

# Operating Systems

## Lecture 11

### lock and condition variables implementation

Prof. Mengwei Xu

# Recap: Atomic Operations

---

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- **Atomic Operation (原子操作)**: 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

# Motivation: “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: Definitions

---

- **Synchronization (同步)**: 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 (互斥)**: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- **Critical Section (临界区)**: piece of code that only one thread can execute at once.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

# Recap: Lock

---

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – 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
- 3 formal properties
  - **Mutual exclusion**: at most one thread holds the lock
  - **Progress**: if no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock
  - **Bounded waiting**: if thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does
    - ❑ Yet, it does not promise that waiting threads acquire the lock in FIFO order.

# Some Advices

- Always acquire the lock at the beginning of a method and release it right before the return
  - Consistent behavior makes it easier to program
  - Also makes it easier to read and debug
- A case: double-checked locking

```
Singleton* Singleton::instance() {  
    if (pInstance == NULL) {  
        pInstance = new Instance();  
    }  
    return pInstance;  
}
```

An unsafe solution

```
Singleton* Singleton::instance() {  
    lock.acquire();  
    if (pInstance == NULL) {  
        pInstance = new Instance();  
    }  
    lock.release();  
    return pInstance;  
}
```

A safe solution

```
Singleton* Singleton::instance() {  
    if (pInstance == NULL) {  
        lock.acquire();  
        if (pInstance == NULL) {  
            pInstance = new Instance();  
        }  
        lock.release();  
    }  
    Return pInstance;  
}
```

An ``optimized'' solution.

Is it safe?

# A Tricky (but Real) Case

```
Singleton* Singleton::instance() {  
    if (pInstance == NULL) {  
        lock.acquire();  
        if (pInstance == NULL) {  
            pInstance = new Instance();  
        }  
        lock.release();  
    }  
    Return pInstance;  
}
```

Reordered by  
compiler

## Thread A

```
if (pInstance == NULL) { // True  
    lock.acquire();  
    if (pInstance == NULL) {  
        // malloc for pInstance;  
        // point pInstance to the memory;  
        // run new() function;  
    }  
    lock.release();  
}  
return pInstance;
```

## Thread B

if (pInstance == NULL); // False  
return pInstance; // uninitialized!

# 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



# 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?
    - ☐ How do you handle 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
  - Preventing external events by disabling interrupts

Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }  
LockRelease { enable Ints; }
```

# Naïve use of Interrupt Enable/Disable: Problems

---

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?

# Better Implementation of Locks

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

```
class Lock {  
    int value = FREE;  
    Queue wait_q;  
}
```

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

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



# Better Implementation of Locks

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

```
class Lock {  
    int value = FREE;  
    Queue wait_q;  
}
```

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop(); WHY??  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

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



# Better Implementation of Locks

---

- 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

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

# Better Implementation of Locks

---

- 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
- Why do we need to disable interrupts at all?

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

# Better Implementation of Locks

- 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
- 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

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```



# Better Implementation of Locks

- 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
- 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
- Before Putting thread on the wait queue?

```
Enable Lock::Acquire() {  
Position  disable interrupts;  
          if (value == BUSY) {  
            put thread on wait_q;  
            next = readyList.pop();  
            cur_thread->state = WAITING;  
            thread_switch(current, next);  
          } else {  
            value = BUSY;  
          }  
          enable interrupts;  
}
```

# Better Implementation of Locks

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


```
Enable Lock::Acquire() {  
Position  disable interrupts;  
          if (value == BUSY) {  
            put thread on wait_q;  
            next = readyList.pop();  
            cur_thread->state = WAITING;  
            thread_switch(current, next);  
          } else {  
            value = BUSY;  
          }  
          enable interrupts;  
}
```

# Better Implementation of Locks

- 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
- 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
- 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

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

Enable  
Position

A red arrow points from the text 'Enable Position' to the line 'next = readyList.pop();' in the code block.

# Better Implementation of Locks

- 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
- 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
- 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!)
  - Note: the **value** is BUSY now!!!

```
Lock::Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait_q;  
        next = readyList.pop();  
        cur_thread->state = WAITING;  
        thread_switch(current, next);  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

Enable  
Position  
→

# How to Re-enable After thread\_switch()?

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

<u>Thread A</u>	<u>Thread B</u>
.	
disable ints	
thread_switch	
	thread_switch return
	enable ints
	.
	.
	.
	disable int
	thread_switch
thread_switch return	
enable ints	
.	
.	

# Atomic Read-Modify-Write Instructions

---

- Can we extend the lock implementation to multi-processors?
  - Not good idea, as 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) {`  
    `result = M[address];`  
    `M[address] = 1;`  
    `return result;`  
}
  - `swap (&address, register) {`  
    `temp = M[address];`  
    `M[address] = register;`  
    `register = temp;`  
}
  - `compare&swap (&address, reg1, reg2) {` /\* 68000 \*/  
    `if (reg1 == M[address]) {`  
        `M[address] = reg2;`  
        `return success;`  
    `} else {`  
        `return failure;`  
    `}`  
}
- /\* most architectures \*/  
/\* return result from "address" and  
set value at "address" to 1 \*/
- /\* x86 \*/  
/\* swap register's value to  
value at "address" \*/

# Implementing Locks with test&set

- Spinlock (自旋锁): 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



# 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 semaphores, waiting thread may wait for an arbitrary long time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives



# 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;
        cur_thread->state = WAITING;
        thread_switch() & 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;
}
```

# 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;
        cur_thread->state = WAITING;
        thread_switch() & 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;
}
```

Must be atomic! What if setting guard before or after thread\_switch()? How to implement?

More details in Figure 5.17 (section 5.7 “Implementing Synchronization Objects”) of our textbook

# 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;  
        thread_switch();  
        // 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 replace

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

# Implementing Condition Variables

---

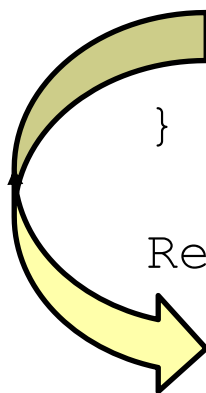
- Recap the operations:
  - `wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

```
while (!testOnSharedState()) {  
    cv.wait(&lock)  
}
```

# Synchronized Queue with Condition Variables

```
Lock lock;  
Condition dataready;  
Queue queue;
```

```
AddToQueue(item) {  
    lock.Acquire();           // Get Lock  
    queue.enqueue(item);      // Add item  
    dataready.signal();       // Signal any waiters  
    lock.Release();           // Release Lock  
}
```



```
RemoveFromQueue() {  
    lock.Acquire();           // Get Lock  
    while (queue.isEmpty()) {  
        dataready.wait(&lock); // If nothing, sleep  
    }  
    item = queue.dequeue();    // Get next item  
    lock.Release();           // Release Lock  
    return(item);  
}
```

# Implementing Condition Variables

- Recap the 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

```
Class CV {  
    Queue waiting;  
    void wait(Lock *lock);  
    void signal();  
    void broadcast();  
}  
  
void CV::signal() {  
    if (waiting.notEmpty()) {  
        thread = waiting.remove();  
        scheduler.makeReady(thread);  
    }  
}
```

```
void CV::wait(Lock *lock) {  
    assert(lock.isHeld());  
    waiting.add(currentTCB);  
    // switch to new thread and release lock  
    // in atomic manner  
    scheduler.suspend(&lock);  
    lock->acquire();  
}
```

# Mesa vs. Hoare monitors

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

```
while (queue.isEmpty()) {  
    dataready.wait(&lock); // If nothing, sleep  
}  
item = queue.dequeue(); // Get next item
```

- Why didn't we do this?

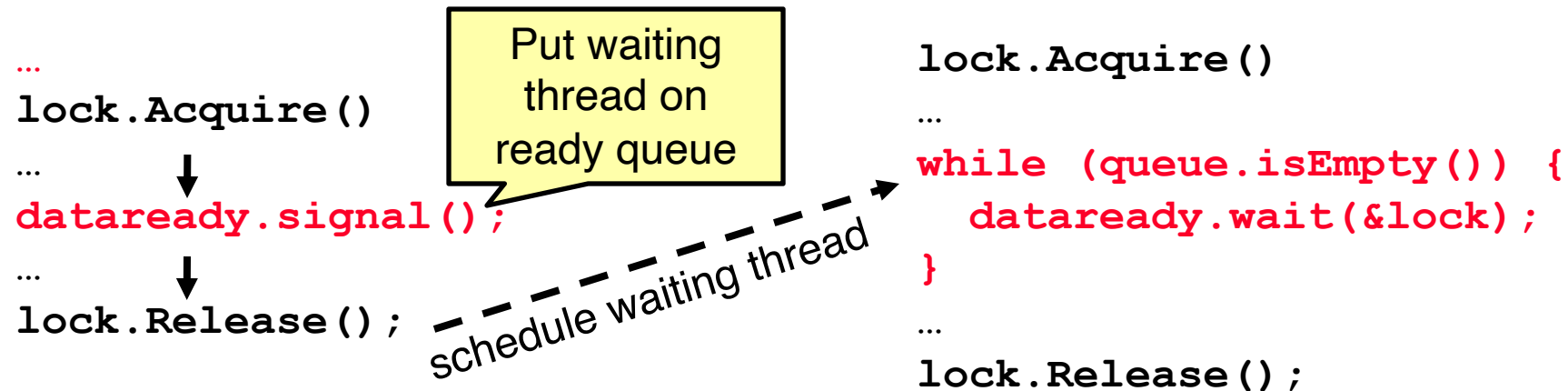
```
if (queue.isEmpty()) {  
    dataready.wait(&lock); // If nothing, sleep  
}  
item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling (管程模型)
  - Hoare-style
  - Mesa-style



# Mesa monitors

- Signaler keeps lock and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait
- Most real operating systems



# Mesa Monitor: Why “while()”?

- Why do we use “while()” instead of “if()” with Mesa monitors?
  - Example illustrating what happens if we use “if()”, e.g.,

```
if (queue.isEmpty()) {  
    dataready.wait(&lock); // If nothing, sleep  
}
```

- We'll use the synchronized (infinite) queue example

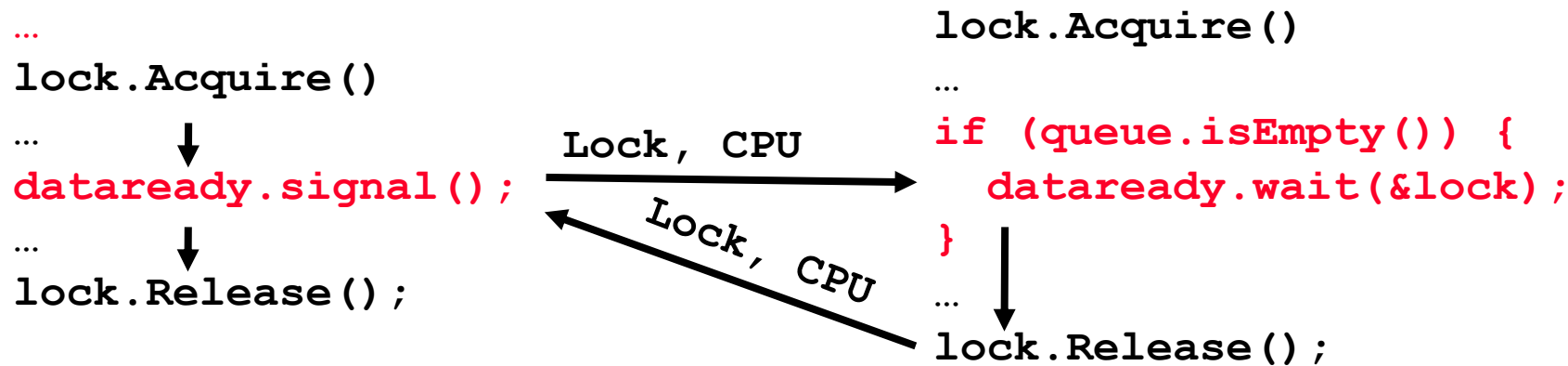
```
AddToQueue(item) {  
    lock.Acquire();  
    queue.enqueue(item);  
    dataready.signal();  
    lock.Release();  
}
```

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

Replace “while” with  
“if”

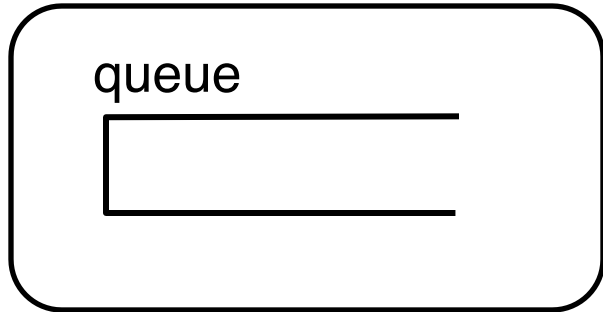
# Hoare monitors

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



# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE  
dataready → NULL  
queue

CPU State

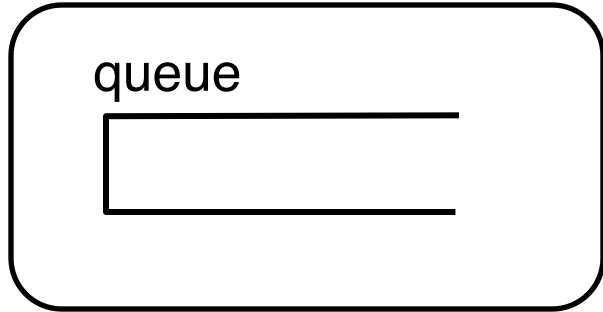
Running: T1  
Ready  
queue → NULL  
...

T1 (Running)

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **BUSY (T1)**  
dataready → NULL  
queue → NULL

CPU State

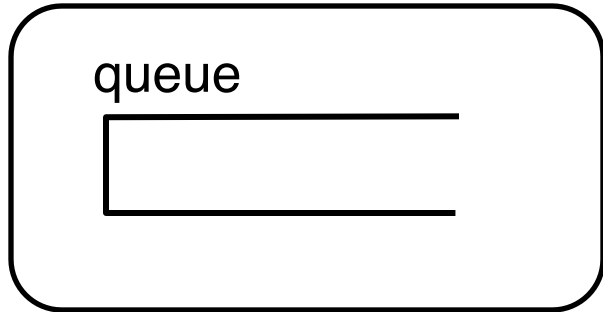
Running: T1  
Ready  
queue → NULL  
...

T1 (**Running**)

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **FREE**

dataready  
queue → **T1**

CPU State

Running:  
Ready  
queue → NULL

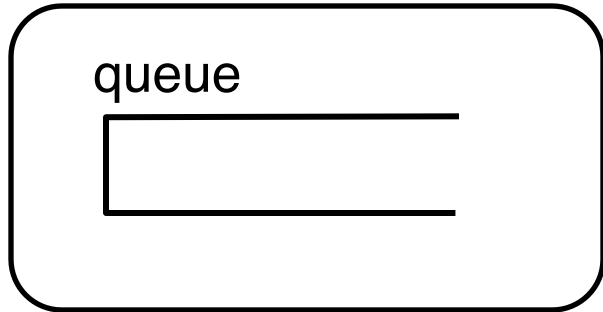
wait(&lock) puts thread on  
dataready queue and  
releases lock

T1 (**Waiting**)

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE  
dataready → T1  
queue

CPU State

Running: T2  
Ready  
queue → NULL  
...

T1 (Waiting)

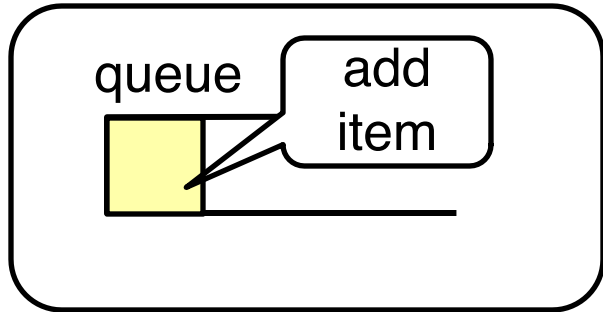
```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

T2 (Running)

```
AddToQueue(item) {  
    lock.Acquire();  
    queue.enqueue(item);  
    dataready.signal();  
    lock.Release();  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **BUSY** (T2)  
dataready → T1  
queue

CPU State

Running: T2  
Ready  
queue → NULL  
...

T1 (Waiting)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

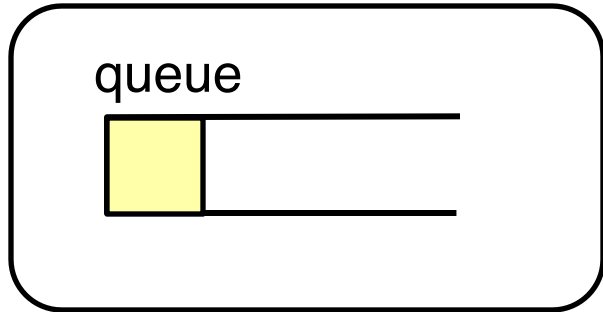
T2 (**Running**)

```
AddToQueue(item) {  
  lock.Acquire();  
  queue.enqueue(item);  
  dataready.signal();  
  lock.Release();  
}
```



# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **BUSY (T2)**  
dataready → NULL  
queue

CPU State

Running: T2  
Ready  
queue → **T1**  
...

T1 (**Ready**)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

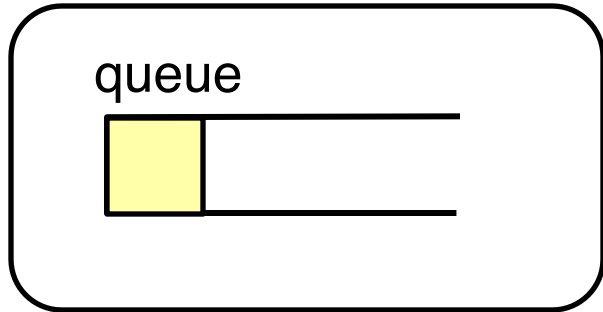
T2 (Running)

```
AddToQueue(item) {  
  lock.Acquire();  
  queue.enqueue(item);  
  dataready.signal();  
  lock.Release();  
}
```

signal() wakes up T1 and moves it on ready queue

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T2)  
dataready → NULL  
queue

CPU State

Running: T2  
Ready  
queue → T1, T3  
...

T1 (Ready)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

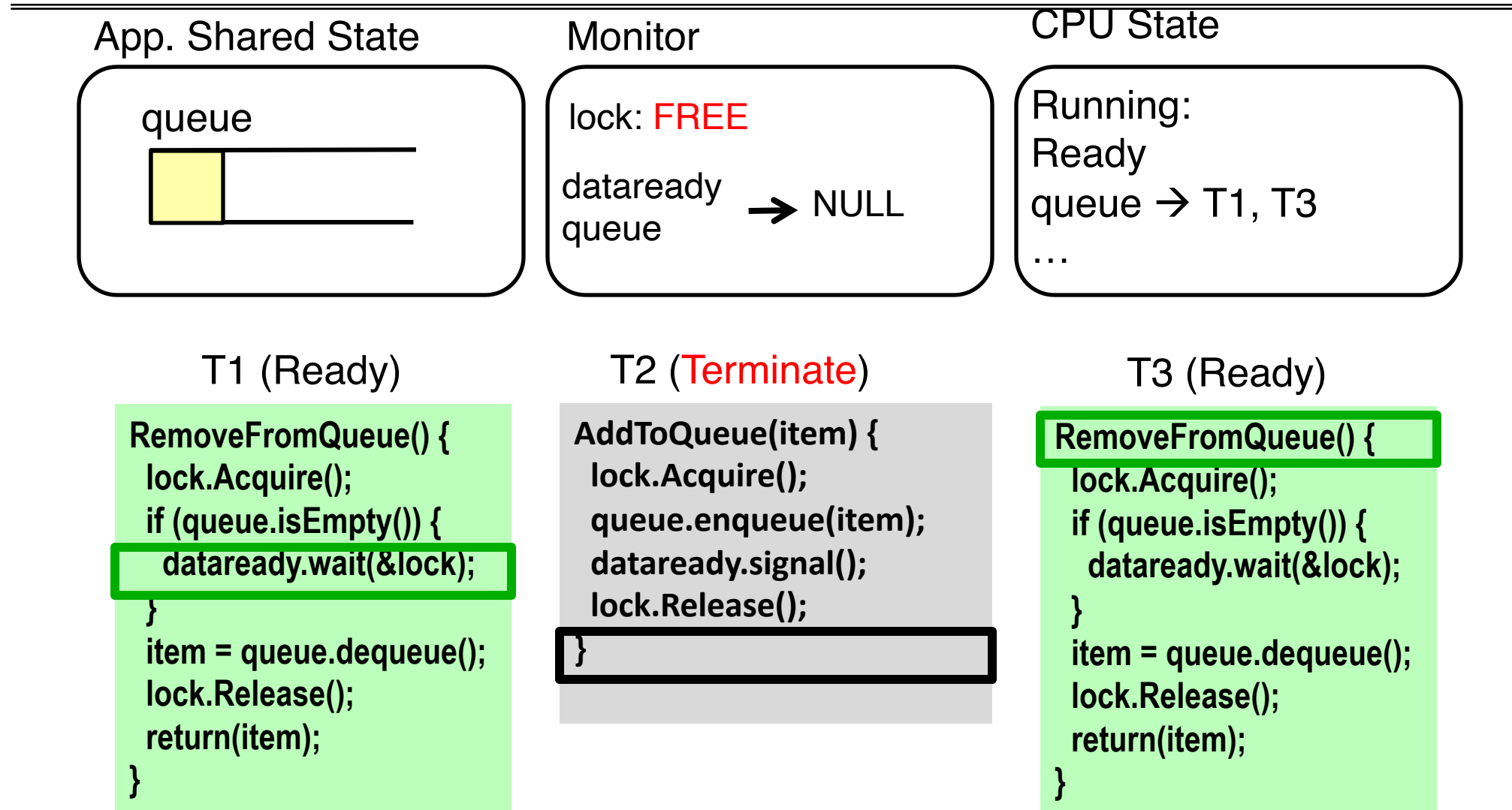
T2 (Running)

```
AddToQueue(item) {  
  lock.Acquire();  
  queue.enqueue(item);  
  dataready.signal();  
  lock.Release();  
}
```

T3 (Ready)

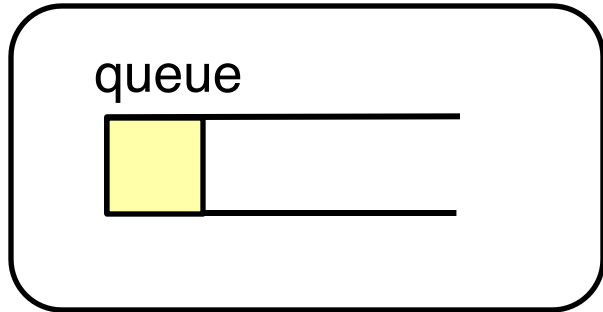
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

# Mesa Monitor: Why “while()”?



# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: FREE  
dataready → NULL  
queue

CPU State

Running: **T3**  
Ready  
queue → T1

T3 scheduled first!

T1 (Ready)

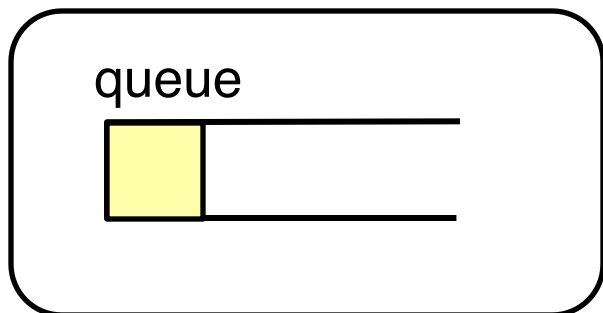
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

T3 (**Running**)

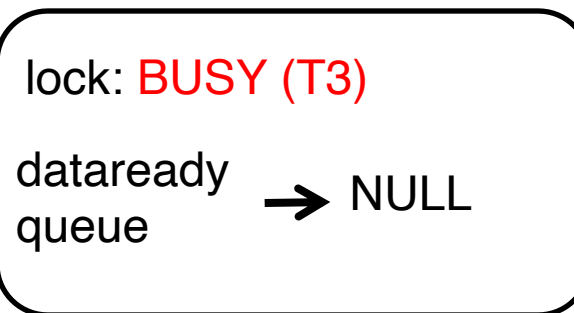
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

# Mesa Monitor: Why “while()”?

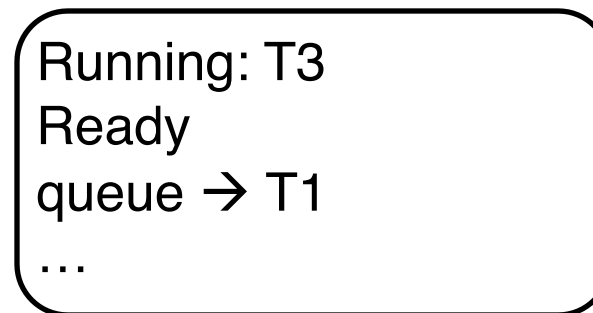
App. Shared State



Monitor



CPU State



T1 (Ready)

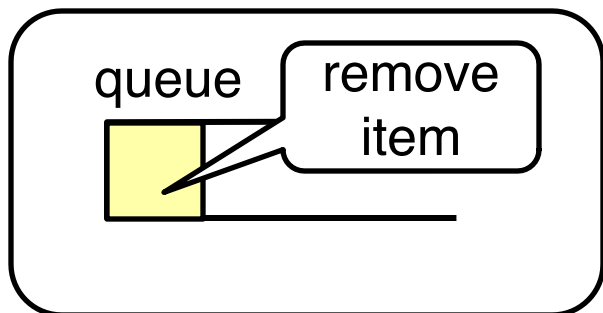
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

T3 (Running)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: BUSY (T3)  
dataready → NULL  
queue

CPU State

Running: T3  
Ready  
queue → T1  
...

T1 (Ready)

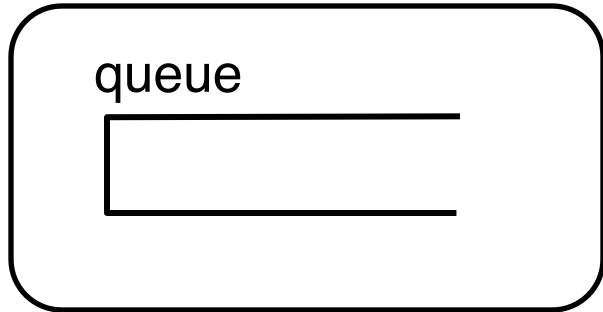
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

T3 (Running)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **FREE**  
dataready → NULL  
queue

CPU State

Running:  
Ready  
queue → T1  
...

T1 (Ready)

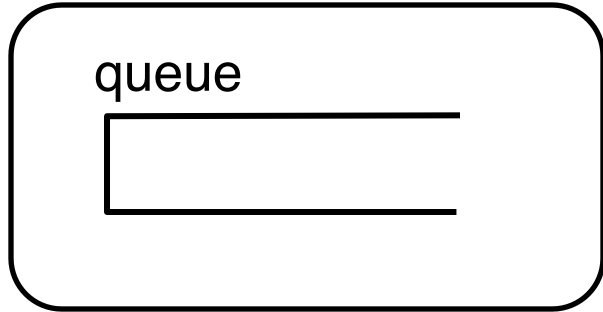
```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

T3 (**Finished**)

```
RemoveFromQueue() {  
  lock.Acquire();  
  if (queue.isEmpty()) {  
    dataready.wait(&lock);  
  }  
  item = queue.dequeue();  
  lock.Release();  
  return(item);  
}
```

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **BUSY (T1)**  
dataready → NULL  
queue → NULL

CPU State

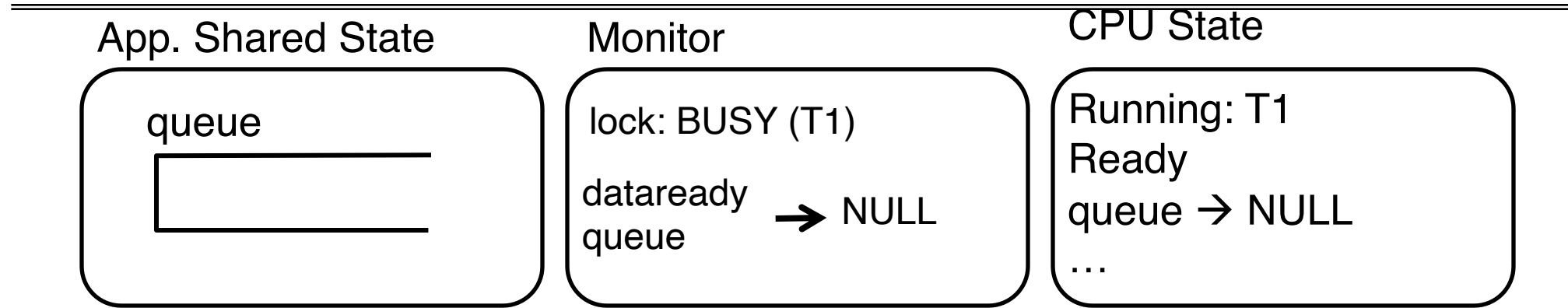
Running: **T1**  
Ready  
queue → NULL  
...

T1 (**Running**)

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```



# Mesa Monitor: Why “while()”?



T1 (**Running**)

```
RemoveFromQueue() {  
    lock.Acquire();  
    if (queue.isEmpty()) {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

ERROR:  
Nothing in the  
queue!

# Mesa Monitor: Why “while()”?

App. Shared State

queue



Monitor

lock: BUSY (T1)

dataready  
queue → NULL

CPU State

Running: T1

Ready

queue → NULL

...

T1 (Running)

```
RemoveFromQueue() {  
    lock.Acquire();  
    while (queue.isEmpty())  
    {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

Replace  
“if” with  
“while”

# Mesa Monitor: Why “while()”?

App. Shared State

queue



Monitor

lock: BUSY (T1)

dataready → NULL  
queue

CPU State

Running: T1

Ready

queue → NULL

...

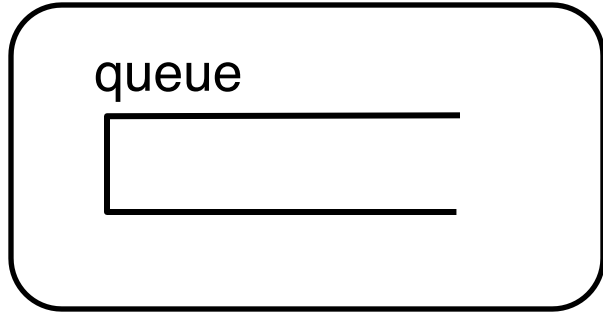
T1 (Ready)

```
RemoveFromQueue() {  
    lock.Acquire();  
    while (queue.isEmpty())  
    {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

Check  
again if  
empty!

# Mesa Monitor: Why “while()”?

App. Shared State



Monitor

lock: **FREE**  
dataready → **T1**  
queue

CPU State

Running: T1  
Ready  
queue → NULL  
...

T1 (**Waiting**)

```
RemoveFromQueue() {  
    lock.Acquire();  
    while (queue.isEmpty())  
    {  
        dataready.wait(&lock);  
    }  
    item = queue.dequeue();  
    lock.Release();  
    return(item);  
}
```

# Quick Questions

---

- Do lock.Acquire() and lock.Release() always trap into kernel?
- Interrupt handlers must use spinlocks instead of queueing locks. Why?
  - Note: interrupt handlers are not supposed to sleep

# Homework

---

- Search for how Java synchronization works.
  - Key words: “synchronized”, “wait”, “notify”, “notifyAll”.
  - Is it based on Hoare or Mesa model?
- Implement semaphores with test&set in pseudo code.