ECE 570/670

David Irwin

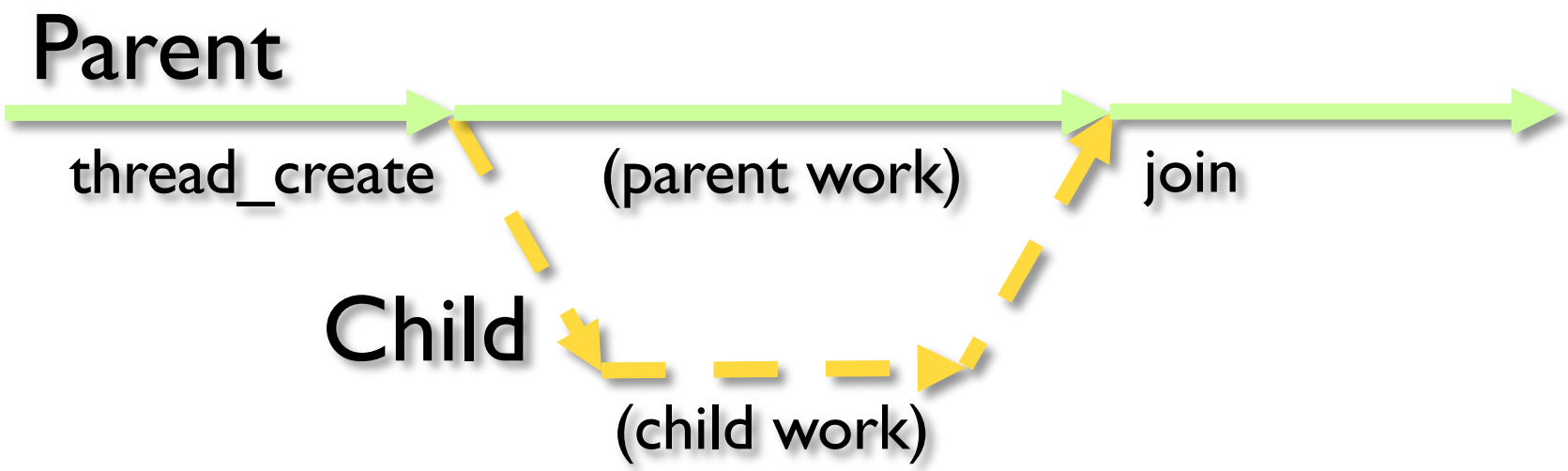
Lecture 9

Administrative Details

- 670 project meetings tomorrow
  - In my office: Knowles Eng. Bldg. 215C
- Should be working on Assignment 1d
  - Assignment 1 disk scheduler (1d)
    - How to use concurrency primitives
  - Assignment 1 thread library (1t)
    - How concurrency primitives actually work
    - Due 3/24
  - **Midterm in two weeks (3/29)**
    - Review in class on 3/28

Thread join

- How can the parent wait for child to finish?



Thread join

- Will this work?
- Sometimes, assuming
  - Uni-processor
  - No pre-emptions
- Never, ever assume these things!
- Yield is like slowing the CPU
  - Program must work +- any yields

```
child () {
  print "child works"
}
```

```
parent () {
  create child thread
  print "parent works"
  yield ()
  print "parent continues"
}
```

```
parent works
child works
parent continues
```

```
child works
parent works
parent continues
```

Thread join

- Will this work?

```
parent () {
  create child thread
  lock
  print "parent works"
  wait
  print "parent continues"
  unlock
}
```

```
child () {
  lock
  print "child works"
  signal
  unlock
}
```


```
parent works
child works
parent continues
```

```
child works
parent works
parent continues
```

- No. Child can call signal first.
- Would this work with semaphores?
  - Yes
  - No missed signals (increment sem value)

How can we solve this?

```
parent () {
  childDone = 0
  create child thread
  print “parent works”
  lock
  while (!childDone) {
    wait
  }
  print “parent continues”
  unlock
}
```

```
child () {
  print “child works”
  lock
  childDone = 1
  signal
  unlock
}
```

```
parent works
child works
parent continues
```

```
child works
parent works
parent continues
```

Thread join

- Many libraries supply “join” operation
  - Makes this a lot easier

```
parent () {
   create child thread
   print “parent works”
   join with child
   print “parent continues”
}
```

```
child () {
   print “child works”
}
```

Switching threads

- What needs to happen to switch threads?

1. **Thread returns control to OS**
   - For example, via the "yield" call
2. **OS chooses next thread to run**
3. **OS saves state of current thread**
   - *To* its thread control block
4. **OS loads context of next thread**
   - *From* its thread control block
5. **Run the next thread**

Assignment 1 Part 2

1. Thread returns control to OS

- How does the thread return control back to OS?
  - Voluntary **internal events**
  - Thread might block inside lock or wait
  - Thread might call into kernel for service
    - (system call)
  - Thread might call yield
- Are internal events enough?

1. Thread returns control to OS

- Involuntary **external events**
  - (events not initiated by the thread)
  - Hardware interrupts
    - Transfer control directly to OS interrupt handlers
      - CPU checks for interrupts while executing
      - Jumps to OS code with interrupt mask set
  - Interrupts lead to pre-emption (a forced yield)
- Common interrupt: timer interrupt

2. Choosing the next thread

- If no ready threads, just spin
  - Modern CPUs: execute a "halt" instruction
- Loop switches to thread if one is ready
- Many ways to prioritize ready threads
  - Involves "scheduling" the CPU
  - Will discuss next week

3. Saving state of current thread

- What needs to be saved?
  - Registers, PC, SP
- What makes this tricky?
  - Self-referential sequence of actions
  - Need registers to save state
  - But you're trying to save all the registers
- Saving the PC is particularly tricky

Saving the PC

- Why won't this work?

Instruction address

```
100 store PC in TCB
101 switch to next thread
```

- Returning thread will execute instruction at 100
  - And just re-execute the switch
  - Really want to save address 102

4. OS loads the next thread

- Where is the thread's state/context?
  - Thread control block (in memory)
- How to load the registers?
  - Use load instructions to grab from memory
- How to load the stack?
  - Stack is already in memory, load SP

5. OS runs the next thread

- How to resume thread's execution?
  - Jump to the saved PC
- On whose stack are these steps running? (or who jumps to the saved PC?)
  - The thread that called yield
  - (or was interrupted or called lock/wait)
- How does this thread run again?
  - Some other thread must switch to it

Example thread switching

```
Thread 1
1    print “start thread 1”
2    yield ()
3    print “end thread 1”

Thread 2
4    print “start thread 2”
5    yield ()
6    print end thread 2”

yield ()
7    print “start yield (thread %d)”
8    swapcontext (tcb1, tcb2)
9    print “end yield (thread %d)”

swapcontext (tcb1, tcb2)
10   save regs to tcb1
11   load regs from tcb2
   // sp points to tcb2’s stack now!
12   jump tcb2.pc
   // sp must point to tcb1’s stack!
13   return
```

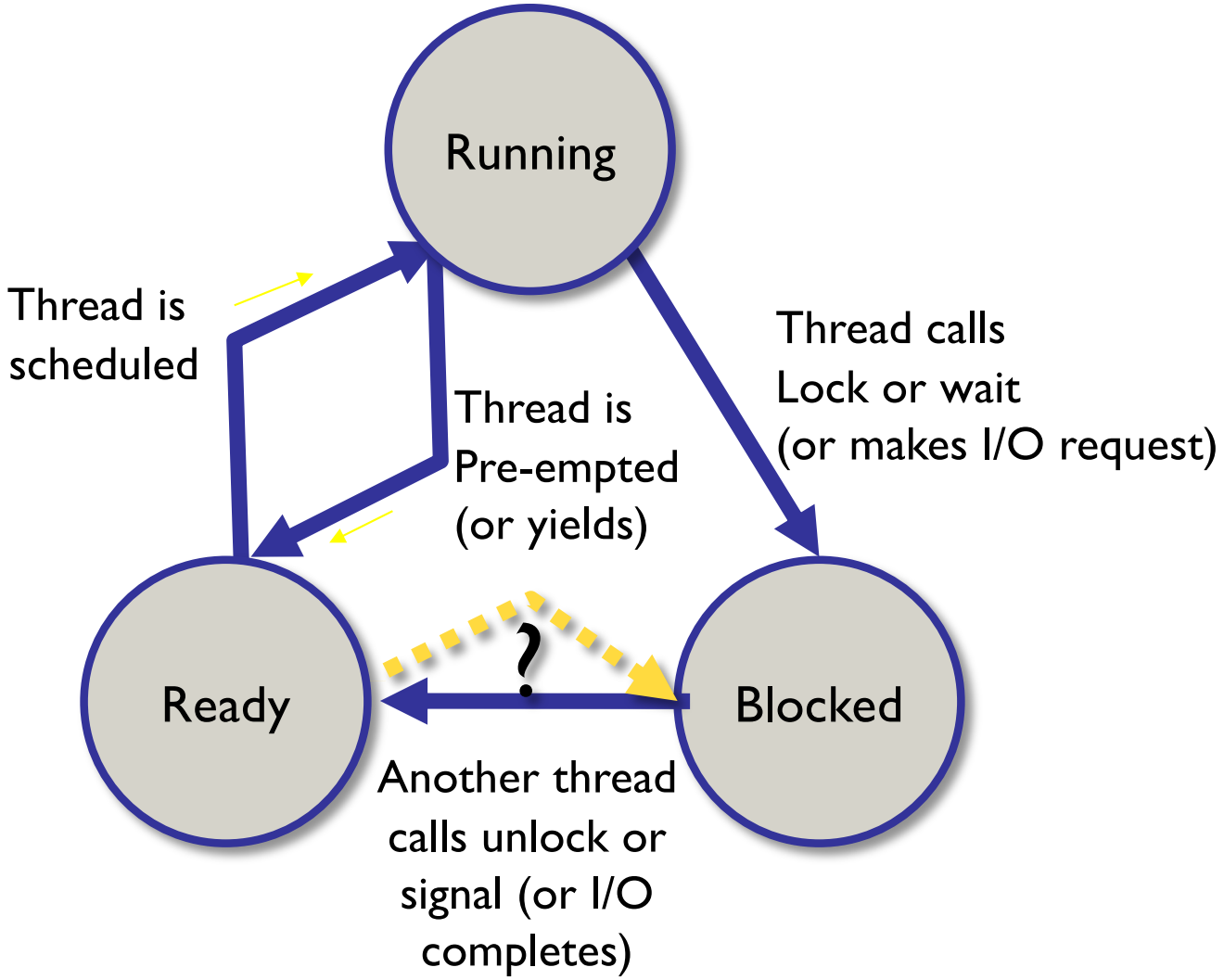
```
Thread 1 output          Thread 2 output
-----------------------------------------
start thread 1
start yield (thread 1)
                         start thread 2
                         start yield (thread 2)
end yield (thread 1)
end thread 1
                         end yield (thread 2)
                         end thread 2
```

Note: this assumes no pre-emptions. Programmers must assume pre-emptions, so this output is not guaranteed.

Thread states

- Running
  - Currently using the CPU
- Ready
  - Ready to run, but waiting for the CPU
- Blocked
  - Stuck in lock (), wait () or down ()

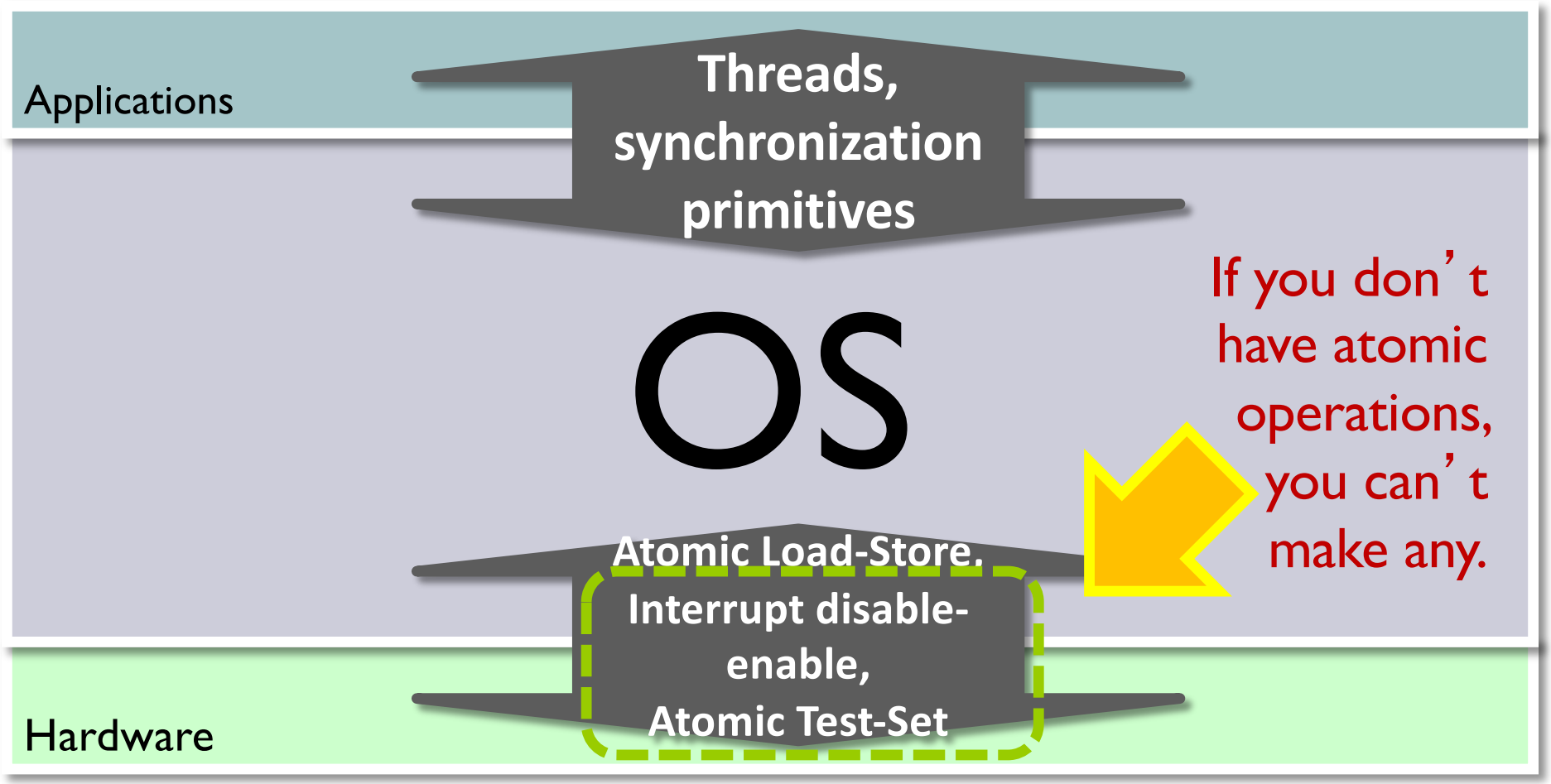
Thread states



Moving on…

- Discuss how locks are implemented

Recap and looking ahead



Using interrupt disable-enable

- Disable-enable on a uni-processor
  - Assume atomic (can use atomic load/store)
- How do threads get switched out (2 ways)?
  - Internal events (yield, I/O request)
  - External events (interrupts, e.g. timers)
- Easy to prevent internal events
- Use disable-enable to prevent external events

Why use locks?

- If we have disable-enable, why do we need locks?
  - Program could bracket critical sections with disable-enable
  - Might not be able to give control back to thread library

```
disable interrupts
while (1){}
```

  - Can't have multiple locks (over-constrains concurrency)
- Assignment 1: only disable interrupts in thread library

Lock implementation #1

- Disable interrupts, busy-waiting

```
lock () {
  disable interrupts
  while (value != FREE) {
    enable interrupts
    disable interrupts
  }
  value = BUSY
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  enable interrupts
}
```

Why is it ok for lock code to disable interrupts?

It's in the trusted kernel (we have to trust something).

Lock implementation #1

- Disable interrupts, busy-waiting

```
lock () {
  disable interrupts
  while (value != FREE) {
    enable interrupts
    disable interrupts
  }
  value = BUSY
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  enable interrupts
}
```

Do we need to disable interrupts in unlock?

Only if "value = FREE" is multiple instructions (safer)

Lock implementation #1

- Disable interrupts, busy-waiting

```
lock () {
  disable interrupts
  while (value != FREE) {
    enable interrupts
    disable interrupts
  }
  value = BUSY
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  enable interrupts
}
```

Why enable-disable in lock loop body?

Otherwise, no one else will run (including unlockers)

Using read-modify-write instructions

- Disabling interrupts
  - Ok for uni-processor, breaks on multi-processor
- Could use atomic load-store to make a lock
  - Inefficient, lots of busy-waiting
- *Need hardware support!*

Using read-modify-write instructions

- Most modern processor architectures
  - Provide an atomic read-modify-write instruction
- Atomically
  - Read value from memory into register
  - Write new value to memory
- Implementation details
  - Lock memory location at the memory controller

Too much milk, Solution 2

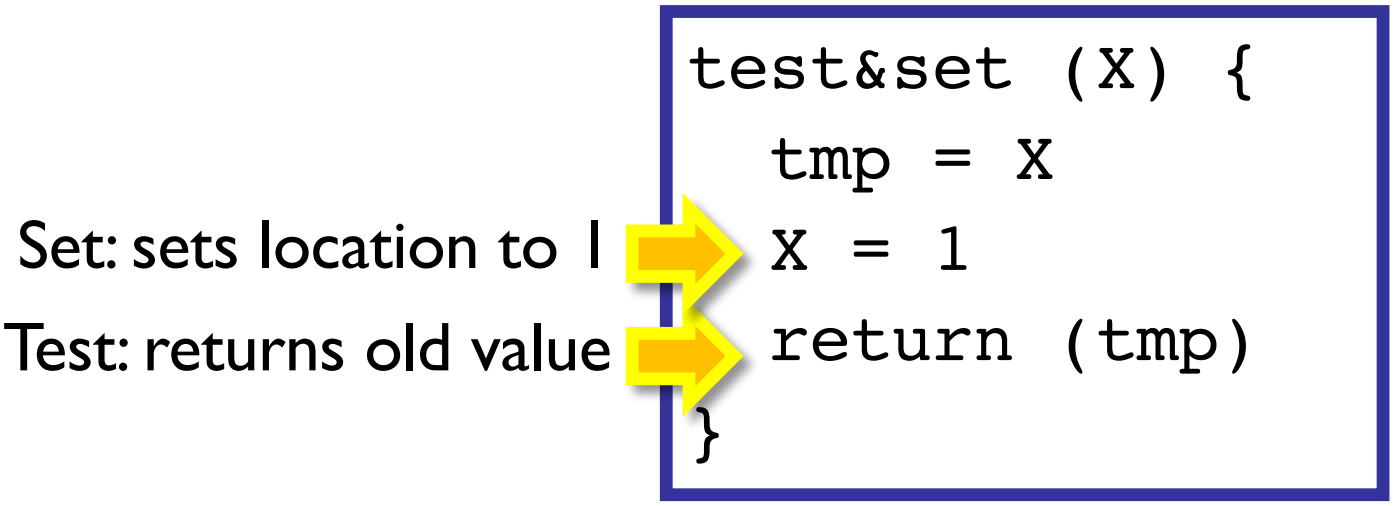
```
if (noMilk) {
   if (noNote){
      leave note;
      buy milk;
      remove note;
   }
}
```

Block is not atomic.
Must atomically

- check if lock is free
- grab it

What key part of lock code had to be atomic?

Test&set on most architectures



- Slightly different on Intel x86 (Exchange)
  - Atomically swaps value between register and memory

Lock implementation #2

- Use test&set
- Initially, value = 0

```
lock () {
  while (test&set(value) == 1) {
  }
}
```

```
unlock () {
  value = 0
}
```

What happens if value = 1?

What happens if value = 0?

Locks and busy-waiting

- All previous implementations have used busy-waiting
  - Wastes CPU cycles
  - Better for thread to sleep and let other threads run
- To reduce busy-waiting, integrate
  - Lock implementation
  - Thread dispatcher data structures

Lock implementation #3

- Interrupt disable, no busy-waiting

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Who gets the lock after someone calls unlock?

Thread calling unlock gives lock to thread on wait queue

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Who might get the lock if it weren't handed-off directly? (e.g. if value weren't set BUSY in unlock)

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

What kind of ordering of lock acquisition guarantees does the hand-off lock provide?

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

What does this mean?

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    lockqueue.push(&current_thread->ucontext);
    swapcontext(&current_thread->ucontext,
                &new_thread->ucontext));
}
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Adding a pointer to the TCB/context.

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Why a separate queue for the lock?

All locked threads are not ready to run; can't control ordering

Lock implementation #3

This is called a "hand-off" lock.

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Assignment 1 note: you must guarantee FIFO ordering of lock acquisition.

Lock implementation #3

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    enable interrupts
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

When and how could this fail?

Someone can unlock before thread added to wait queue

Lock implementation #3


```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    enable interrupts
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```


```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

When and how could this fail?

Lock implementation #3

```
lock () {
  disable interrupts
  if (value == FREE) {
    value = BUSY // lock acquire
  } else {
    add thread to queue of threads waiting for lock
    switch to next ready thread // don't add to ready queue
  }
  enable interrupts
}
```

```
unlock () {
  disable interrupts
  value = FREE
  if anyone on queue of threads waiting for lock {
    take waiting thread off queue, put on ready queue
    value = BUSY
  }
  enable interrupts
}
```

Putting lock thread on lock wait queue, switch must be atomic. Must call switch with interrupts off.