# Implementing Mutual Exclusion

Sarah Diesburg

Operating Systems

CS 3430

# From the Previous Lecture

- The “too much milk” example shows that writing concurrent programs directly with load and store instructions (i.e., C assignment statements) is tricky
- Programmers want to use higher-level operations, such as locks

# Ways of Implementing Locks

- All implementations require some level of hardware support

| | Locking primitives |
|---|---|
| High-level atomic operations | Locks, semaphores, monitors, send and receive |
| Low-level atomic operations | Load/store, interrupt disables, **`test_and_set`** |

# Atomic Memory Load and Store

- C assignment statements
- Examples:  “too much milk” solutions

# Disable Interrupts (for Uniprocessors)

- On a uniprocessor,
  - An operation is atomic as long as a context switch does not occur in the middle of an operation
- Solution 1

```
Lock::Acquire() {
    // disable interrupts;
}
Lock::Release() {
    // enable interrupts;
}
```

# Problems with Solution 1

- A user-level program may not re-enable interrupts
  - The kernel can no longer regain the control
- No guarantees on the duration of interrupts; bad for real-time systems
- Solution 1 will not work for more complex scenarios (nested locks)

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

The lock is initially FREE.

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

Check the lock value while interrupts are disabled.

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

Re-enable interrupts inside the loop, so someone may have a chance to unlock.

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

Disable the interrupts again before checking the lock.

# Solution 2

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

Lock::Acquire() {
   // disable interrupts
   while (value != FREE) {
        // enable interrupts
        // disable interrupts
   }
   value = BUSY;
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   value = FREE;
   // enable interrupts
}
```

If no one is holding the lock, grab the lock.

# Problems with Solution 2

- It works for a single processor
- It does not work on a multi-processor machine
    - Other CPUs can still enter the critical section

# The `test_and_set` Operation

- ***test_and_set*** works on multiprocessors
  - Atomically reads a memory location
  - Sets it to 1
  - Returns the old value of memory location

# The test_and_set Operation

```
value = 0;

Lock::Acquire() {
  // while the previous value is BUSY, loop
  while (test_and_set(value) == 1);
}

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

# Common Problems with Mentioned Approaches

- ***Busy-waiting***: consumption of CPU cycles while a thread is waiting for a lock
  - Very inefficient
  - Can be avoided with a waiting queue

May as well
sleep instead
of busy-wait…

# A tail of two threads…

- Suppose both threads want the lock, but like to be lazy…

Thread 1: Lazy

Thread 2: Lazier

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
    // disable interrupts
    if (value != FREE) {
        // Queue the thread
        // Go to sleep
    } else {
        value = BUSY;
    }
    // enable interrupts
}
```

```
Lock::Release() {
    // disable interrupts
    if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
    } else {
        value = FREE;
    }
    // enable interrupts
}
```

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Thread 1 tries to grab the lock.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

No more busy waiting…

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Grab the lock.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Thread 1 goes on computing.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Thread 2 tries to grab the lock.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
    // disable interrupts
    if (value != FREE) {
        // Queue the thread
        // Go to sleep
    } else {
        value = BUSY;
    }
    // enable interrupts
}
```

```
Lock::Release() {
    // disable interrupts
    if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
    } else {
        value = FREE;
    }
    // enable interrupts
}
```

The lock is busy...

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Put the thread 2 on a waiting queue.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Sleep it off... Context switch; wait for someone to wake up the thread.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Say thread 1 wants to release the lock (interrupts are already disabled by thread 2).

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Hello?  Is someone waiting there?
Thread 2 is waiting.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Put thread 2 on ready queue; context switch.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Thread 2: Who woke me?
I don’t do mornings...

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Thread 2 is done with its computation.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Suppose no one else is waiting.

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Release the lock.  (Thread 1 has finished its work, so it’s okay.)

# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

Warp 9, engage (let’s get out of here)…

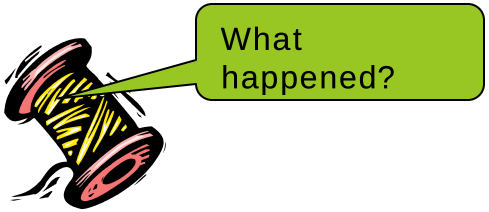
# Locks Using Interrupt Disables, Without Busy-Waiting

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

Lock::Acquire() {
   // disable interrupts
   if (value != FREE) {
        // Queue the thread
        // Go to sleep
   } else {
        value = BUSY;
   }
   // enable interrupts
}
```

```
Lock::Release() {
   // disable interrupts
   if (someone is waiting) {
        // wake a thread
        // Put it on ready queue
   } else {
        value = FREE;
   }
   // enable interrupts
}
```

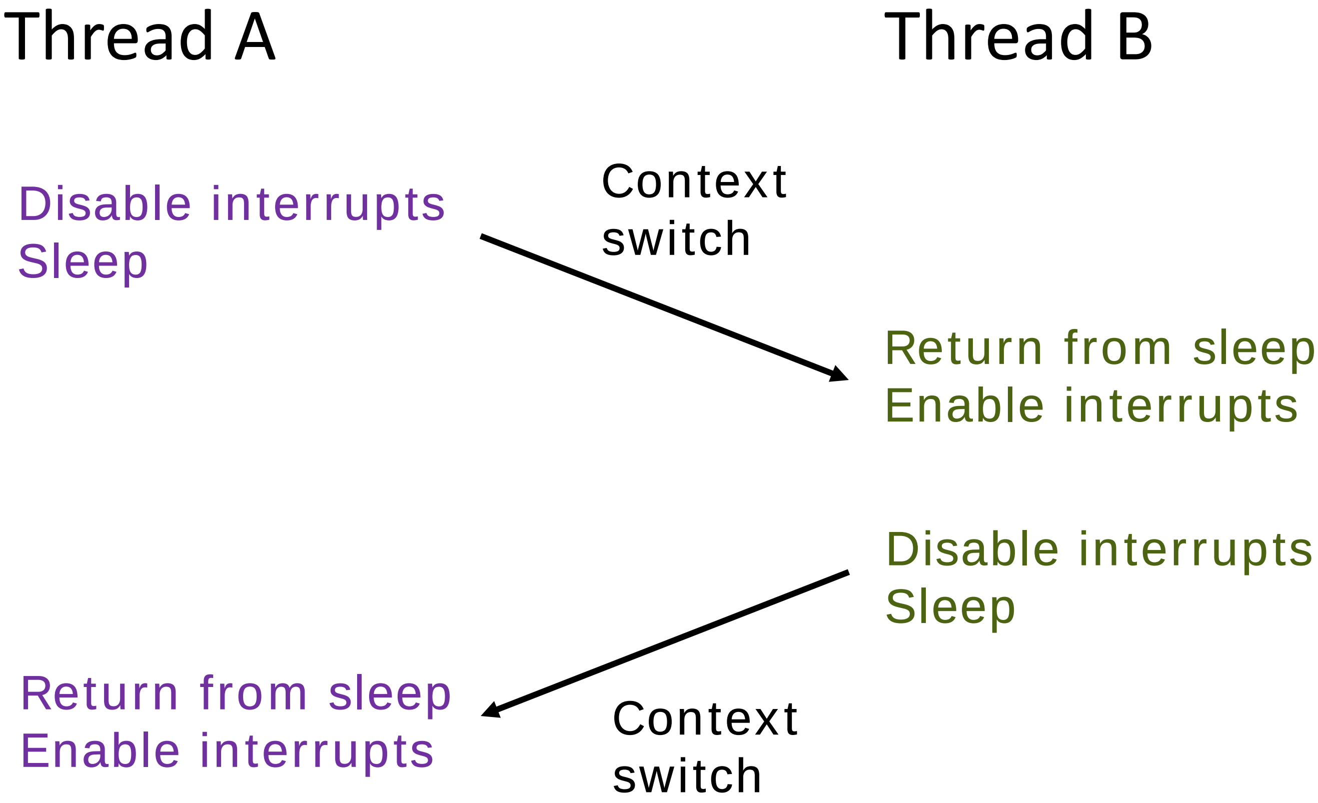
Eventually, the kernel will context switch back to thread 1.



# So, What’s Going On?

- Interrupt disable and enable operations occur across context switches (at the steady state)

# So, What’s Going On?



Thread A

Thread B

Disable interrupts
Sleep

Context
switch

Return from sleep
Enable interrupts

Disable interrupts
Sleep

Context
switch

Return from sleep
Enable interrupts

# Locks Using **`test_and_set`**, With Minimal Busy-Waiting

- Impossible to use **`test_and_set`** to avoid busy-waiting
- However, waiting can be minimized with a waiting queue

# Locks Using `test_and_set`, With Minimal Busy-Waiting

```
class Lock {
   int value = FREE;
   int guard = 0;
}

Lock::Acquire() {
   while (test_and_set(guard));
   if (value != FREE) {
        // queue the thread
        // guard = 0 and sleep
   } else {
        value = BUSY;
   }
   guard = 0;
}
```

```
Lock::Release() {
   while (test_and_set(guard));
   if (anyone waiting) {
        // wake up one thread
        // put it on ready queue
   } else {
        value = FREE;
   }
   guard = 0;
}
```