# Housekeeping (Lecture 13 - 10/9/2013)

- **Office hour canceled this Thursday**
  - sorry about the inconvenience
- **Kernel #1 due at 11:45pm on Friday, 10/25/2013**
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- **Any system issue, please get it resolved NOW**
  - come to office hours to get help
- **There is only one student in each section who does not have partners**
  - even if your team already has 4 students, you can add this student to your team!
- **Post questions about the kernel assignments to class Google Group**
  - extra credit for posting good responses

# 5.1 Threads Implementations

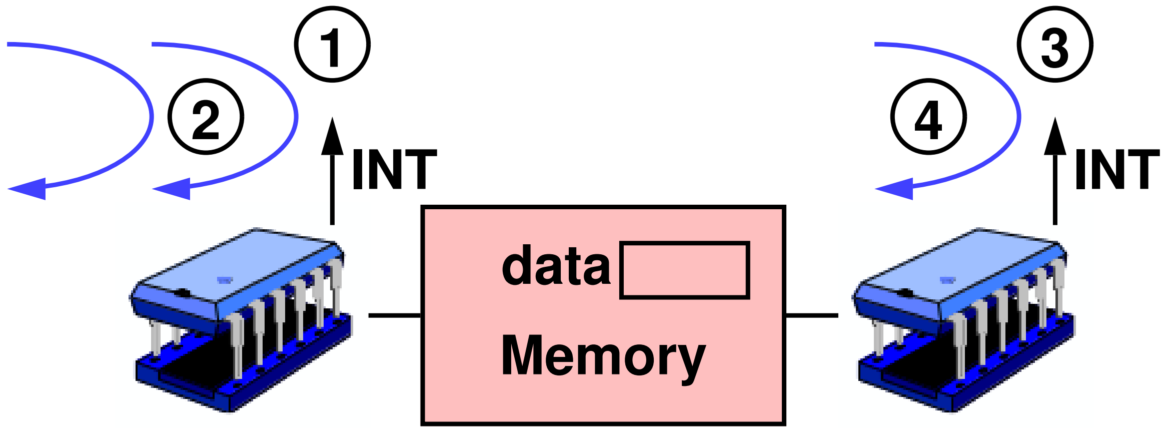
- Strategies
- *A Simple Thread Implementation*
- Multiple Processors

# Thread Synchronization

- How to implmement mutexes?
  - spin locks
  - sleep/blocking locks
  - futexes

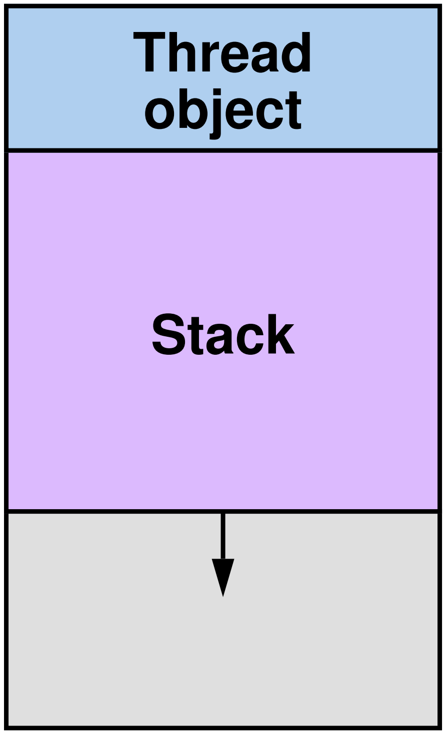
# A Simple Threads Implementation

- The challenge with implementing mutexes is that you have to ensure that they perform correctly under different kinds of concurrency

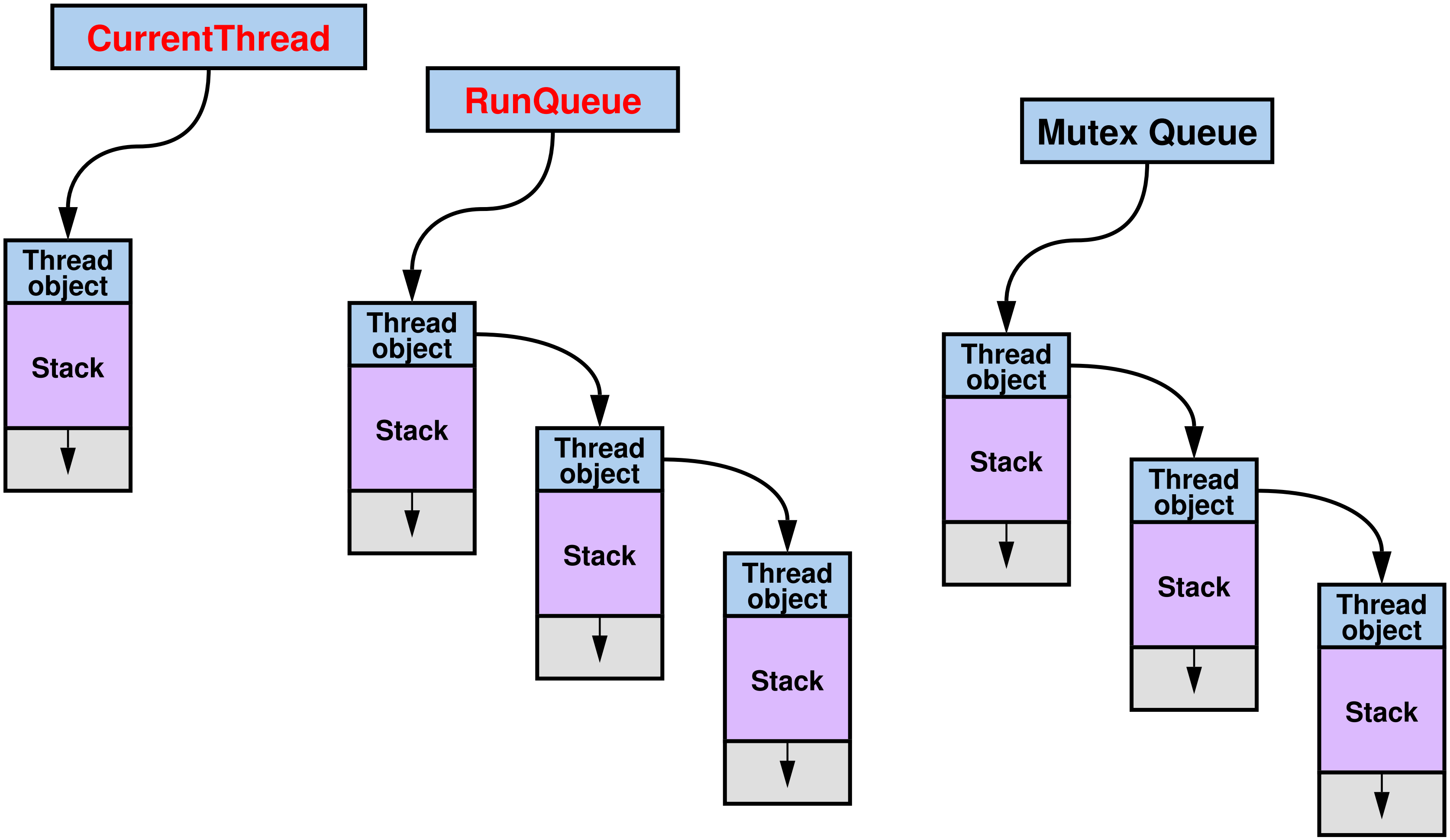



- Basis for user-level threads package
  - therefore, we are talking about kernel threads here
- ***Straight-threads*** implementation
  - everything happens in thread contexts
    - no interrupt
  - one processor

# Basic Representation



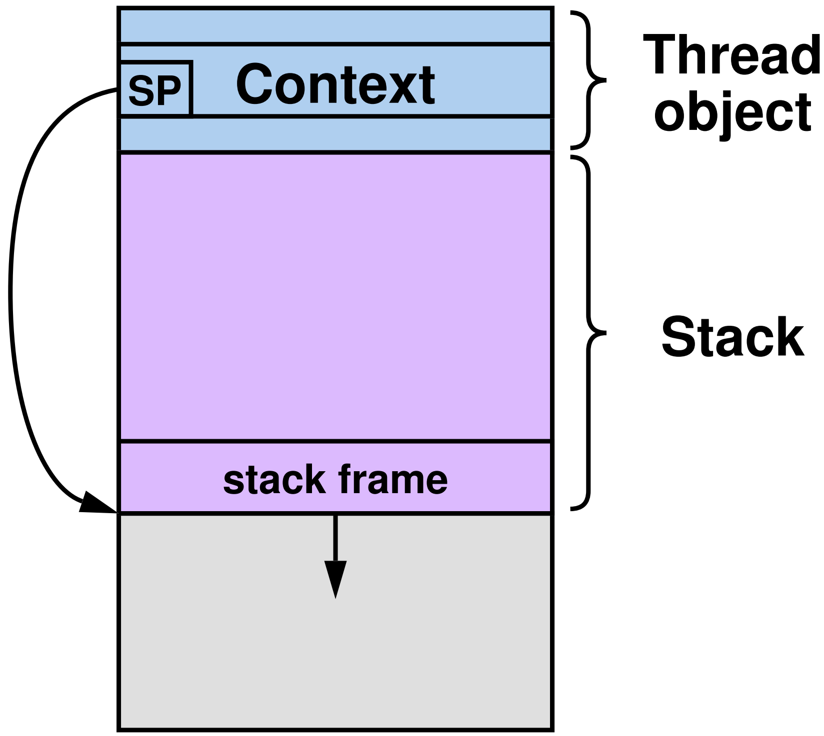
# A Collection of Threads




- Each thread must be in one of these data structures
  - your kernel assignment looks like this

# Context Pointer

- Recall from Ch 3




  - if this thread is not currently running, "stack frame" corresponds to `switch()`

# Straight-threads - Thread Switch

- Need a `thread_switch()` function to yield the processor

```
void thread_switch( ) {
  thread_t NextThread, OldCurrent;

  NextThread = dequeue(RunQueue);
  OldCurrent = CurrentThread;
  CurrentThread = NextThread;
  swapcontext(&OldCurrent->context,
              &NextThread->context);
  // We're now in the new thread's context
}
```

  - `switch()` in Ch 3 has a target thread argument
  - `swapcontext(old, new)` ***saves*** the caller's context into the `old` context and ***restores*** from the `new` context
  - note that the RunQueue may be empty, so this code is incomplete
  - before you get here, the current thread is queued onto somewhere else already (e.g., a mutex queue)

# Straight-threads - Synchronization

- According to the textbook

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

  - `mutex_unlock()` does not seem to work becuase when it returns, the mutex can be locked and the new mutex holder is not holding the mutex
  - after further analysis, it actually does work!

# Straight-threads - Synchronization

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

- **Why is the code atomic?**

# Straight-threads - Synchronization

```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    enqueue(m->queue, CurrentThread);
    thread_switch();
  } else
    m->locked = 1;
}

void mutex_unlock(mutex_t *m) {
  if (queue_empty(m->queue))
    m->locked = 0;
  else
    enqueue(runqueue, dequeue(m->queue));
}
```

- Why is the code atomic?
  - single process and no interrupts
  - no way to preempt a thread's execution
    - a thread holds on to the processor as long as it wants, until it reliquishes processor all by itself

# 5.1 Threads Implementations

- Strategies
- A Simple Thread Implementation
- *Multiple Processors*

# Straight-threads - Multiple Processors

- `thread_switch()` is no longer sufficient
  - it's meant for uniprocessor
- Simple approach
  - run on each *processor* an *idle thread*

```
void idle_thread() {
  while(1)
    thread_switch()
}
```

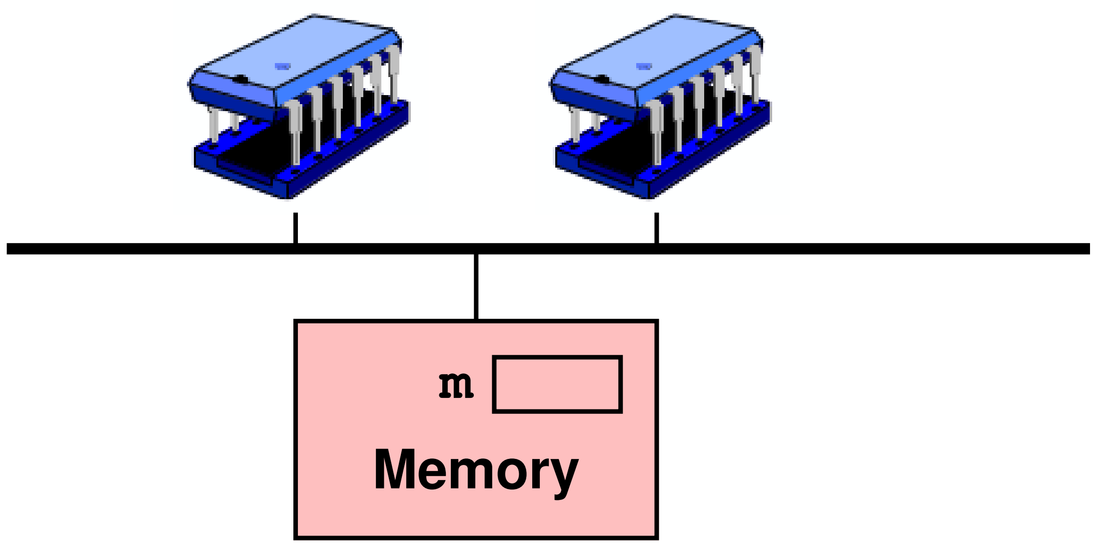
    - code is incomplete (may be because `thread_switch()` is incomplete, the way it was presented here )
    - this thread never blocks
    - make sure there is always something to run to avoid boundary condition
  - normal threads join the RunQueue when ready

# Straight-threads - Multiple Processors

- When there are multiple processors, the difficulty lies in locking

```
if (!m->locked) {
  m->locked = 1;
}
```

  - if both threads execute the above code concurrently, in different processors, both threads think they got the lock



# Hardware Support

- Compare and swap *instruction*

```
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;   // get the value of mutex
    if (*ptr == old) // if it equals to old
       *ptr = new    //   set it to new
    return tmp;      // return old if locked
}
```

  - often implemented as a machine-level instruction
    - must execute atomically

- Spin lock
  - mutex is represented as a bit, 0 if unlocked, 1 if locked

# Spin Lock

## Naive spin lock

```
void spin_lock(int *mutex) {
   while(CAS(mutex, 0, 1)) // textbook is wrong
      ;
}

void spin_unlock(int *mutex) {
   *mutex = 0;
}
```

## Better spin lock

```
void spin_lock(int *mutex) {
  while (1) {
    if (*mutex == 0) {
      // the mutex was at least momentarily unlocked
      if (!CAS(mutex, 0, 1))
        break;  // we have locked the mutex
      // some other thread beat us to it, try again
    }
  }
}
```

# Blocking Locks

- Spin locks are wasteful
  - processor time wasted waiting for the lock to be released
  - barely acceptable if locks are held only briefly
- A better approach is to have a blocking lock
  - threads wait by having their execution suspended
  - a thread much yield the processor and join a queue of waiting threads
    - later on, get resumed explicitly

# Blocking Locks

```
void blocking_lock(mutex_t *m) {
  if (m->holder != 0)
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else
    m->holder = CurrentThread;
}

void blocking_unlock(mutex_t *m) {
  if (queue_empty(m->wait_queue))
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
}
```

- This code only works on a uniprocessor

# Blocking Locks

1,2

```
void blocking_lock(mutex_t *m) {
  if (m->holder != 0)
    enqueue(m->wait_queue, CurrentThread);
    thread_switch();
  } else
    m->holder = CurrentThread;
}

void blocking_unlock(mutex_t *m) {
  if (queue_empty(m->wait_queue))
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
}
```

- On a multiprocessor, it may not work
  - threads 1 and 2 can both think they've got the lock

# Blocking Locks

```
1 → void blocking_lock(mutex_t *m) {
      if (m->holder != 0)
        enqueue(m->wait_queue, CurrentThread);
        thread_switch();
      } else
        m->holder = CurrentThread;
    }

2 → void blocking_unlock(mutex_t *m) {
      if (queue_empty(m->wait_queue))
        m->holder = 0;
      else {
        m->holder = dequeue(m->wait_queue);
        enqueue(RunQueue, m->holder);
      }
    }
```

- On a multiprocessor, it may not work
  - thread 2 holds the mutex and wait queue is empty and thread 1 tries to lock the mutex at the same time thread 2 is releasing the mutex
  - thread 1 may wait forever

# Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock); // okay to spin here
  if (m->holder != 0)
    enqueue(m->wait_queue, CurrentThread);
    spin_unlock(m->spinlock);
    thread_switch();
  } else {
    m->holder = CurrentThread;
    spin_unlock(m->spinlock);
  }
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  } else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

- Has a different problem

# Working Blocking Locks (?)

```
    void blocking_lock(mutex_t *m) {
      spin_lock(m->spinlock);
      if (m->holder != 0)
        enqueue(m->wait_queue, CurrentThread);
        spin_unlock(m->spinlock);
1       thread_switch();
      } else {
        m->holder = CurrentThread;
        spin_unlock(m->spinlock);
      }
    }

    void blocking_unlock(mutex_t *m) {
      spin_lock(m->spinlock);
      if (queue_empty(m->wait_queue)) {
        m->holder = 0;
      else {
2       m->holder = dequeue(m->wait_queue);
        enqueue(RunQueue, m->holder);
      }
      spin_unlock(m->spinlock);
    }
```

Thread 2 can move thread 1 to another processor! (Can it?)

# Working Blocking Locks (?)

```
void blocking_lock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (m->holder != 0)
    enqueue(m->wait_queue, CurrentThread);
    spin_unlock(m->spinlock);
    thread_switch();
  } else {
    m->holder = CurrentThread;
    spin_unlock(m->spinlock);
  }
}

void blocking_unlock(mutex_t *m) {
  spin_lock(m->spinlock);
  if (queue_empty(m->wait_queue)) {
    m->holder = 0;
  else {
    m->holder = dequeue(m->wait_queue);
    enqueue(RunQueue, m->holder);
  }
  spin_unlock(m->spinlock);
}
```

**Solution is to do `spin_unlock()` inside `thread_switch()`**

# Futexes

- ***Futex:*** fast user-mode mutexes
  - safe, efficient kernel conditional queueing in Linux
    - most of the time when you try to lock a mutex, it's unlocked; so just go ahead and lock it (no system call)
    - if it's locked (by another thread), then a system call is required for this thread to obtain the lock
  - contained in it is an unsigned integer state called `value` and a queue of waiting threads
- Two *system calls* are provided to support futexes

```
futex_wait(futex_t *futex, int val) {
  if (futex->val == val)
    sleep();
}

futex_wake(futex_t *futex) {
  // wake up one thread from wait queue if
  //      there is any
  ...
}
```

# Ancillary Functions

- Add 1 to `*val`, return its original value

```
unsigned int atomic_inc(unsigned int *val) {
  // performed atomically
  return((*val)++); // textbook is wrong
}
```

- Subtract 1 to `*val`, return its original value

```
unsigned int atomic_dec(unsigned int *val) {
  // performed atomically
  return((*val)--); // textbook is wrong
}
```

# Attempt 1

- `futex->val`
  - 0 means unlocked; otherwise, locked

```
void lock(futex_t *futex) {
  unsigned int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

- Problem with `unlock()`
  - slow because `futex_wake()` is a system call
- Problem with `lock()`
  - threads run in lock steps in a multiprocessor environment!
  - `futex->val` may wrap-around

# Attempt 2

- `futex->val` can only take on values of 0, 1, and 2
  - 0 means unlocked
  - 1 means locked but no waiting thread
  - 2 means locked with the possibility of waiting threads

textbook is wrong

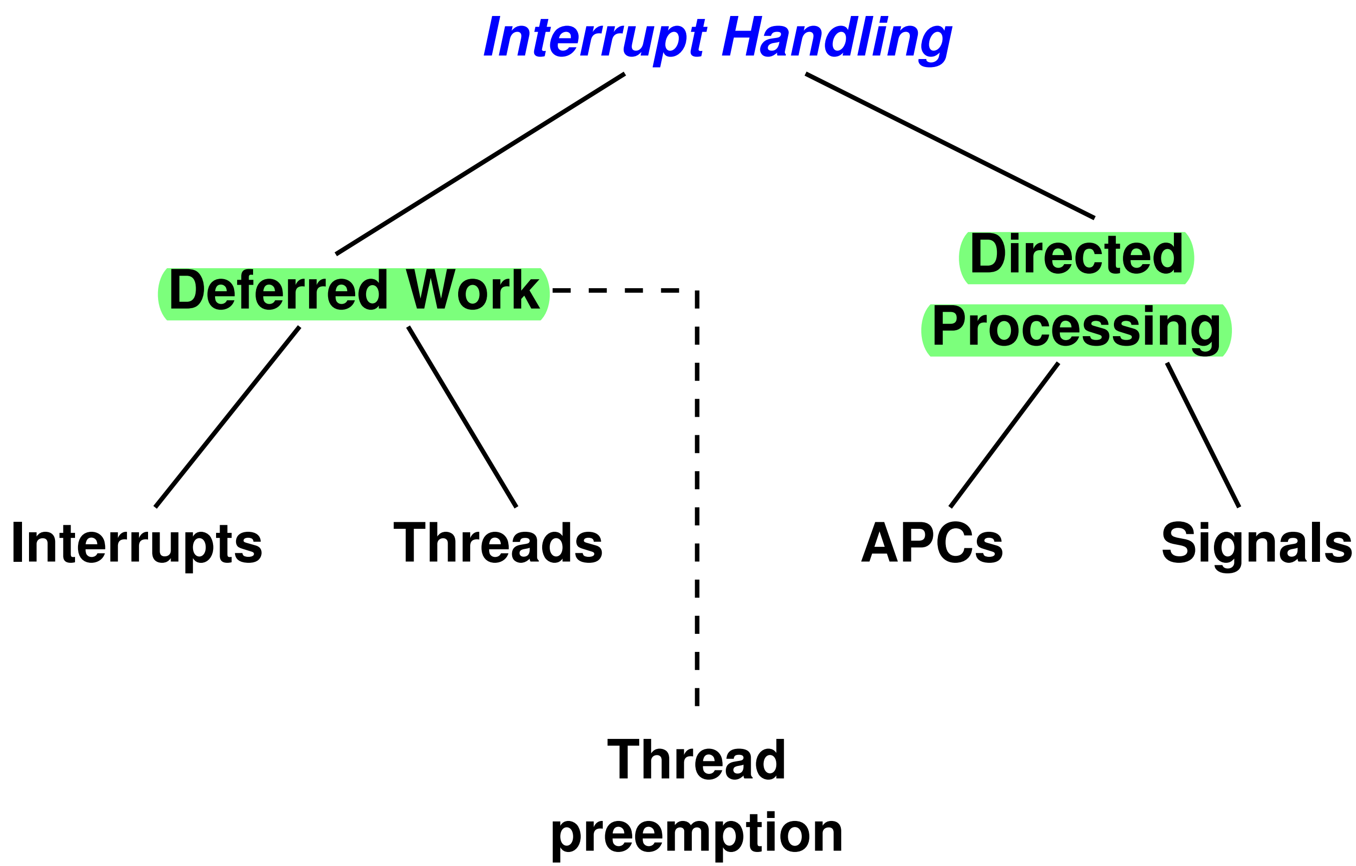
```
void lock(futex_t *futex) {
  unsigned int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) == 1))
        futex_wait(futex, 2);
    } while ((c = CAS(&futex->val, 0, 2)) != 0));
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

# Thread Synchronization Summary

- Spin locks
  - used if the duration of waiting is expected to be small
    - as in the case at the beginning of `blocking_lock()`
- Sleep (or blocking) locks
  - used if the duration of waiting is expected to be long
- Futexes
  - optimized version of blocking locks
- In your kernel assignmen #1, you need to implement *kernel* threads
  - very different from *user* threads
    - keep in mind that the `weenix` kernel is *non-preemptive*
    - the kernel is all powerful (and therefore, must be bug free)
  - in kernel assignmen #3, you need to implement user threads
    - variable-weight processes
      - `clone()`

# 5.2 Interrupts

# Interrupt Handling - Overview



# Interrupt Handling

- We are focusing on dealing with synchronization/concurrency issues
- What to do if you have *non-preemption kernels*?
  - in these systems, you *never* preempt a thread running in the kernel
    - threads running in privileged mode yield the processor only voluntarily
    - as the thread returned from the kernel, it can be preempted
    - this makes the kernel a lot easier to implement!
      - because don't have to implement *locking inside the kernel* (except to watch out for interrupts)
    - done in early Unix systems
    - done in weenix
      - related to your kernel assignment #1
  - use *interrupt masking*

# Interrupt Handling

- What to do if you have *preemption kernels*?
  - threads running in privileged mode may be forced to yield the processor
  - so you *disable preemption*
    - then you can use interrupt masking
  - *spin locks*

# Interrupt Masking

- Unmasked interrupts interrupt current processing
- What causes interrupts to be masked?
  - the occurrence of a particular class of interrupts masks further occurences
  - explicit programmatic action
- Some architectures impose a hierarchy of interrupt levels
  - Intel architectures use APIC
    - advanced programmable interrupt controller

# Non-Preemptive Kernel Synchronization

```
int X = 0;
```

```
void AccessXThread() {
  ...
  X = X+1;
  ...
}
```

```
void AccessXInterrupt() {
  ...
  X = X+1;
  ...
}
```

- Sharing a variable between a thread and an interrupt handler
  - since we have a non-preemptive kernel, the only thing that can prevent a kernel thread from executing till completion is an interrupt
- The above code does not work
  - cannot use locks to fix it

# Non-Preemptive Kernel Synchronization

```
int X = 0;
```

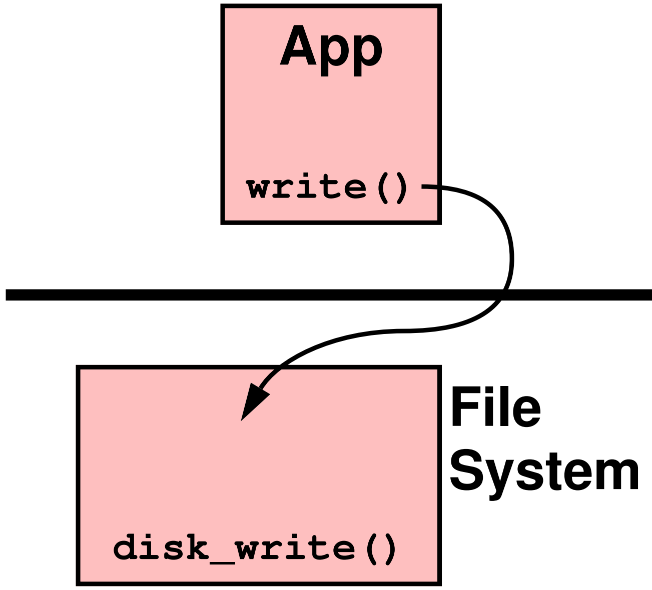
```
void AccessXThread() {
  int oldIPL;
  oldIPL = setIPL(IHLevel);
  X = X+1;
  setIPL(oldIPL);
}
```

```
void AccessXInterrupt() {
  ...
  X = X+1;
  ...
}
```

- Solution is to mask the interrupt
  - works well in a non-preemptive kernel

# Example: Disk I/O

```
int disk_write(...) {
  ...
  startIO(); // start disk operation
  ...
  enqueue(disk_waitq, CurrentThread);
  thread_switch();
    // wait for disk operation to complete
  ...
}

void disk_intr(...) {
  thread_t *thread;
  ...
  // handle disk interrupt
  ...
  thread = dequeue(disk_waitq);
  if (thread != 0) {
    enqueue(RunQueue, thread);
    // wakeup waiting thread
  }
  ...
}
```



# Example: Disk I/O

```
int disk_write(...) {
    ...
    startIO(); // start disk operatio
1   ...
    enqueue(disk_waitq, CurrentThread
    thread_switch();
      // wait for disk operation to c
    ...
}

void disk_intr(...) {
    thread_t *thread;
    ...
    // handle disk interrupt
    ...
2   thread = dequeue(disk_waitq);
    if (thread != 0) {
      enqueue(RunQueue, thread);
      // wakeup waiting thread
    }
    ...
}
```

**Problem**
- disk may be too fast
- `disk_intr()` gets called before `enqueue()`
- this is a synchronization problem / race condition

# Improved Disk I/O

```
int disk_write(...) {
  ...
  oldIPL = setIPL(diskIPL);
  startIO();  // start disk operati
  ...
  enqueue(disk_waitq, CurrentThread
  thread_switch();
    // wait for disk operation to c
  setIPL(oldIPL);
  ...
}
```

**Solution**

- mask disk interrupt

# Improved Disk I/O

```
int disk_write(...) {
   ...
   oldIPL = setIPL(diskIPL);
   startIO();  // start disk operati
   ...
   enqueue(disk_waitq, CurrentThread
   thread_switch();
     // wait for disk operation to c
   setIPL(oldIPL);
   ...
}
```

**Solution**

- mask disk interrupt

**Doesn't quite work!**

- `thread_switch()` will switch to another thread and won't return back here any time soon to unmask interrupt
  - who will enable the disk interrupt?
  - complication caused by the fact that `thread_switch()` does not function like a normal procedure call
- moving `setIPL(oldIPL)` to before `thread_switch()` may have race condition in accessing the `RunQueue`

# Modified `thread_switch`

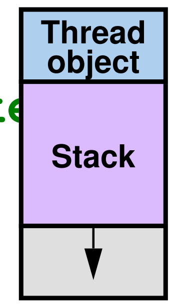
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by
    //     masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts, then check
    //     RunQueue
    setIPL(0);  // 0 means no interrupts are masked
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context,
              CurrentThread->context);
  setIPL(oldIPL);
}
```

# Modified `thread_switch`

```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to Run
    //      masking all inter
  while(queue_empty(RunQueue)
    // repeatedly allow interrupts, then check
    //      RunQueue
    setIPL(0);  // 0 means no
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(Ru
  swapcontext(OldThread->cont
              CurrentThread->
  setIPL(oldIPL);
}
```

This code is actually much more tricky that it looks
- it can be invoked by a thread that's not doing I/O
- `oldIPL` is the `oldIPL` of a different thread!

Let's say that another thread calls `thread_switch()`
- it's not doing I/O
- its `oldIPL` is set to 0




Now we call `thread_switch()`
- our `oldIPL` set to `diskIPL`
- then we switch to this other thread and set IPL to 0 (disk interrupt enabled)
- `RunQueue` only accessed when *all interrupts blocked*