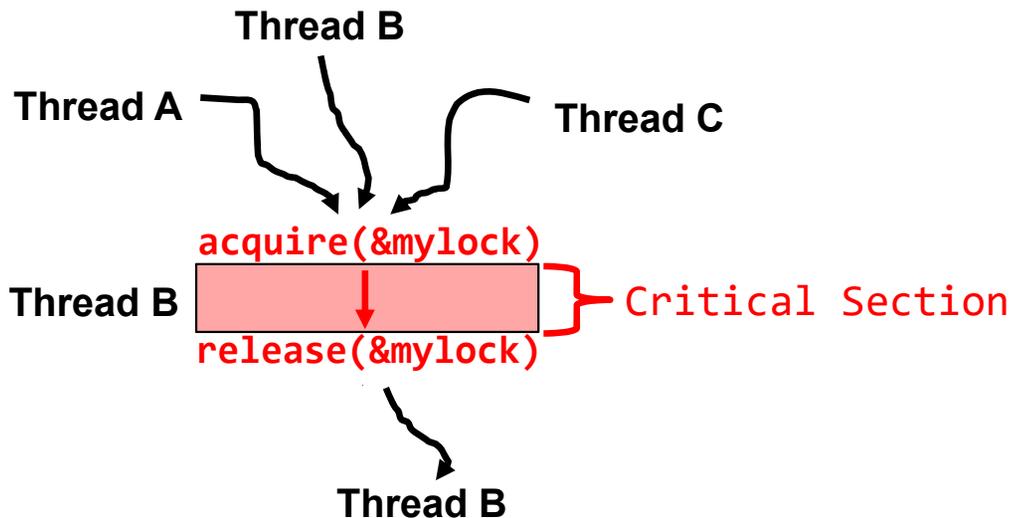


Thread T's switch returns to Thread S
[Thread T on Ready queue,
Thread S is Running]

Recall: Fix banking problem with Locks!

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

```
Deposit(acctId, amount) {  
  acquire(&mylock)           // Wait if someone else in critical section!  
  acct = GetAccount(acctId);  
  acct->balance += amount;  
  StoreAccount(acct);  
  release(&mylock)          // 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!

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

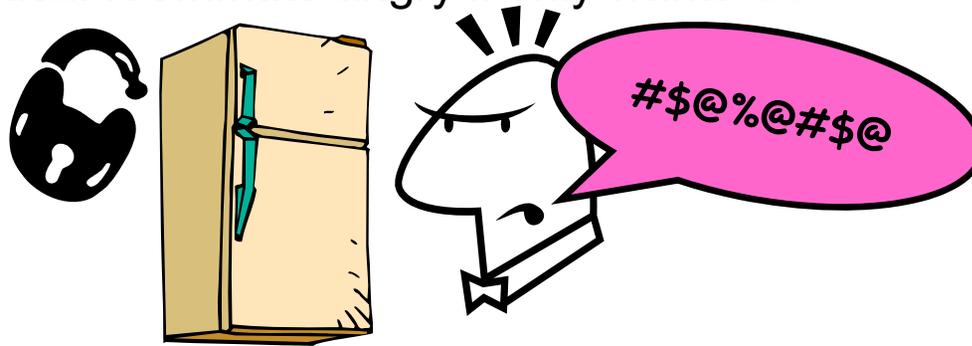
Solve with a lock?

- **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!

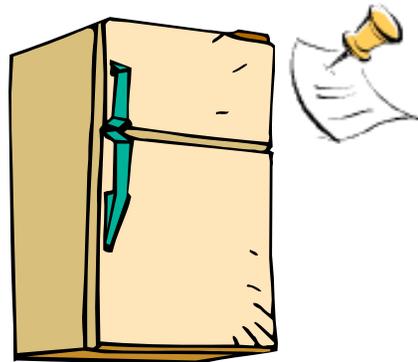
Too Much Milk: Correctness Properties

- 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):

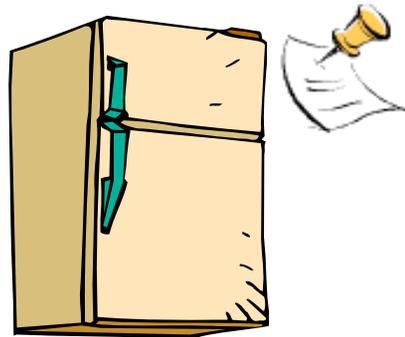
```
Thread A  
if (noMilk) {  
  
    if (noNote) {  
        leave Note;  
        buy Milk;  
        remove Note;  
    }  
}
```

```
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



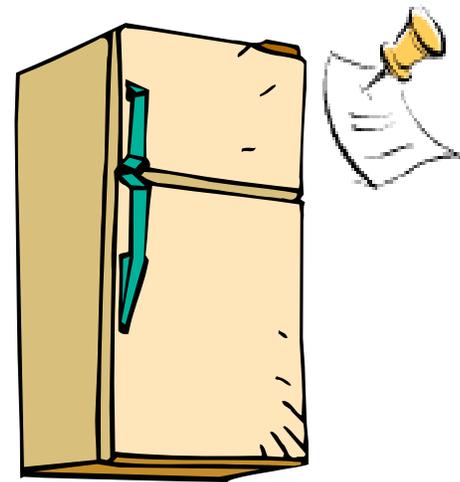
Too Much Milk Solution #2

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

<u>Thread A</u>	<u>Thread B</u>
leave note A;	leave note B;
if (noNote B) {	if (noNoteA) {
if (noMilk) {	if (noMilk) {
buy Milk;	buy Milk;
}	}
}	}
remove note A;	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

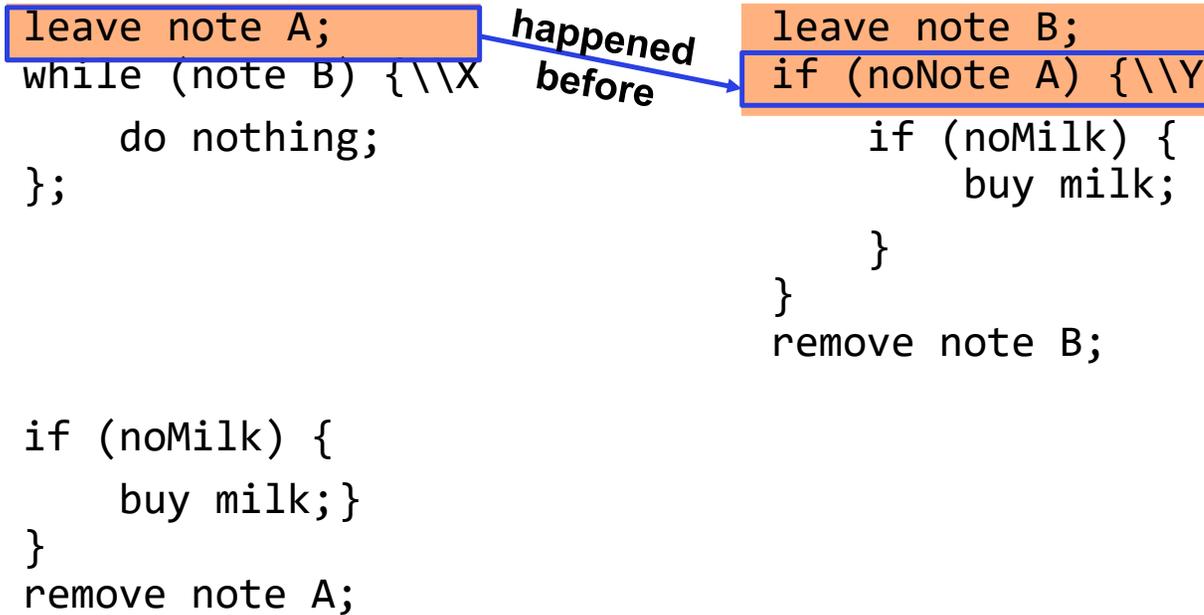
- Here is a possible two-note solution:

<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

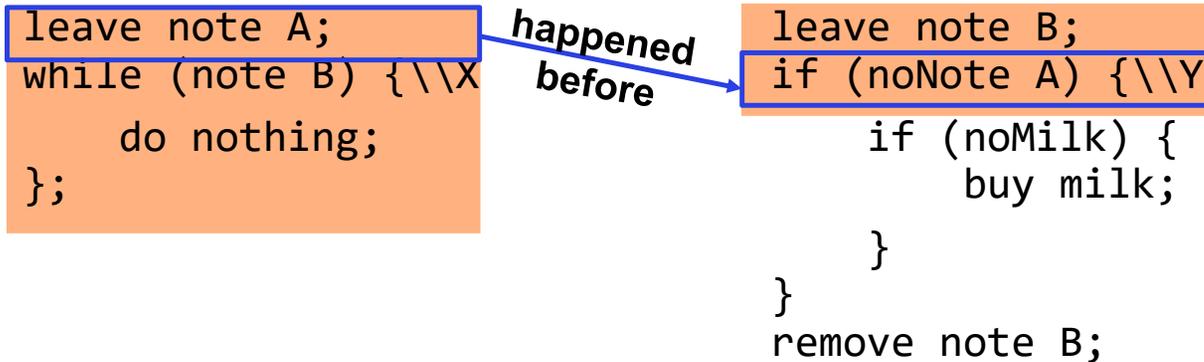
Case 1

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



Case 1

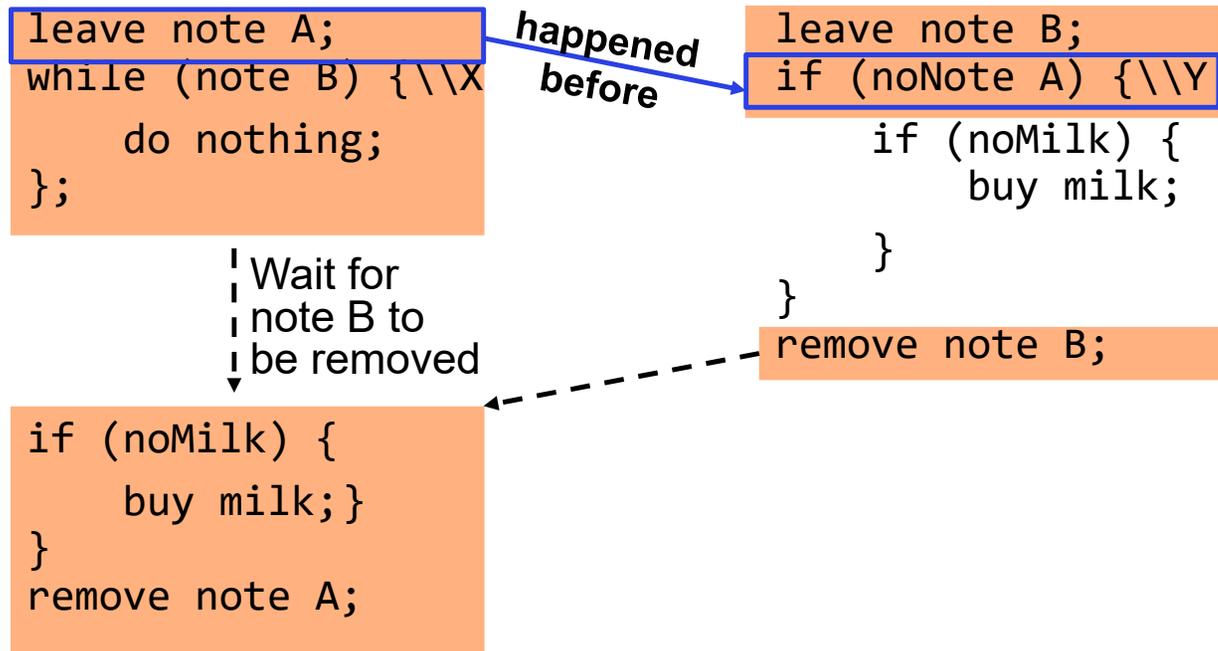
- “leave note A” happens before “if (noNote A)”



```
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”

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

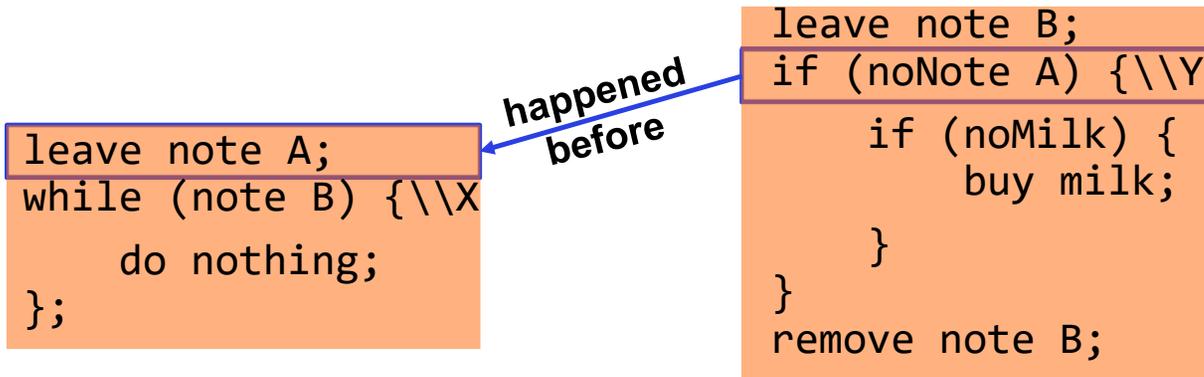
if (noMilk) {
    buy milk;
}
remove note A;
```

happened before

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

Case 2

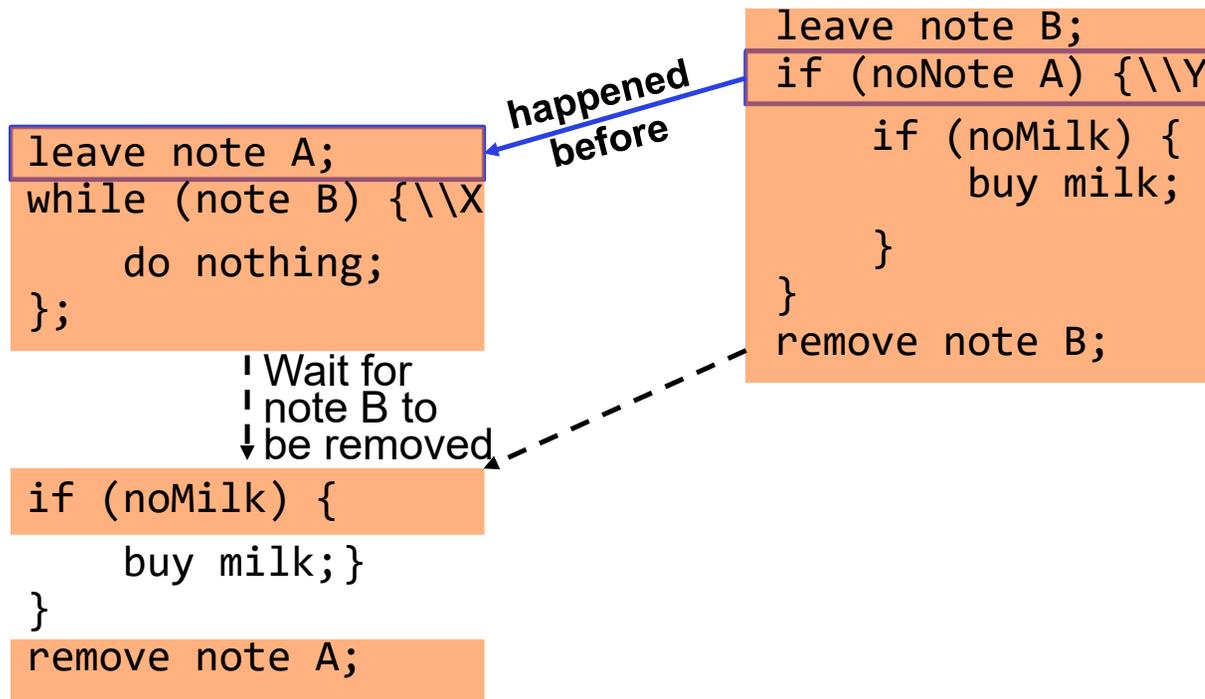
- “if (noNote A)” happens before “leave note A”



```
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

Administrivia

- Midterm Next Thursday (February 15, 8-10pm)!
 - 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 Saturday
- 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?

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: 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 (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?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        Enable Position? → 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?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        Enable Position? → 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?

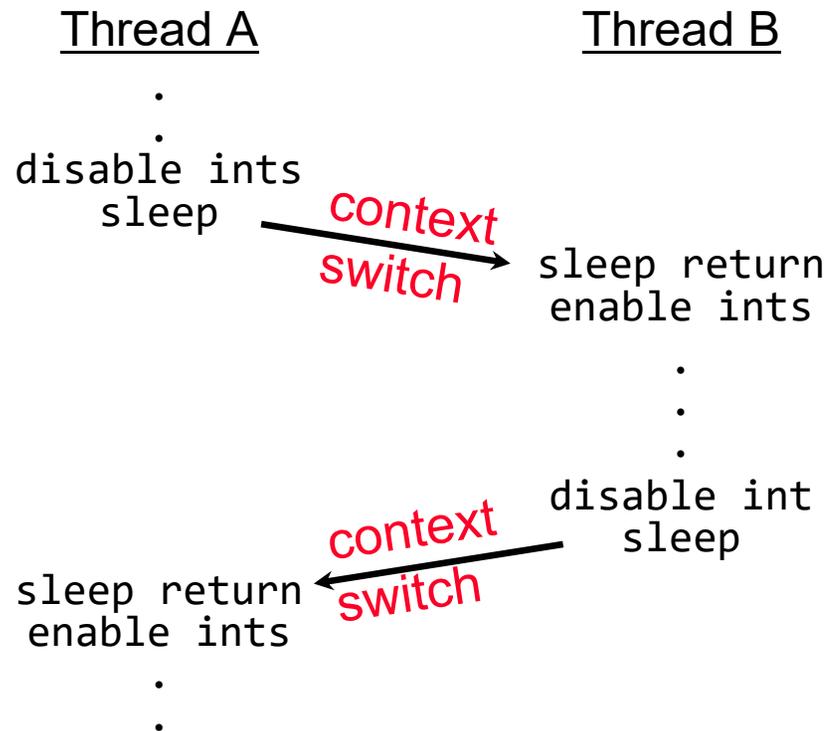
```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Enable Position? 

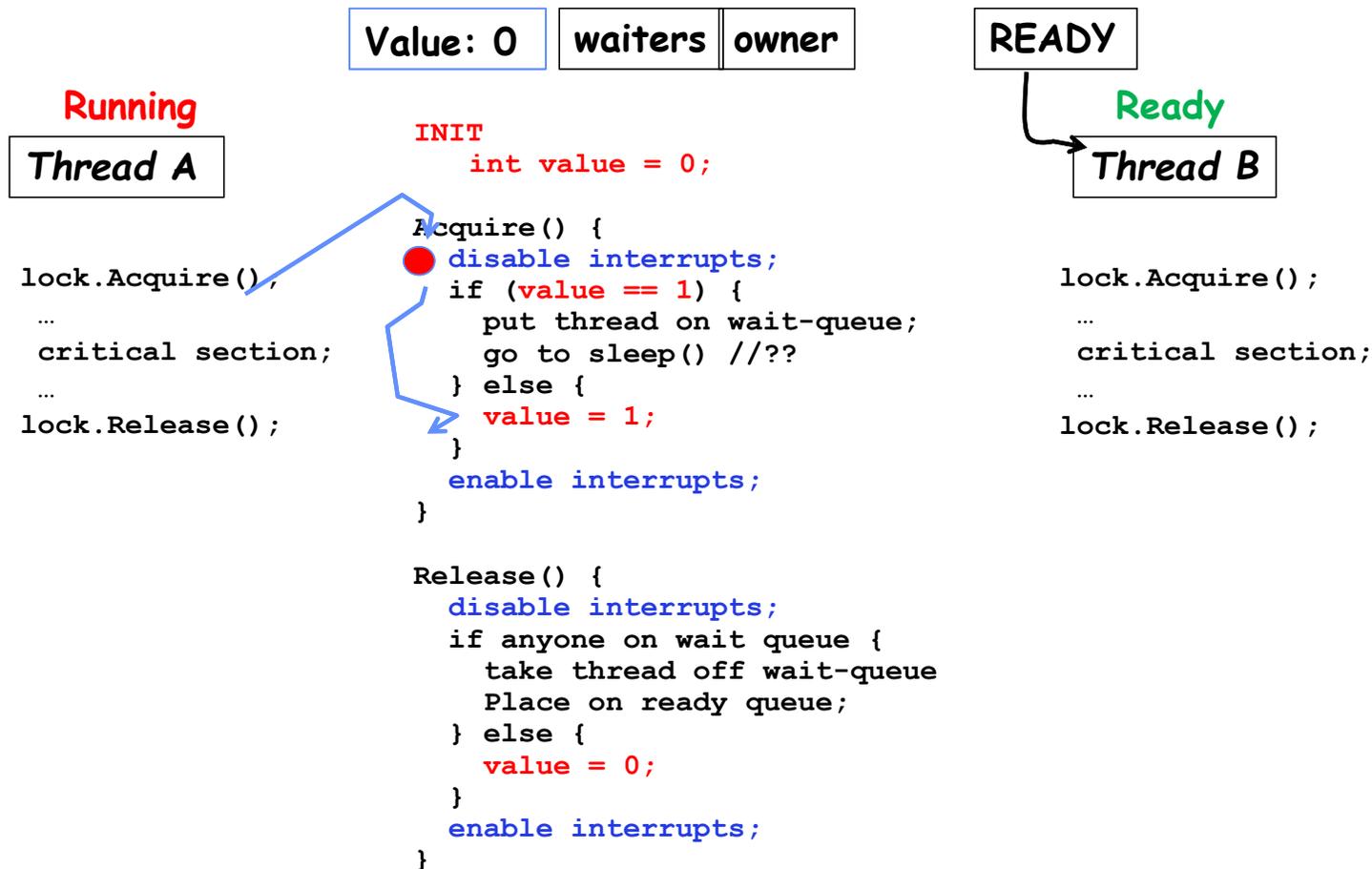
- Before Putting thread on the wait queue
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!)
- Want to put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

How to Re-enable After Sleep()? ---

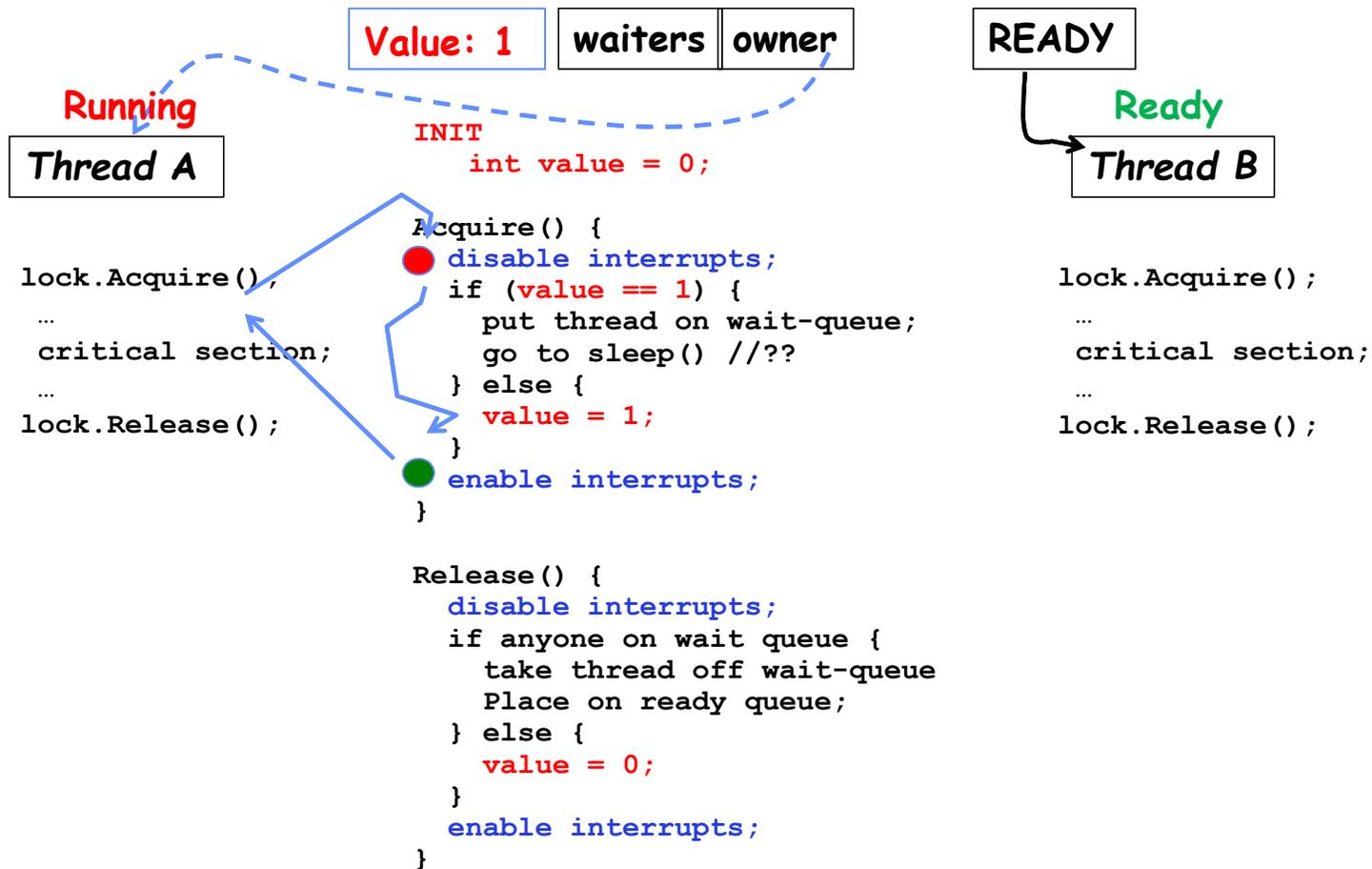
- In scheduler, since interrupts are disabled when you call sleep:
 - Responsibility of the next thread to re-enable ints
 - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



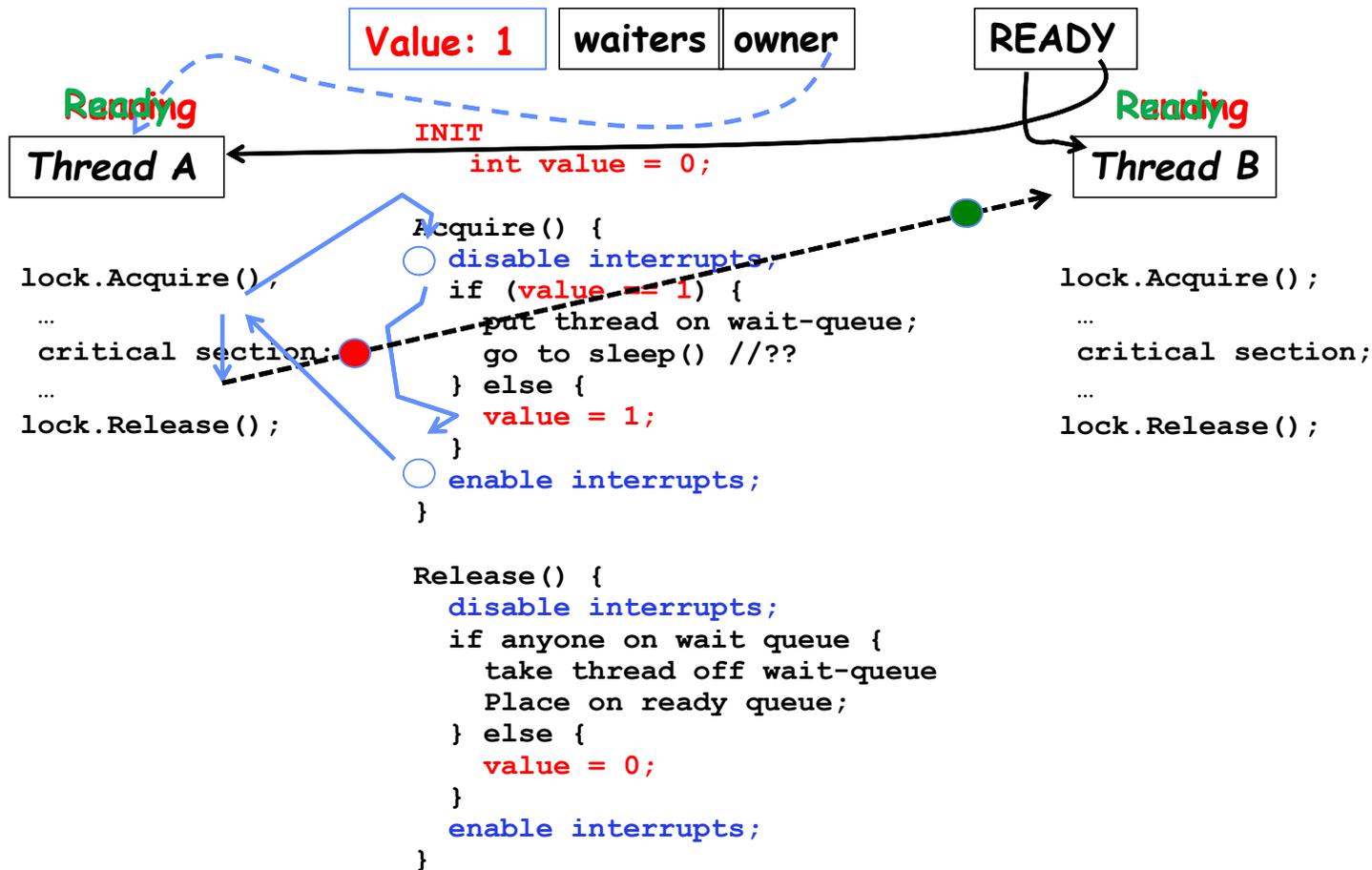
In-Kernel Lock: Simulation



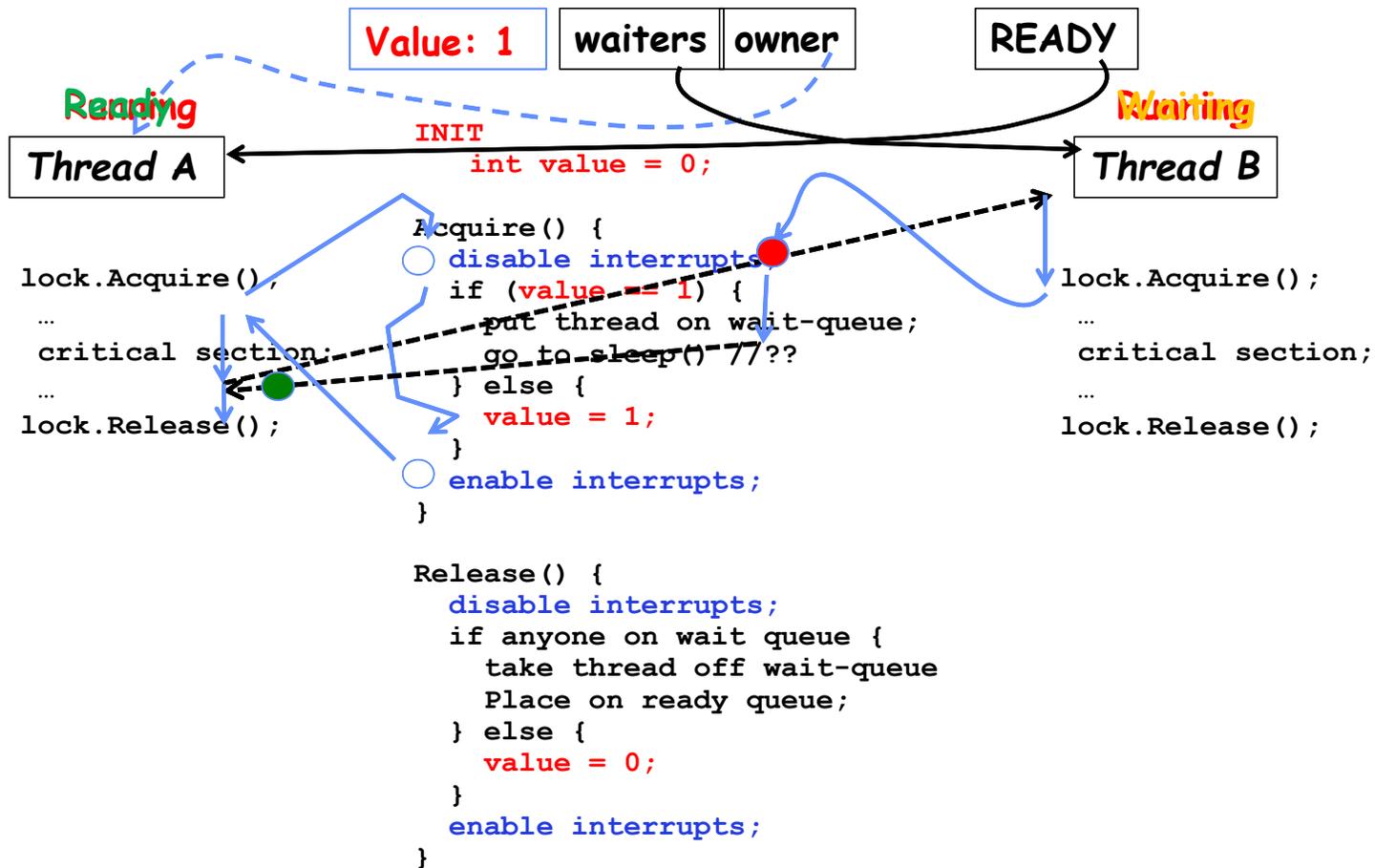
In-Kernel Lock: Simulation



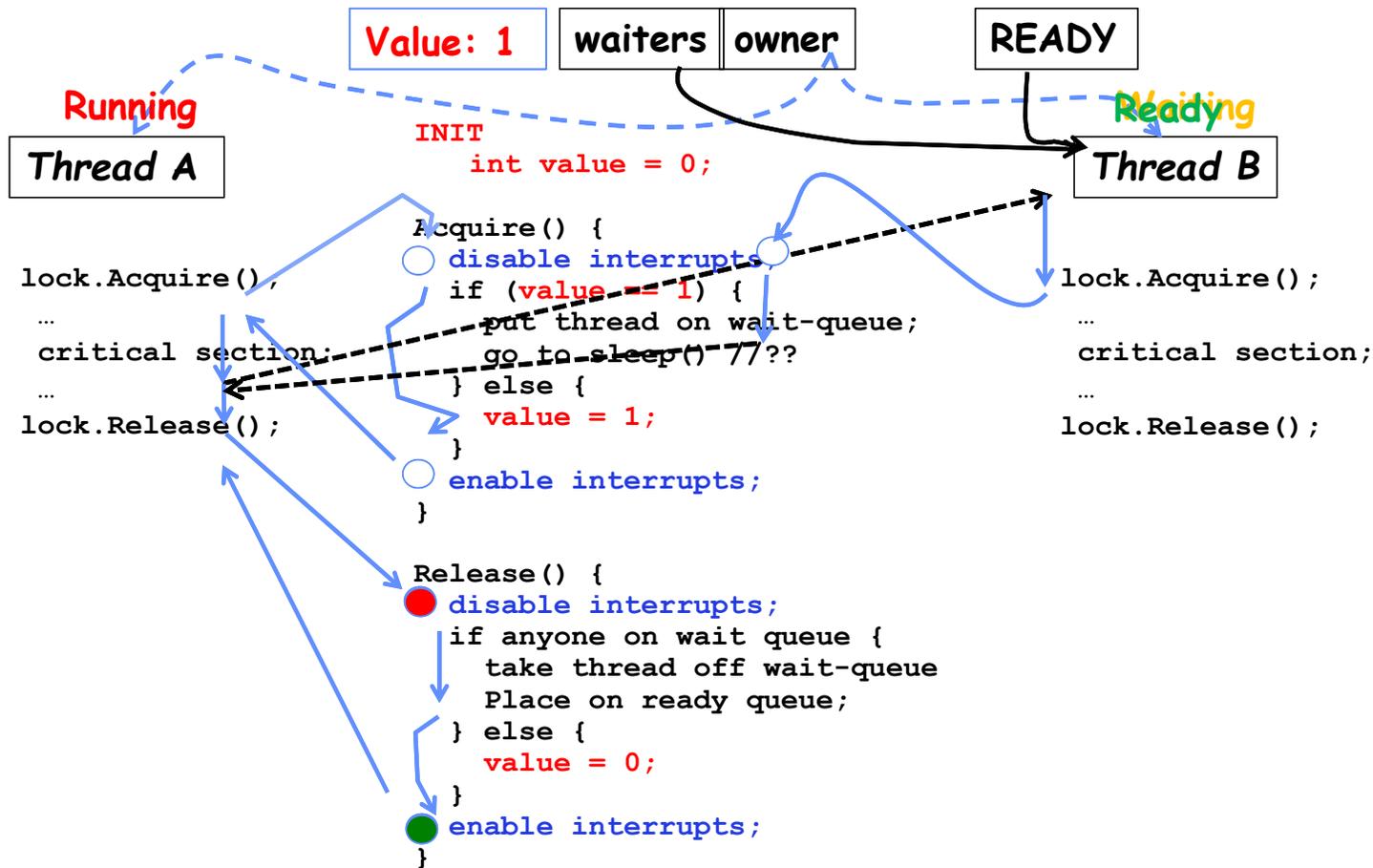
In-Kernel Lock: Simulation



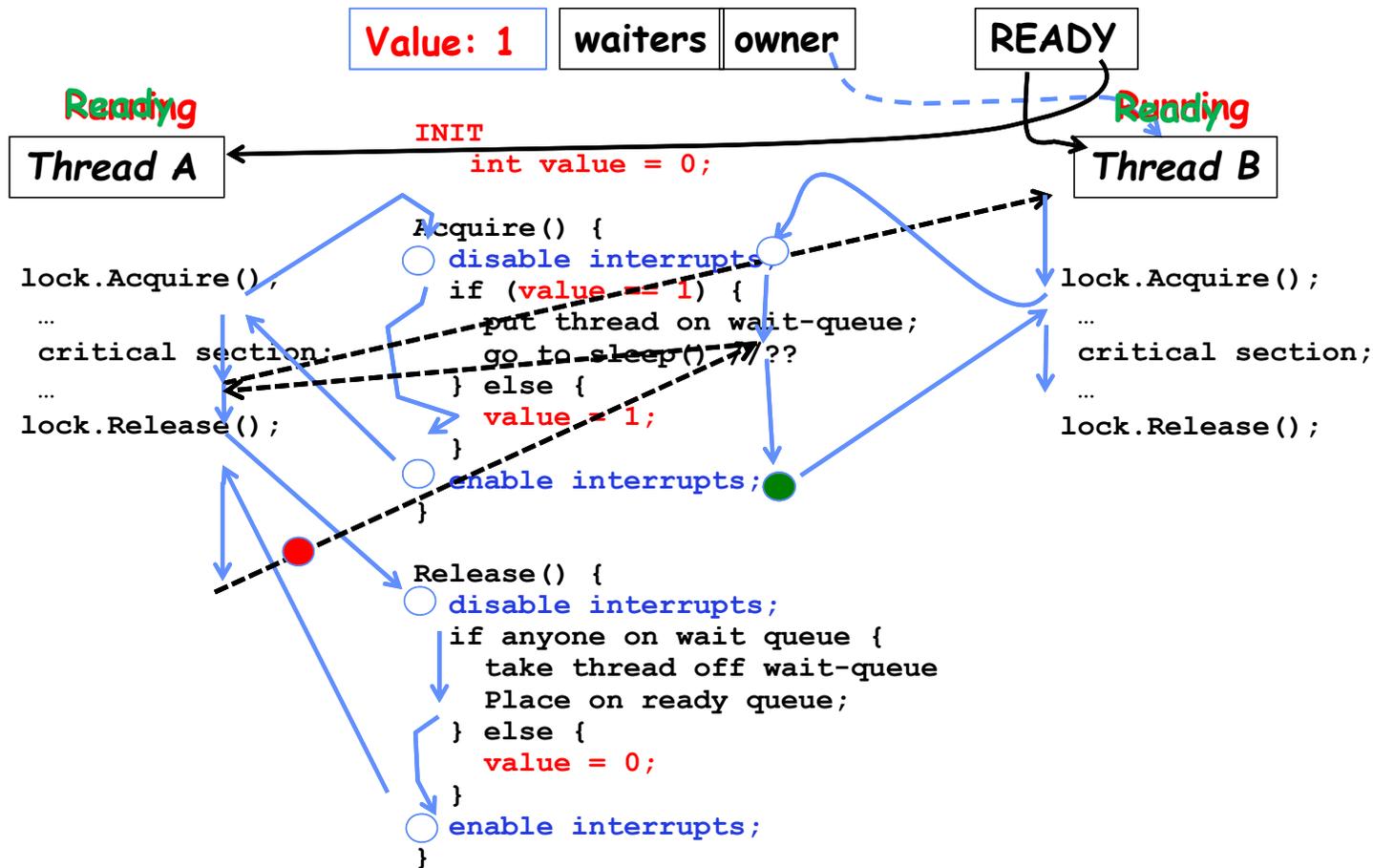
In-Kernel Lock: Simulation



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

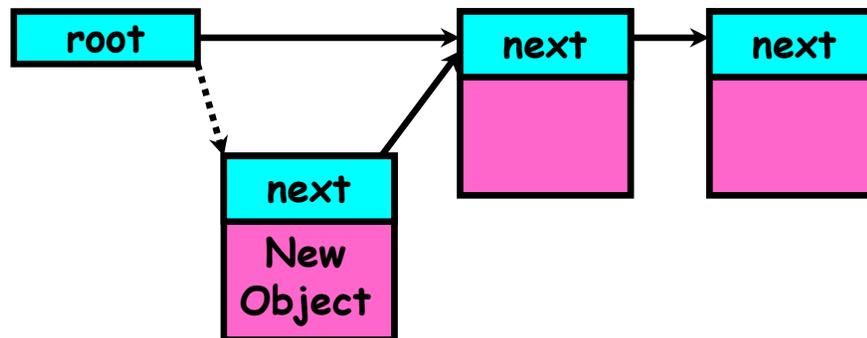
- `test&set (&address) {` `/* most architectures */`
 `result = M[address];` `// return result from "address" and`
 `M[address] = 1;` `// set value at "address" to 1`
 `return result;`
}
- `swap (&address, register) {` `/* x86 */`
 `temp = M[address];` `// swap register's value to`
 `M[address] = register;` `// 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;`
 `}`

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



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 mylock = 0; // Interface: acquire(&mylock);  
                //                release(&mylock);  
  
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
⇒ no progress!
- Priority Inversion problem with original Martian rover
- For higher-level synchronization primitives (e.g. semaphores or monitors), waiting thread 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 mylock = 0; // Interface: acquire(&mylock);  
                //                release(&mylock);  
  
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 mylock = FREE; // Interface: acquire(&mylock);  
                                //           release(&mylock);
```

```
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

```
int mylock=0;
acquire(&mylock);
...
critical section;
...
release(&mylock);
```

```
acquire(int *thelock) {
    disable interrupts;
}
```

```
release(int *thelock)
{
    enable interrupts;
}
```

```
acquire(int *thelock) {
    // Short busy-wait time
    disable interrupts;
    if (*thelock == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        *thelock = 1;
        enable interrupts;
    }
}
```

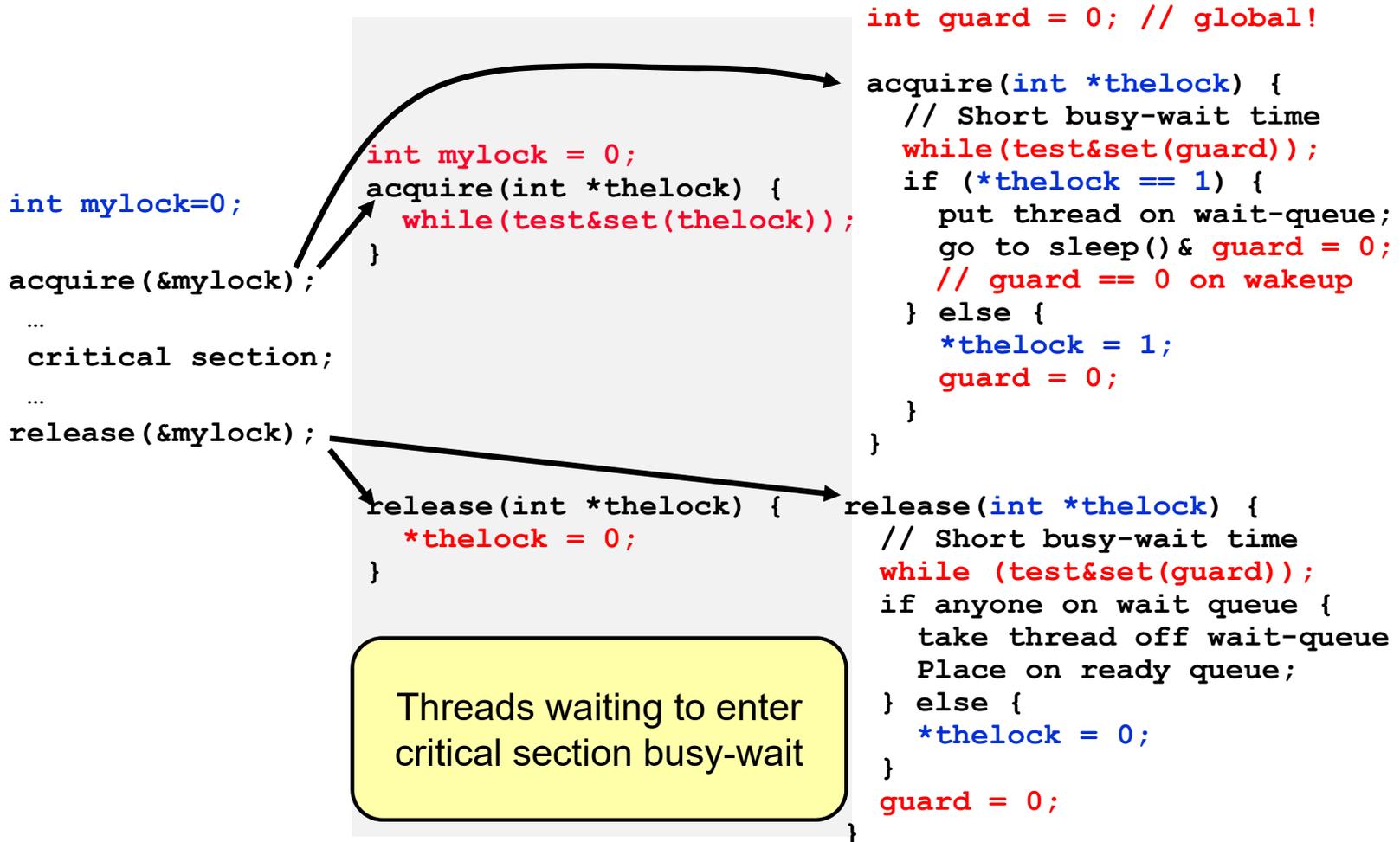
```
release(int *thelock) {
    // Short busy-wait time
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        *thelock = 0;
    }
    enable interrupts;
}
```

If one thread in critical section, no other activity (including OS) can run!

Lock argument not used!

The diagram illustrates the implementation of a lock using interrupts. It shows a sequence of operations: declaring a lock variable, acquiring it, entering a critical section, and releasing it. The acquisition and release functions are shown in two versions: a simplified one and a more detailed one. The detailed version shows that during acquisition, interrupts are disabled, and if the lock is already held, the thread is put on a wait-queue and goes to sleep. When the lock is released, interrupts are disabled, and if there are threads on the wait-queue, they are moved to the ready queue. The lock argument is not used in the simplified version, but the callout box explains that this is because the lock is held by the thread in the critical section, and no other activity can occur.

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 pthreads**
 - 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

```
bool maybe_waiters = false;
int 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

- Much better: Three (3) states:
 - **UNLOCKED**: No one has lock
 - **LOCKED**: One thread has lock
 - **CONTESTED**: Possibly more than one (with someone sleeping)
- Clean interface!
- Lock grabbed cleanly by either
 - **compare&swap()**
 - First **swap()**
- No overhead if uncontested!
- Could 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);
}
```

Recall: Where are we going with synchronization?

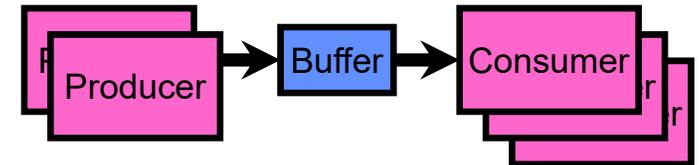
Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

- 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

Producer-Consumer with a Bounded Buffer

- Problem Definition

- Producer(s) put things into a shared buffer
- Consumer(s) take them out
- Need synchronization to coordinate producer/consumer



- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them

- Need to synchronize access to this buffer
- Producer needs to wait if buffer is full
- Consumer needs to wait if buffer is empty

- Example 1: GCC compiler

- `cpp | cc1 | cc2 | as | ld`

- Example 2: Coke machine

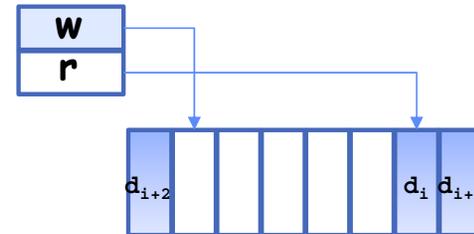
- Producer can put limited number of Cokes in machine
- Consumer can't take Cokes out if machine is empty

- Others: Web servers, Routers,



Bounded Buffer Data Structure (sequential case)

```
typedef struct buf {  
    int write_index;  
    int read_index;  
    <type> *entries[BUFSIZE];  
} buf_t;
```



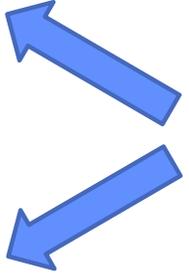
- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- *How to tell if Full (on insert) Empty (on remove)?*
- *And what do you do if it is?*
- *What needs to be atomic?*

Bounded Buffer – first cut

mutex buf_lock = <initially unlocked>

```
Producer(item) {  
    acquire(&buf_lock);  
    while (buffer full) {}; // Wait for a free slot  
    enqueue(item);  
    release(&buf_lock);  
}
```

```
Consumer() {  
    acquire(&buf_lock);  
    while (buffer empty) {}; // Wait for arrival  
    item = dequeue();  
    release(&buf_lock);  
    return item  
}
```



Will we ever come out
of the wait loop?

Bounded Buffer – 2nd cut



mutex buf_lock = <initially unlocked>

```
Producer(item) {  
  acquire(&buf_lock);  
  while (buffer full) {release(&buf_lock); acquire(&buf_lock);}  
  enqueue(item);  
  release(&buf_lock);  
}
```

```
Consumer() {  
  acquire(&buf_lock);  
  while (buffer empty) {release(&buf_lock); acquire(&buf_lock);}  
  item = dequeue();  
  release(&buf_lock);  
  return item  
}
```

What happens when one is waiting for the other?
- Multiple cores ?
- Single core ?

Higher-level Primitives than Locks

- Goal of last couple of lectures:
 - What is right abstraction for synchronizing threads that share memory?
 - Want as high a level primitive as possible!
- Good primitives and practices important!
 - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
 - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
 - This lecture and the next presents a some ways of structuring sharing

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