# Operating Systems
# (Honor Track)

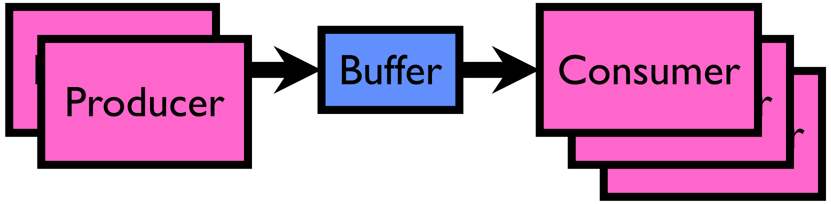
# Synchronization 3: Lock Implementation, Atomic Instructions, Monitors

Xin Jin

Spring 2023

Acknowledgments: Ion Stoica, Berkeley CS 162

# Recap: 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: 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, ….

# Recap: Circular 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?

# Recap: Full Solution to Bounded Buffer (coke machine)

```
Semaphore fullSlots = 0;        // Initially, no coke
Semaphore emptySlots = bufSize;
                                // Initially, num empty slots
Semaphore mutex = 1;            // No one using machine

Producer(item) {
    semaP(&emptySlots);         // Wait until space
    semaP(&mutex);              // Wait until machine free
    Enqueue(item);
    semaV(&mutex);
    semaV(&fullSlots);          // Tell consumers there is
                                // more coke
}
Consumer() {
    semaP(&fullSlots);          // Check if there's a coke
    semaP(&mutex);              // Wait until machine free
    item = Dequeue();
    semaV(&mutex);
    semaV(&emptySlots);         // tell producer need more
    return item;
}
```

Critical sections using mutex protect integrity of the queue

`fullSlots` signals coke

`emptySlots` signals space

# Recap: 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

# Recap: Motivating Example: "Too Much Milk"

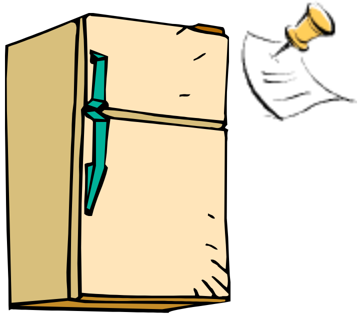
- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

| 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 |

# Recap: 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;
   }
}
```



# Recap: 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

# Recap: 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 (noNote A) {
   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

# Recap: Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave Note A;
while (Note B) {\\X
   do nothing;
}
if (noMilk) {
   buy Milk;
}
remove Note A;
```

Thread B

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

- 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

# Recap: 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?

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

- 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

# 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?

# 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

# 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
- 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?

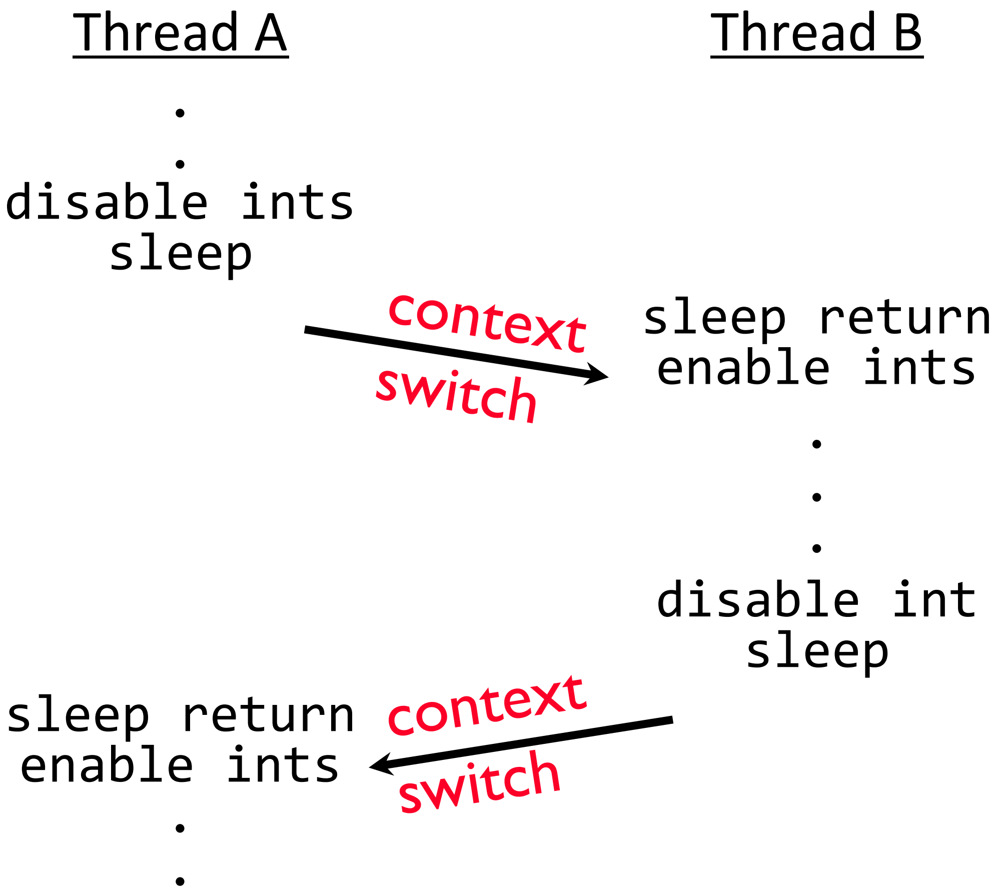
```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep();
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

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



```
      Thread A                      Thread B

          .
          .
    disable ints
       sleep
                               sleep return
                               enable ints
                                    .
                                    .
                                    .
                                disable int
                                   sleep
    sleep return
    enable ints
          .
          .
```

# In-Kernel Lock: Simulation

| Value: 0 | waiters | owner | READY |
|---|---|---|---|

Running

*Thread A*

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

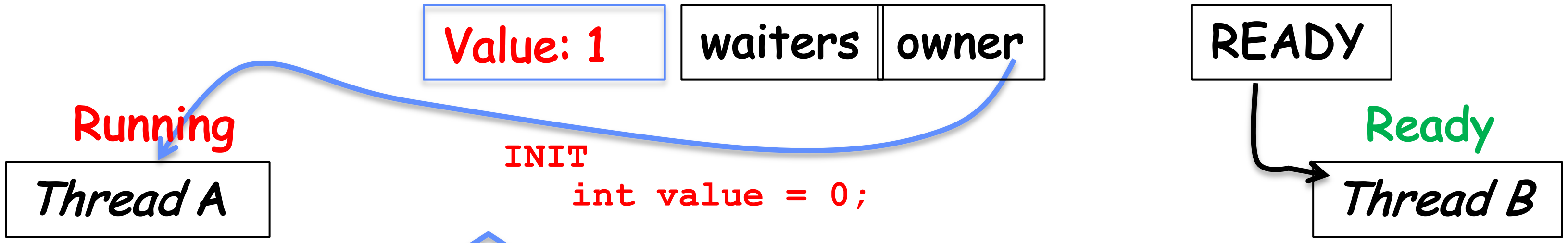
```
INIT
   int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

Ready

*Thread B*

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

# In-Kernel Lock: Simulation



```
INIT
   int value = 0;

Acquire() {
●  disable interrupts;
   if (value == 1) {
     put thread on wait-queue;
     go to sleep() //??
   } else {
     value = 1;
   }
●  enable interrupts;
}

Release() {
   disable interrupts;
   if anyone on wait queue {
     take thread off wait-queue
     Place on ready queue;
   } else {
     value = 0;
   }
   enable interrupts;
}
```

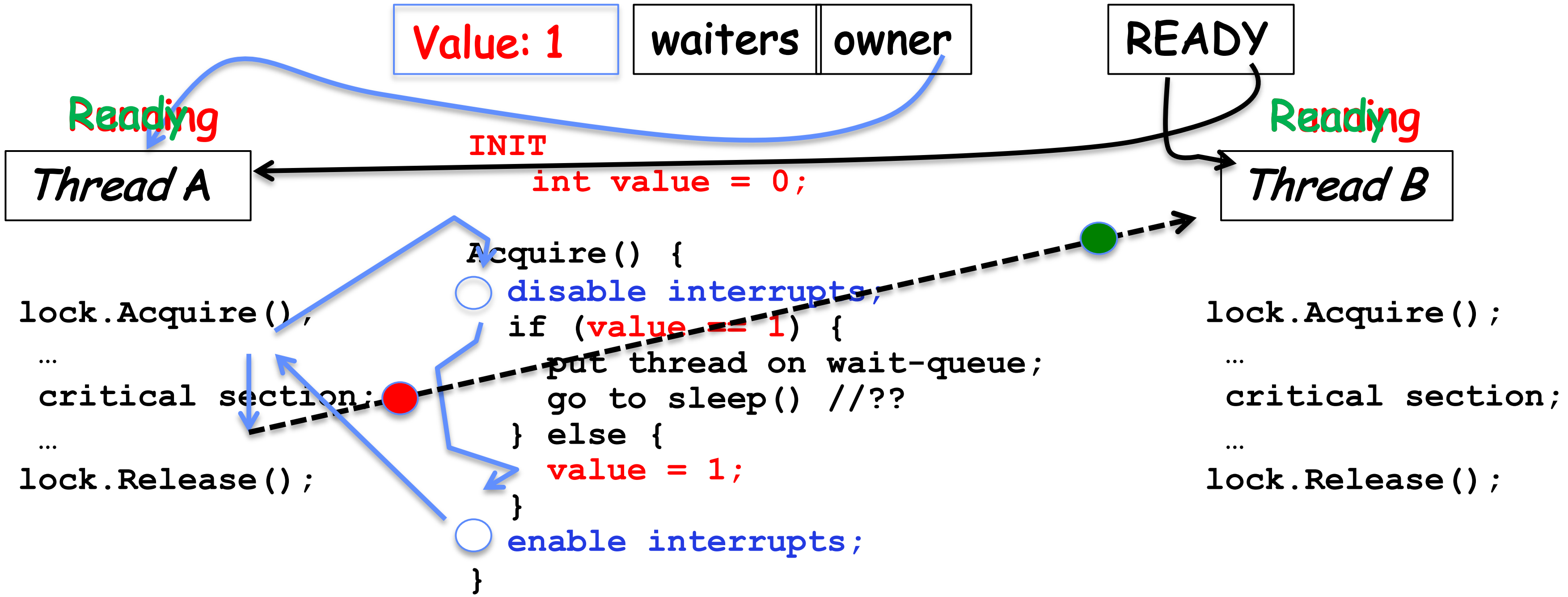
Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

# In-Kernel Lock: Simulation



INIT

```
int value = 0;
```

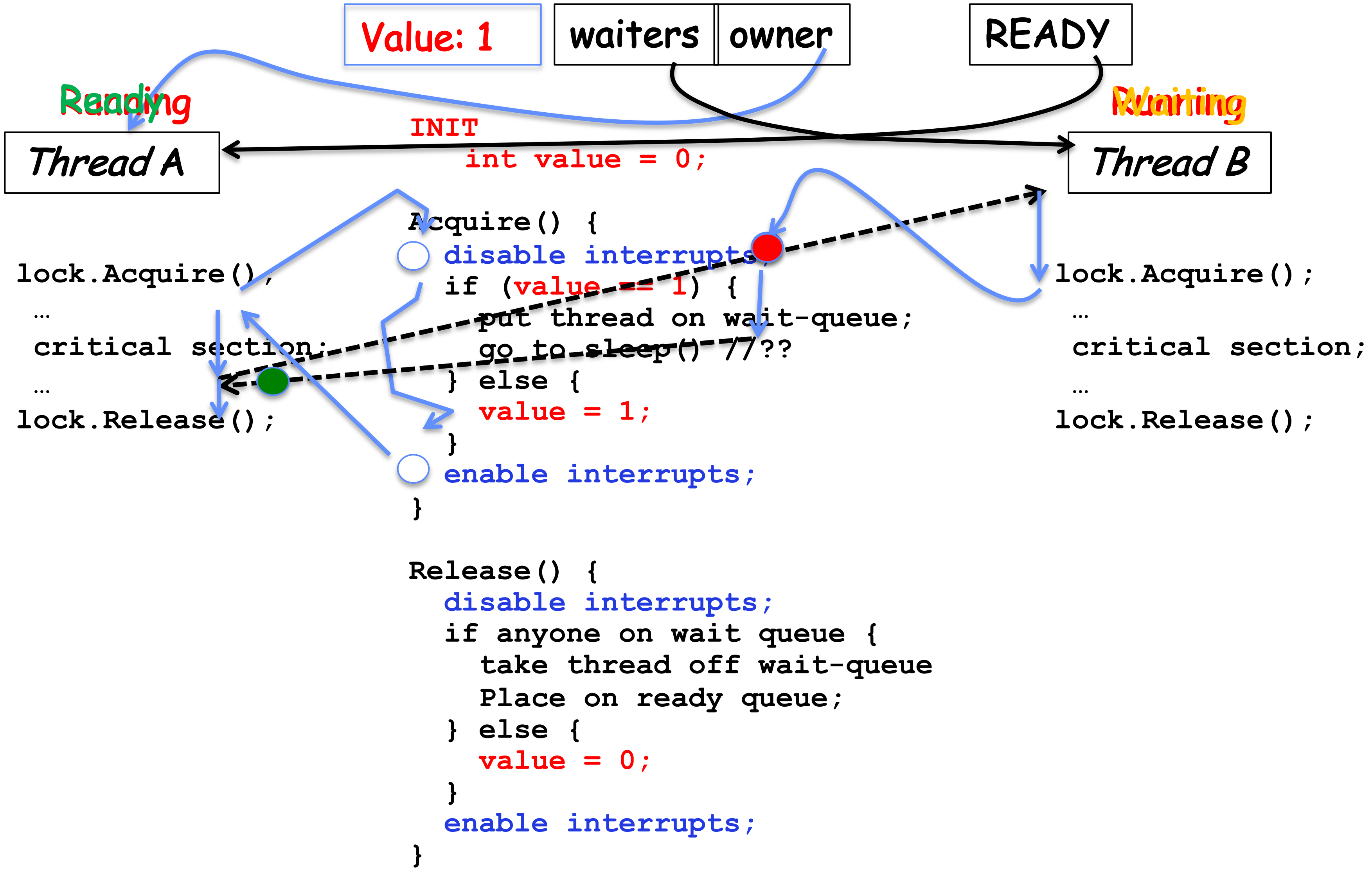
```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

# In-Kernel Lock: Simulation



```
INIT
    int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

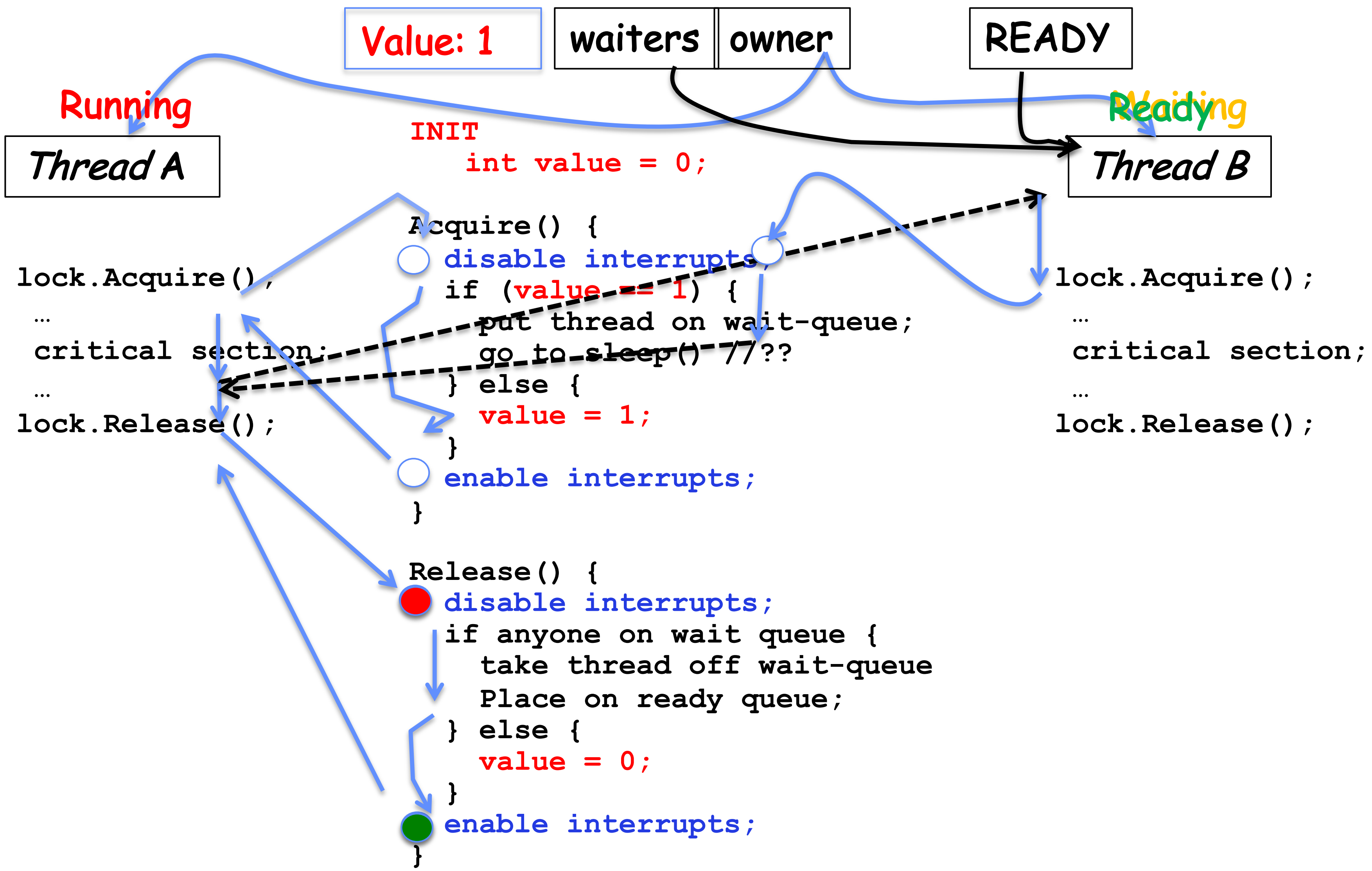
Thread A:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

Thread B:

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

# In-Kernel Lock: Simulation

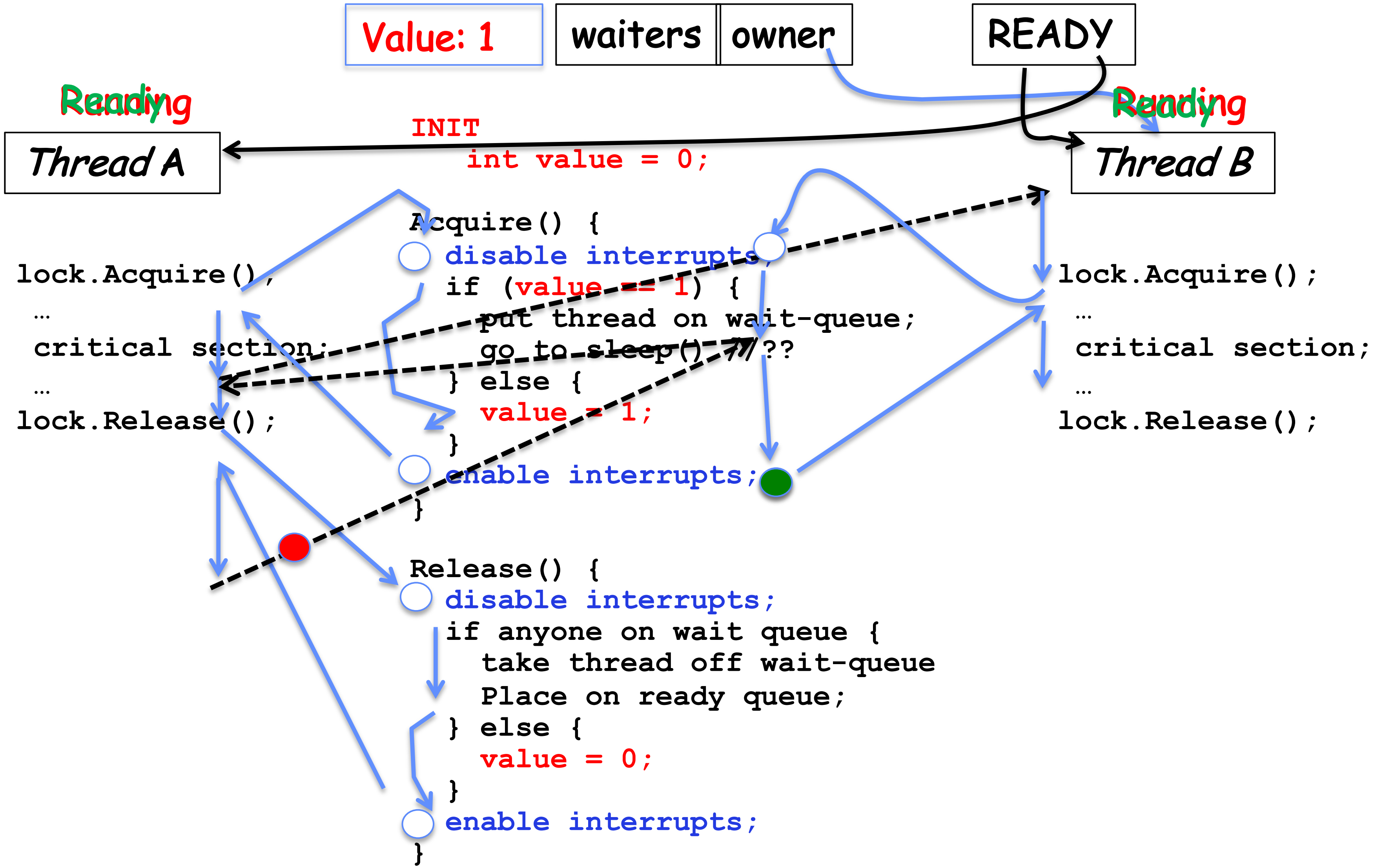


```
INIT
   int value = 0;

Acquire() {
○ disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
○ enable interrupts;
}

Release() {
● disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
● enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

# In-Kernel Lock: Simulation



```
int value = 0;
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
  }
  enable interrupts;
}
```

```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait-queue
    Place on ready queue;
  } else {
    value = 0;
  }
  enable interrupts;
}
```

# 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;
}
```

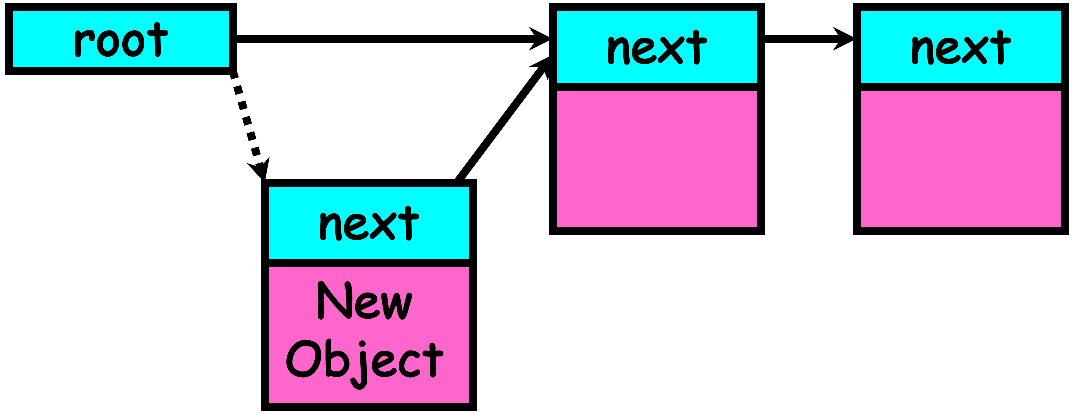
```
compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {   // If memory still == reg1,
        M[address] = reg2;      // then  put reg2 => memory
        return success;
    } else {                    // Otherwise do not change memory
        return failure;
    }
}
```

# 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 {                        // 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));
  }
```



# 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 ⇒ 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

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

- 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

# 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

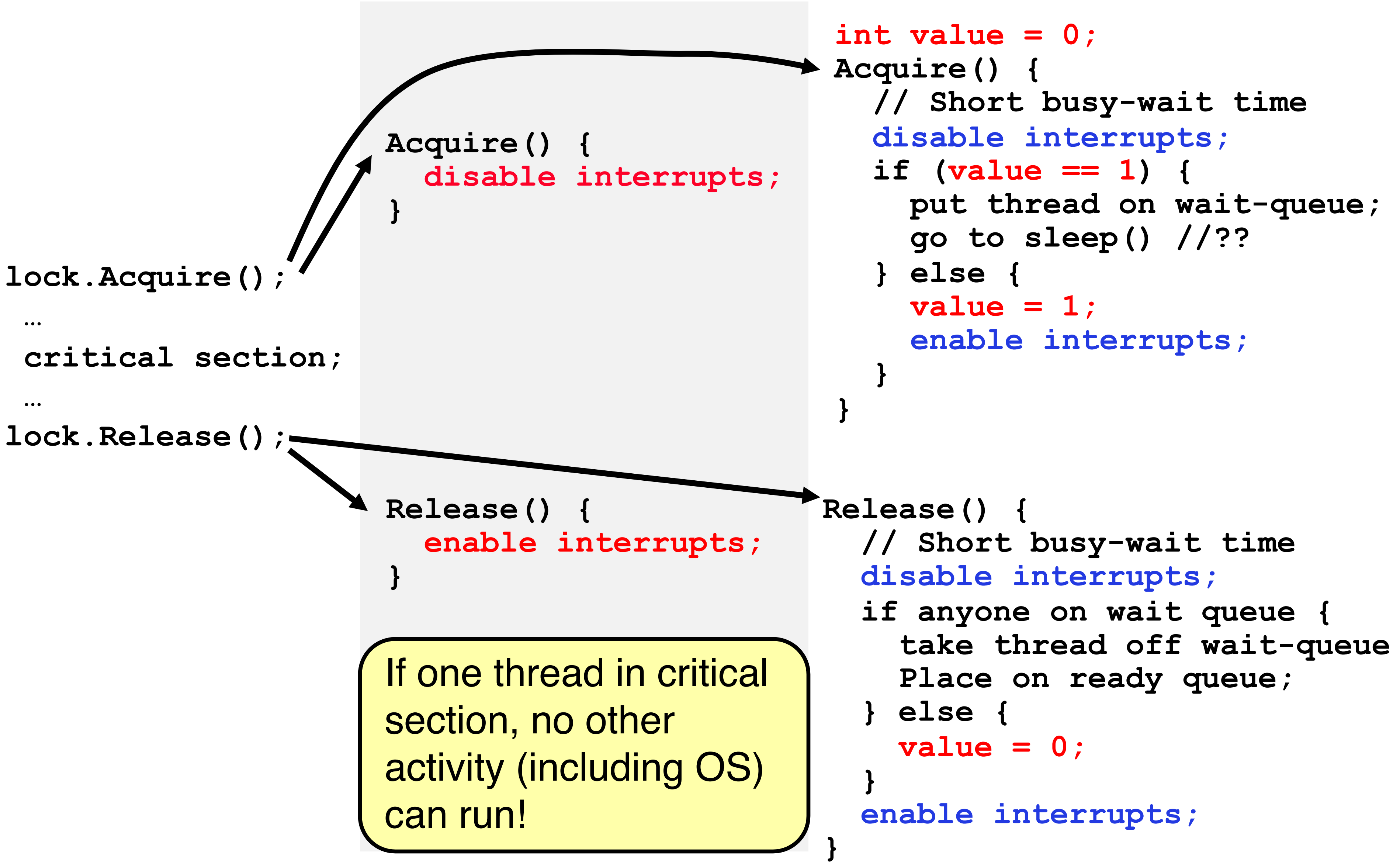
```
int value = FREE;

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



```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
Acquire() {
  disable interrupts;
}
```

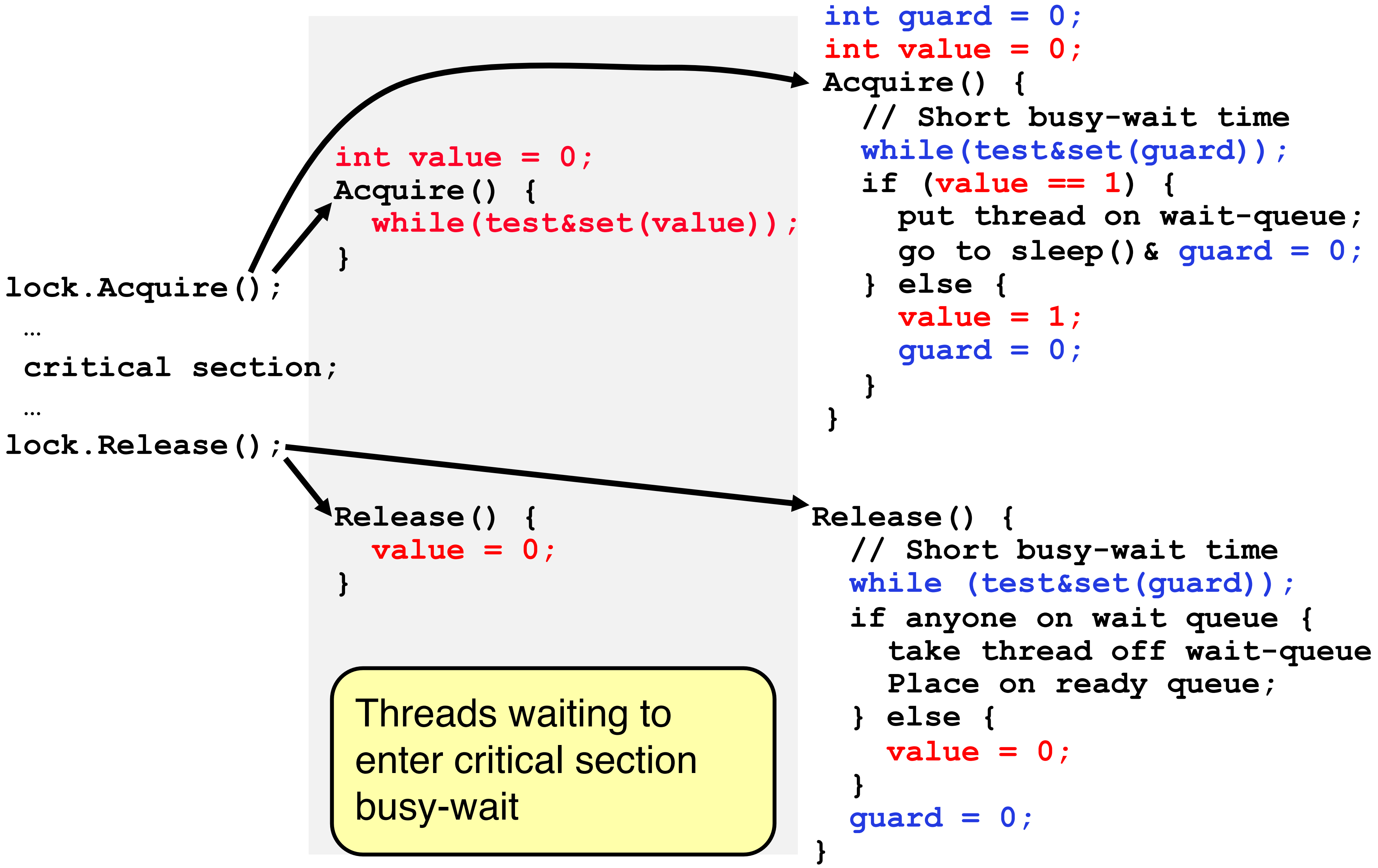
```
Release() {
  enable interrupts;
}
```

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

```
int value = 0;
Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait-queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}
```

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

# Recap: Locks using test & set



```
lock.Acquire();
 …
 critical section;
 …
lock.Release();
```

```
int value = 0;
Acquire() {
  while(test&set(value));
}
```

```
Release() {
  value = 0;
}
```

Threads waiting to enter critical section busy-wait

```
int guard = 0;
int value = 0;
Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait-queue;
    go to sleep()& guard = 0;
  } else {
    value = 1;
    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 = 0;
  }
  guard = 0;
}
```

# 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

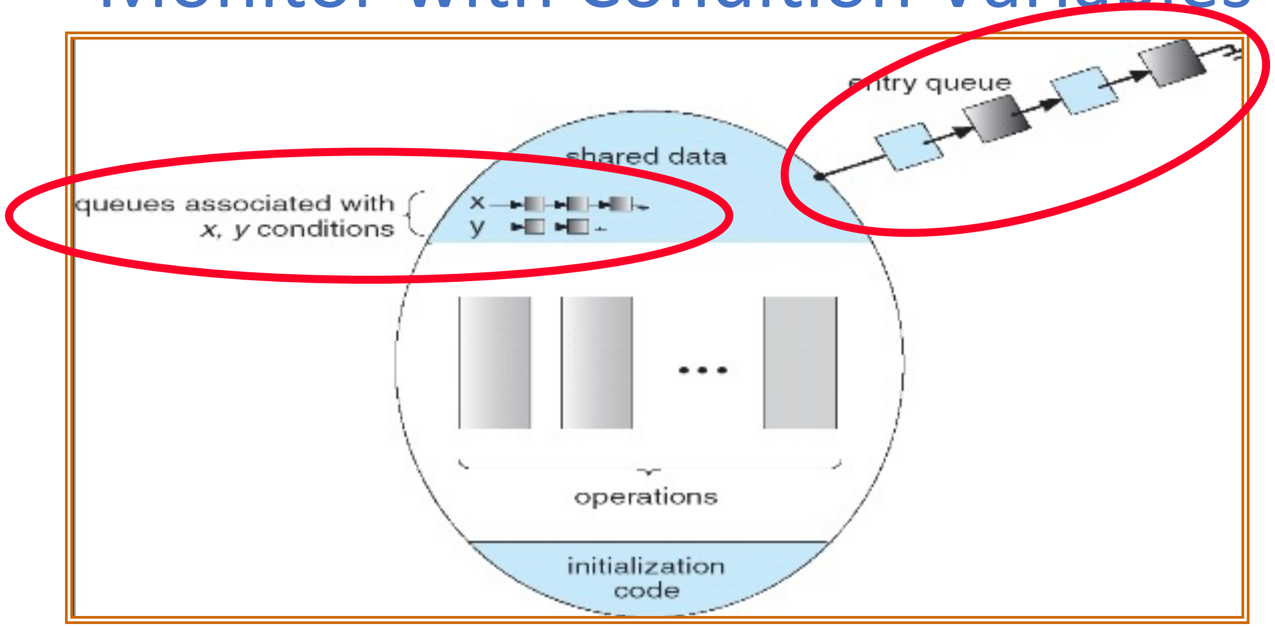
# Semaphores are good but…Monitors are better!

- 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);
}
```

# Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (isEmpty(&queue)) {
   cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue);   // Get next item
```

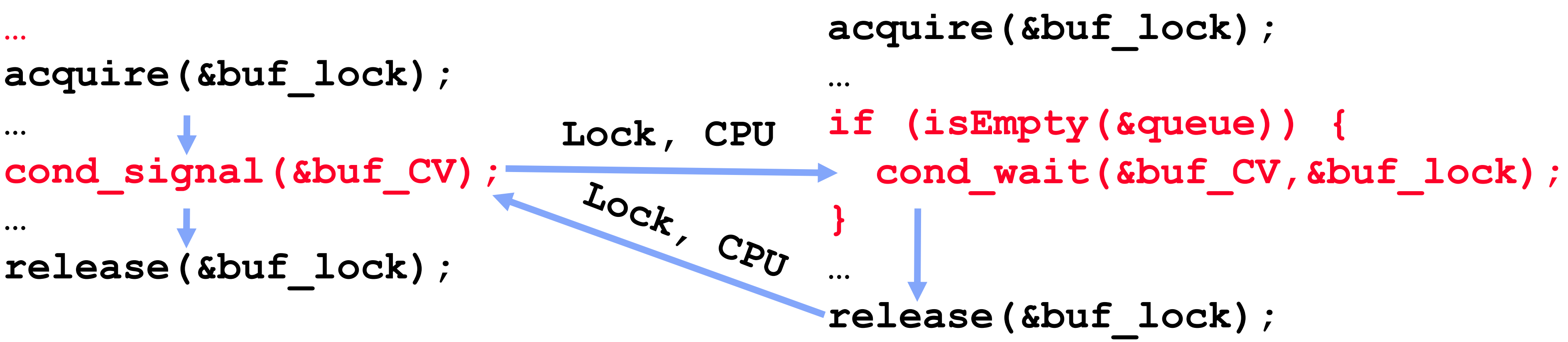
  - Why didn’t we do this?

```
if (isEmpty(&queue)) {
   cond_wait(&buf_CV,&buf_lock); // If nothing, sleep
}
item = dequeue(&queue);   // Get next item
```

- Answer: depends on the type of scheduling
  - Mesa-style: Named after Xerox-Park Mesa Operating System
    - Most OSes use Mesa Scheduling!
  - Hoare-style: Named after British logician Tony Hoare

# Hoare monitors

- Signaler gives up lock, CPU to waiter; waiter runs immediately
- Then, Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again



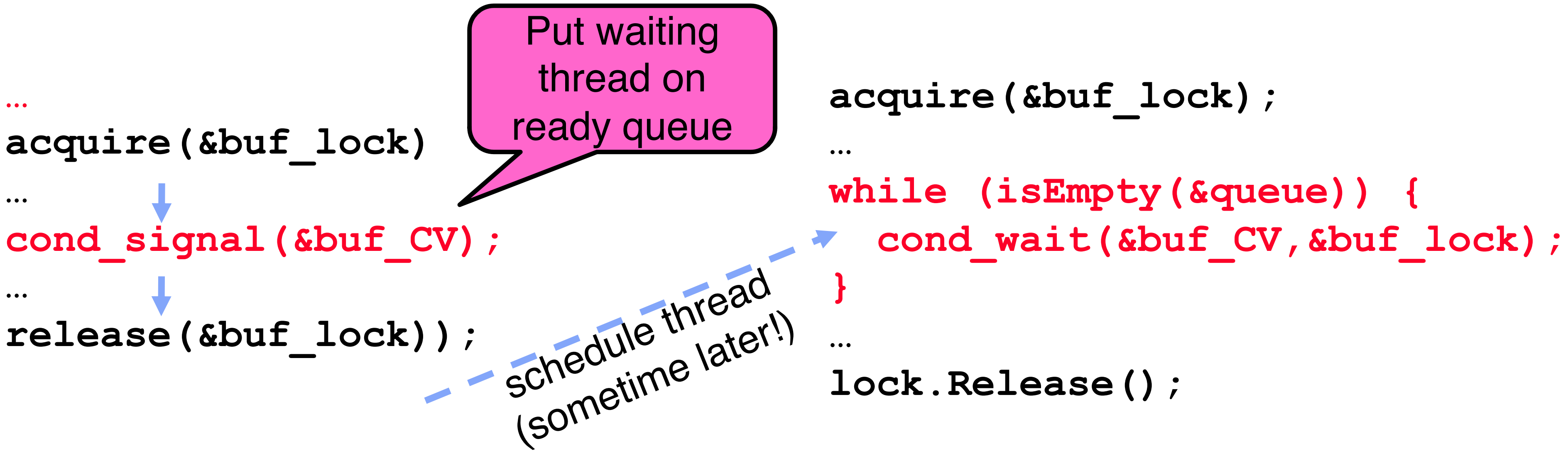
```
…
acquire(&buf_lock);
…
cond_signal(&buf_CV);
…
release(&buf_lock);
```

```
acquire(&buf_lock);
…
if (isEmpty(&queue)) {
  cond_wait(&buf_CV,&buf_lock);
}
…
release(&buf_lock);
```

- On first glance, this seems like good semantics
  - Waiter gets to run immediately, condition is still correct!
- Most textbooks talk about Hoare scheduling
  - However, hard to do, not really necessary!
  - Forces a lot of context switching (inefficient!)

# Mesa monitors

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority



```
…
acquire(&buf_lock)
…
cond_signal(&buf_CV);
…
release(&buf_lock));
```

Put waiting thread on ready queue

schedule thread (sometime later!)

```
acquire(&buf_lock);
…
while (isEmpty(&queue)) {
  cond_wait(&buf_CV,&buf_lock);
}
…
lock.Release();
```

- Practically, need to check condition again after wait
  - By the time the waiter gets scheduled, condition may be false again – so, just check again with the “while” loop
- Most real operating systems do this!
  - More efficient, easier to implement
  - Signaler’s cache state, etc. still good

# Circular Buffer – 3rd cut (Monitors, pthread-like)

```
lock buf_lock = <initially unlocked>
condition producer_CV = <initially empty>
condition consumer_CV = <initially empty>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&producer_CV, &buf_lock); }
  enqueue(item);
  cond_signal(&consumer_CV);
  release(&buf_lock);
}

Consumer() {
  acquire(buf_lock);
  while (buffer empty) { cond_wait(&consumer_CV, &buf_lock); }
  item = dequeue();
  cond_signal(&producer_CV);
  release(buf_lock);
  return item
}
```

What does thread do when it is waiting?
 - Sleep, not busywait!

# Group Discussion

- Topic: synchronization APIs
  - How to implement producer-consumer with a circular buffer with locks, semaphores and monitors?
  - What are the pros and cons of each solution?

- 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

# 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