Locks

Edited slides from http://cs162.eecs.Berkeley.edu

Too Much Milk: Solution #4

- 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
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between **`Acquire()`** and **`Release()`** called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream ;-)

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

Goals for Today

- Explore several implementations of locks
- Continue with Synchronization Abstractions
  - Semaphores, Monitors, and Condition variables
- Very Quick Introduction to scheduling

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 (although virtual memory tricky)
  - 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!

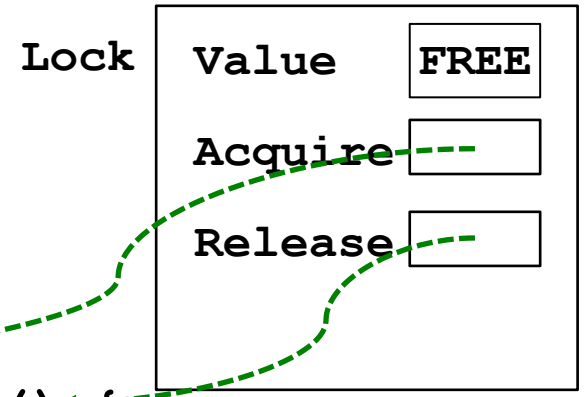
- Critical Sections might be arbitrarily long

What happens with I/O or other important events?

- "Reactor about to meltdown. Help?"

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;
   while(value == BUSY) {
      enable interrupts;
         // allow interrupts
      disable interrupts;
   }
   value = BUSY;
   enable interrupts;
}
```

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

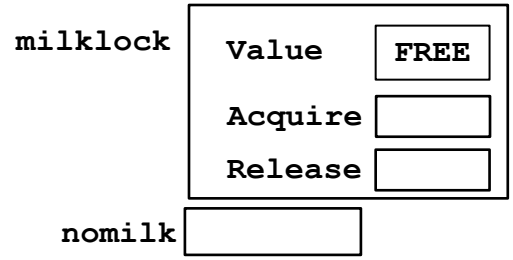
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

main

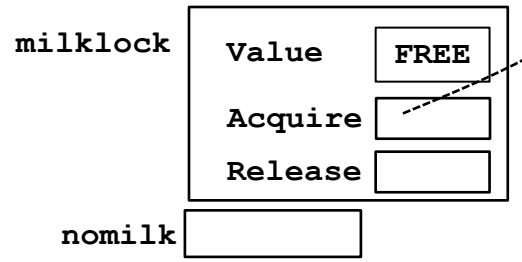
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



Acquire
main

main

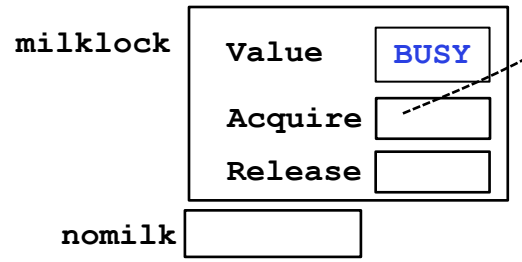
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



Acquire
main

main

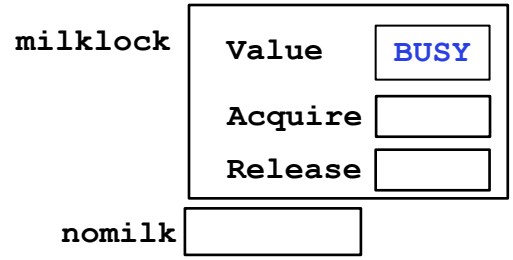
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

main

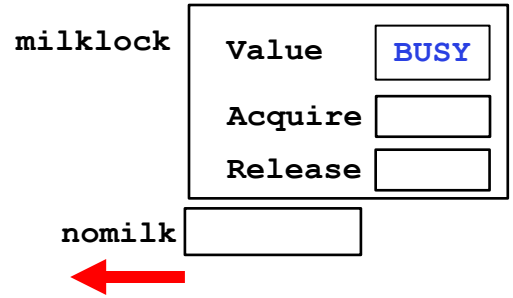
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

main

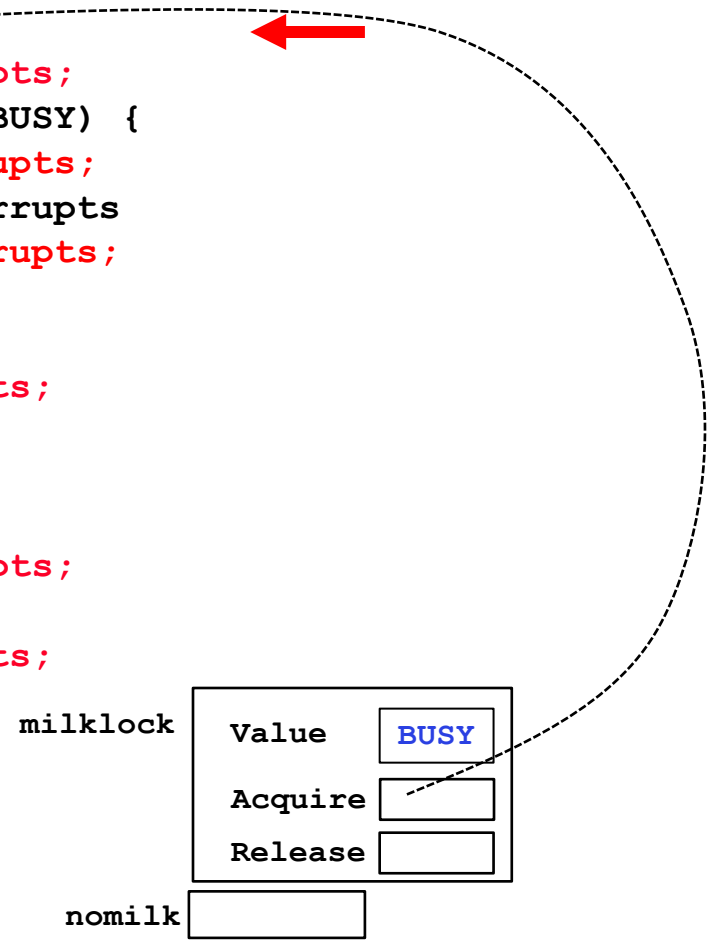
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

Acquire

main

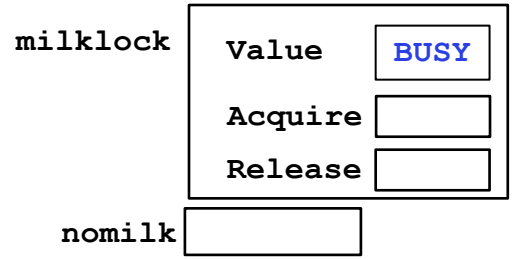
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

Acquire

main

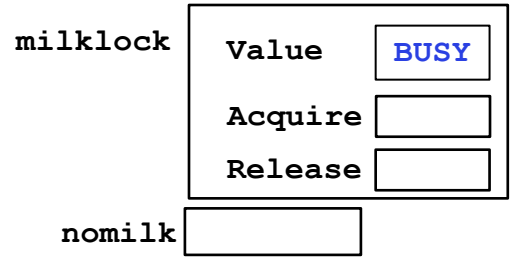
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

Acquire

main

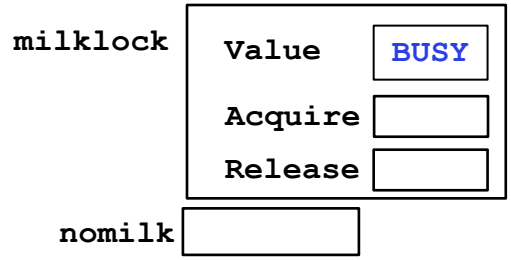
```
ClassLock() {
int value = FREE;

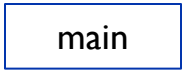
Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

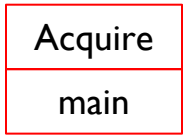
Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```







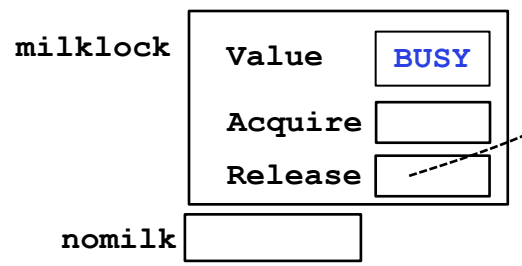
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



Release
main

Acquire
main

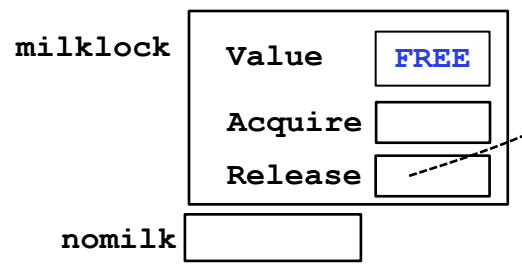
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



Release
main

Acquire
main

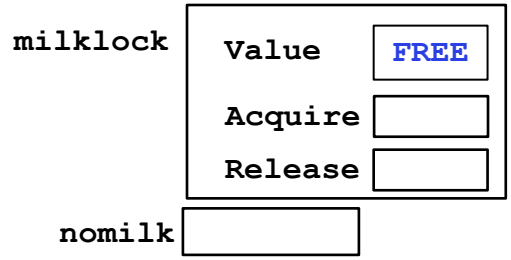
```
ClassLock() {
int value = FREE;

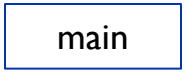
Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

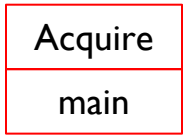
Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```







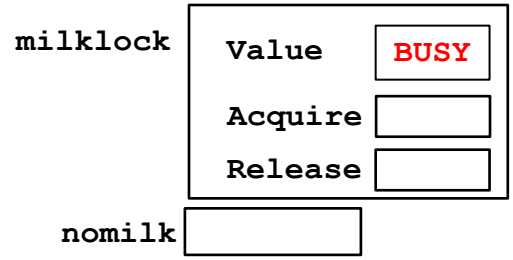
```
ClassLock() {
int value = FREE;

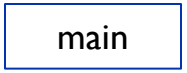
Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

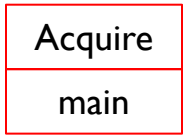
Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```







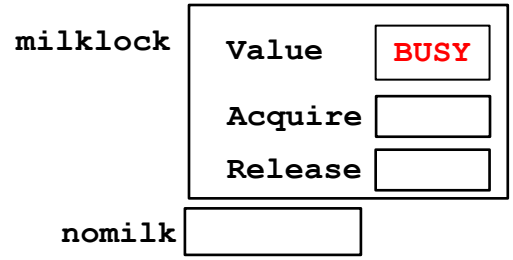
```
ClassLock() {
int value = FREE;

Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



main

main

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

```
ClassLock() {
int value = FREE;

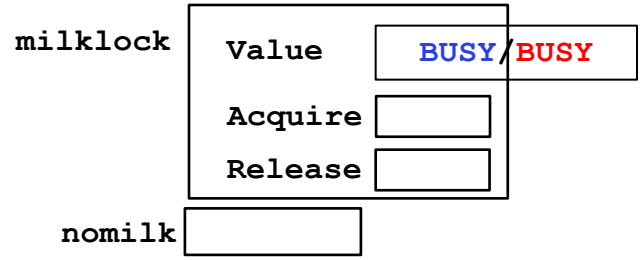
Acquire() {
  disable interrupts;
  while(value == BUSY) {
    enable interrupts;
    // allow interrupts
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}

Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

On Multiprocessor



Acquire
main

Acquire
main

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]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```
  ```

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

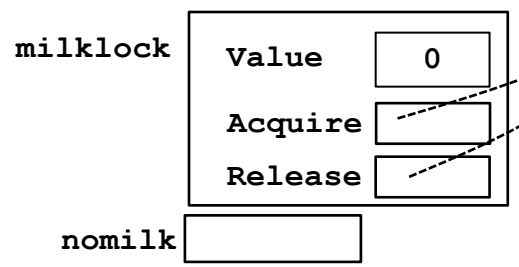
```
ClassLock() {
int value = 0; // Free

Acquire() {
  while (test&set(value)); // while busy
}

Release() {
  value = 0;
}
}

Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```



On Multiprocessor

```
ClassLock() {
int value = 0; // Free

Acquire() {
  while (test&set(value));
          // while busy
}

Release() {
  value = 0;
}
}

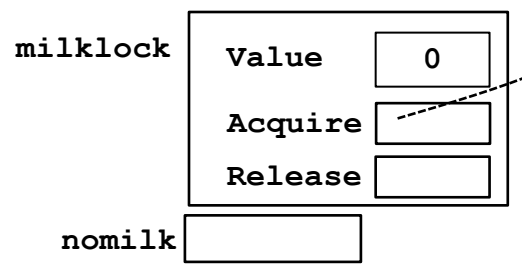
Lock milklock;

milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

test&set(value)

test&set(value)

Test&set이 atomic 하므로 반드시 둘 중에 하나가 먼저 실행된 후에 다른 하나가 나중에 실행된다.



Acquire
main

Acquire
main

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!
- For semaphores and 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

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;
}
```

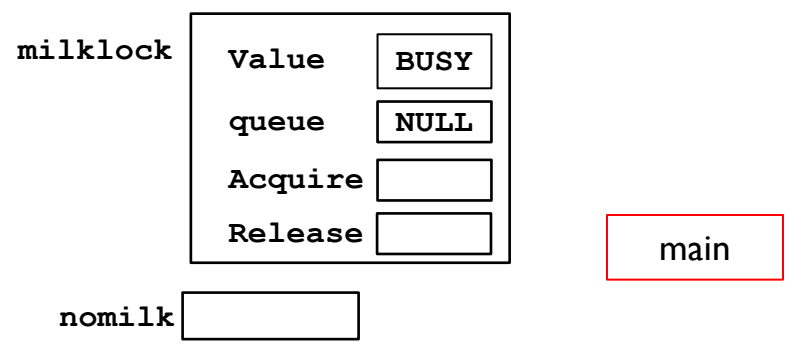
```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

```
ClassLock() {
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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

main



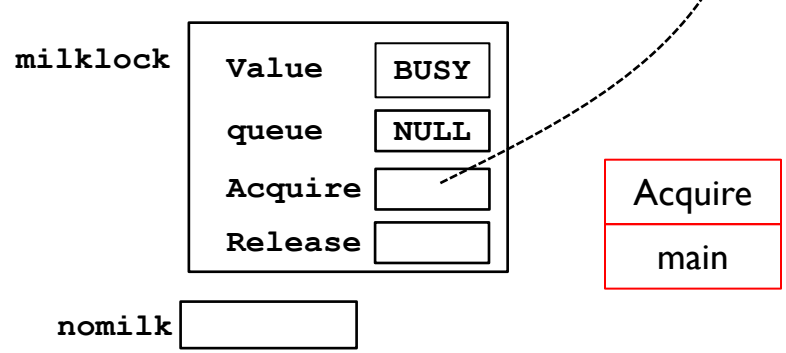
main

```
ClassLock() {
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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

main

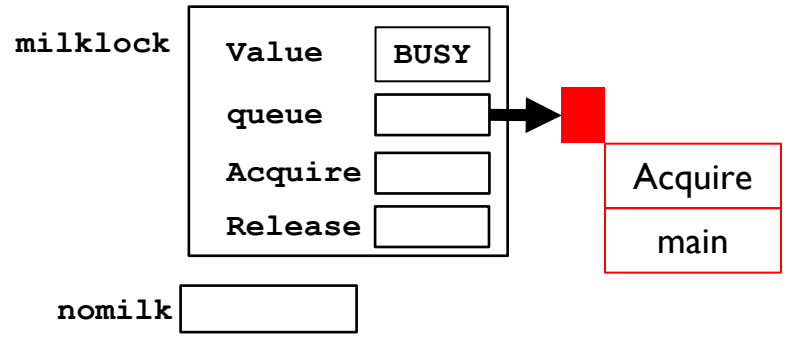


```
ClassLock() {
int value = FREE;

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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

main

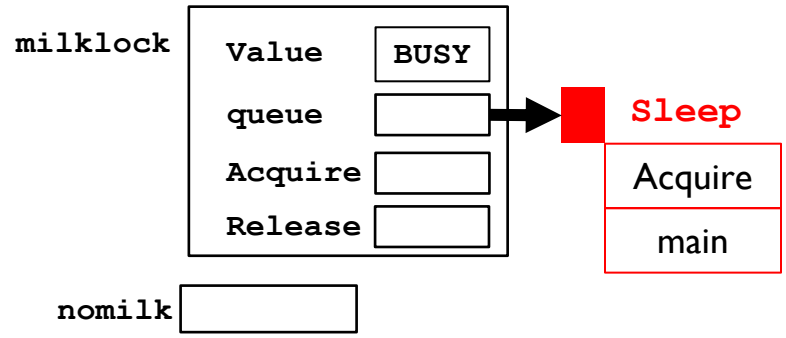


```
ClassLock() {
int value = FREE;

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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

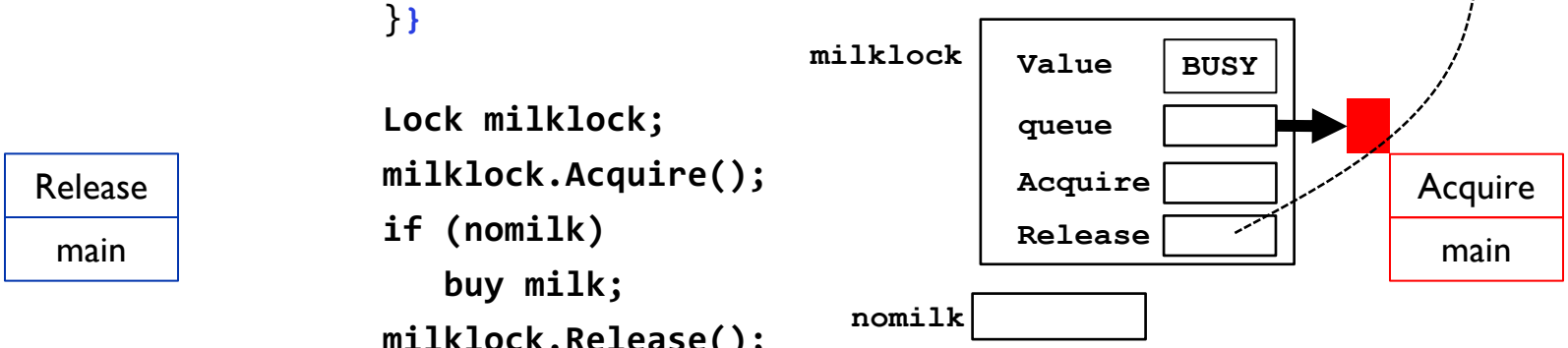
main



```
ClassLock() {
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;
}}
```

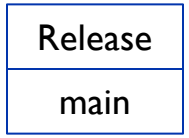
```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

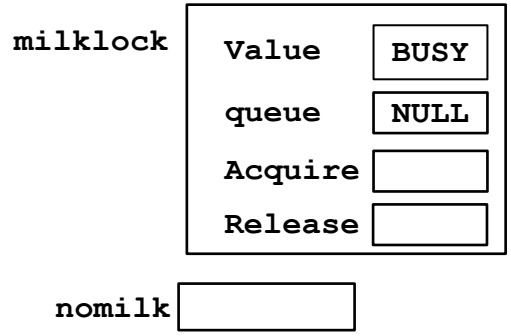


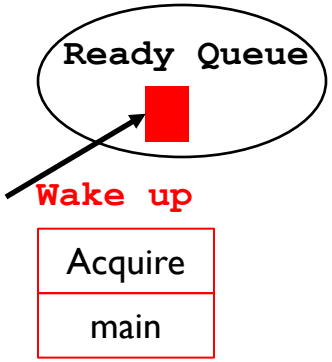
```
ClassLock() {
int value = FREE;

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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```





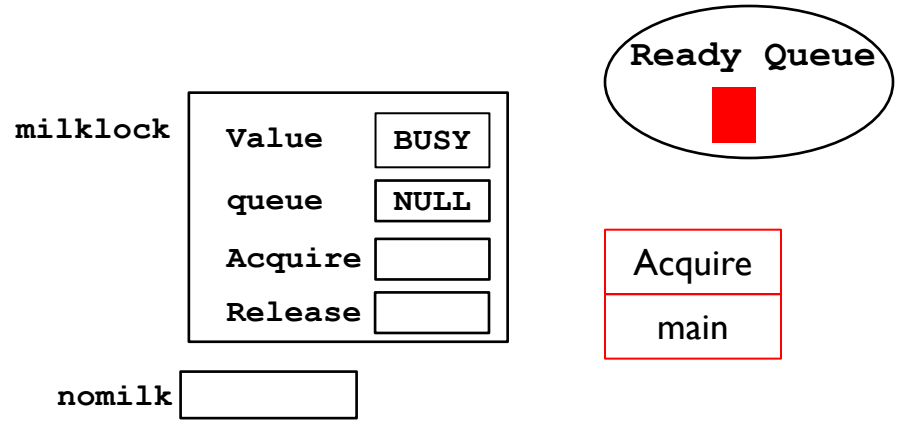


```
ClassLock() {
int value = FREE;

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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

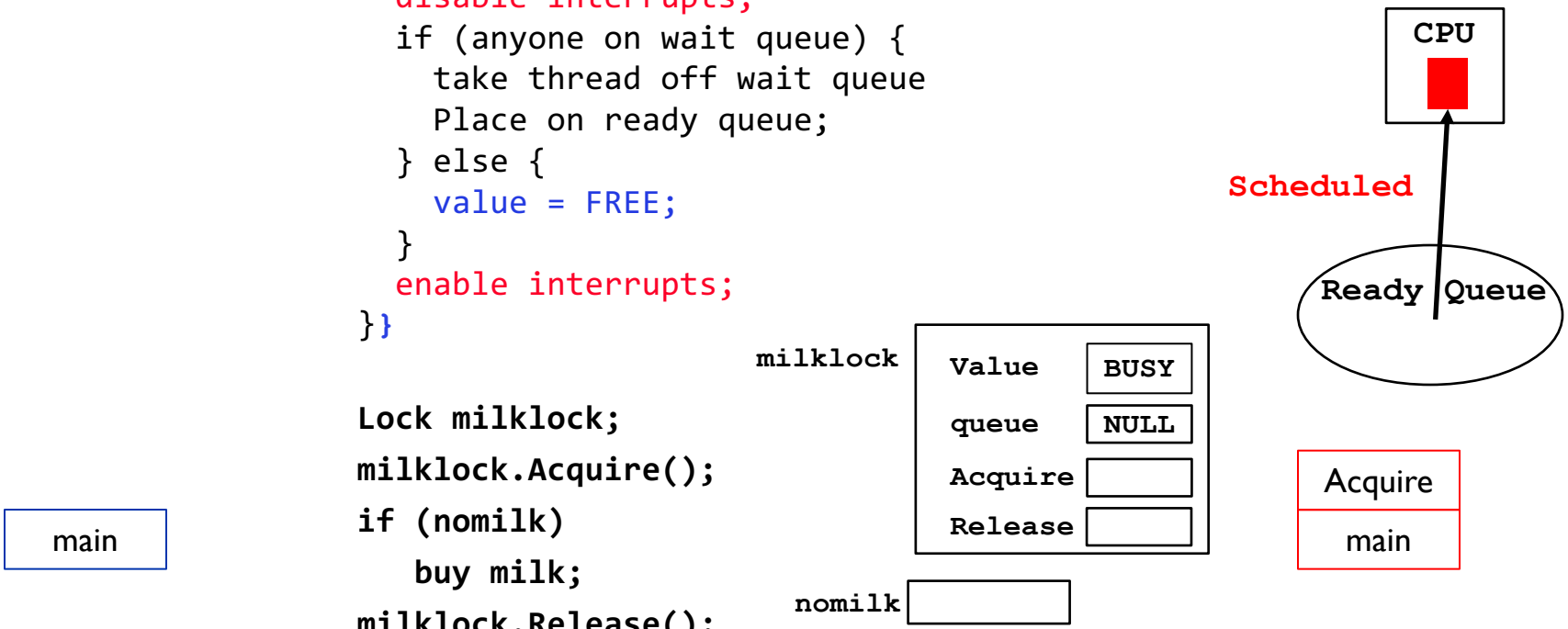
main



```
ClassLock() {
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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

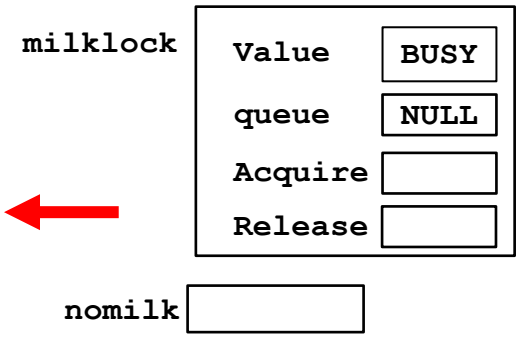


```
ClassLock() {
int value = FREE;

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;
}}
```

```
Lock milklock;
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

main



main

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!

Interrupt Re-enable in Going to Sleep

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

Enable Position
Enable Position
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
  - 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?

Interrupt Re-enable in Going to Sleep

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

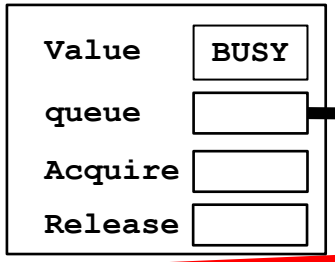
Value	BUSY
queue	NULL
Acquire	
Release	

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

Value	BUSY
queue	
Acquire	
Release	

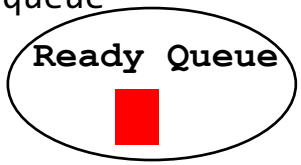
- Before Putting thread on the wait queue?
  - Release can check the queue and not wake up thread

Interrupt Re-enable in Going to Sleep

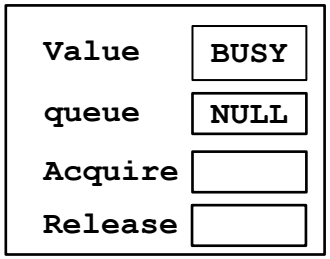



```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        enable interrupts;
```

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




```
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```



- 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

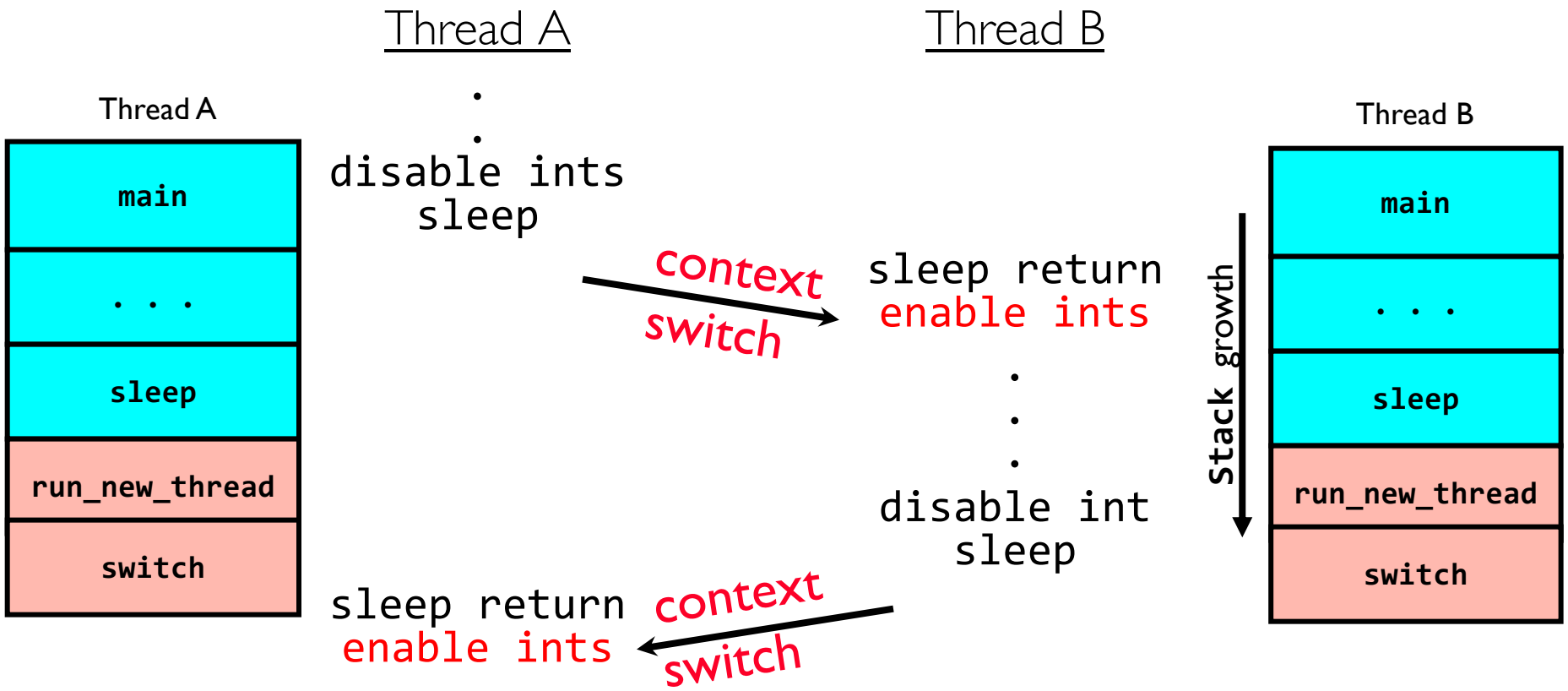
- Want to put it after `sleep()`. But – how?

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

    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

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




```
run_new_thread() {
  newThread = PickNewThread();
  switch(curThread, newThread);
  ThreadHouseKeeping(); /* Do any cleanup */
}
```

Saving/Restoring state (often called "Context Switch)

```
Switch(tCur,tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
            …
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
            …
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
```

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?

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 replace

- `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() //enable int
  } 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 & wait

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

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

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

```
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;
}
```

Threads waiting to enter critical section busy-wait

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, 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