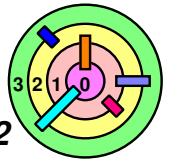
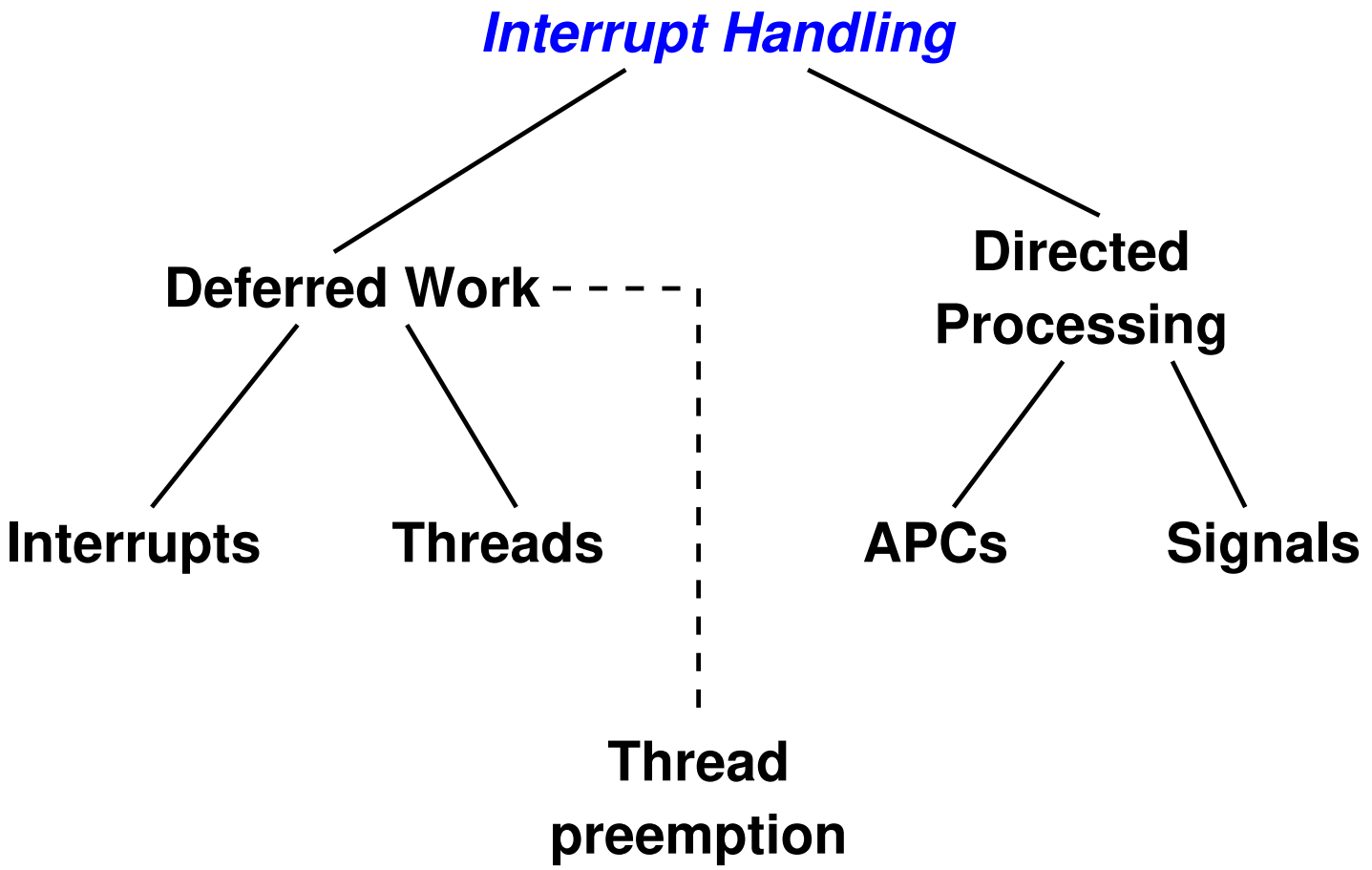
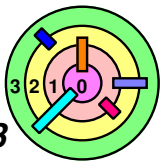
5.2 Interrupts



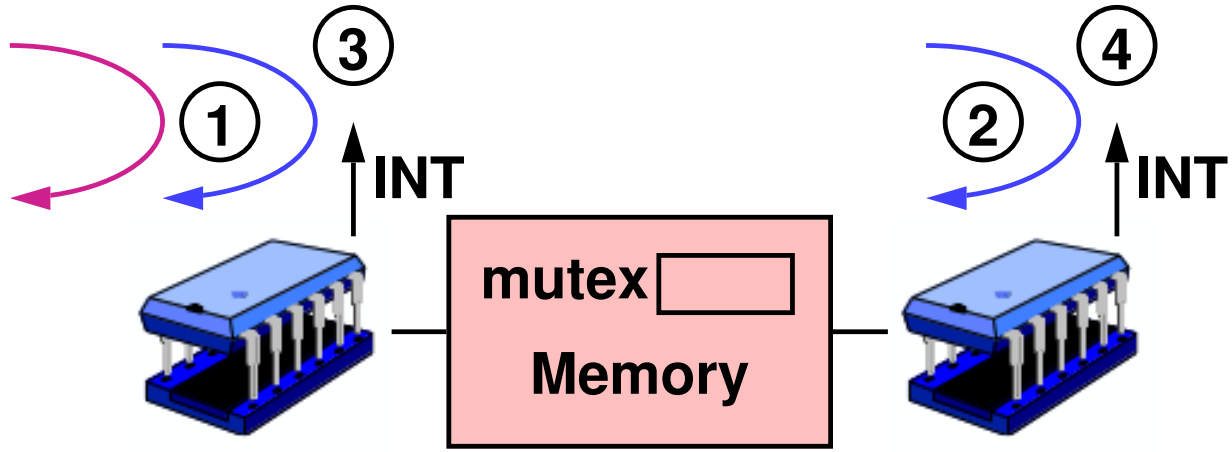
Interrupt Handling - Overview





Thread Synchronization

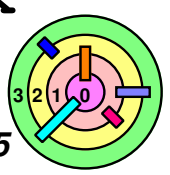
- Recall asynchronous activies that may require concurrency control
  1) *another thread* running on the *same processor* may *preempt* this thread and accesses the same data structure
  2) *another thread* running on *another processor* might access the same data structure
  3) an *interrupt handler* running on the *same processor* that accesses the same data structure
  4) an *interrupt handler* running on *another processor* might access the same data structure



- Futex is a solution to (1) and (2)
  - let's look at (3) and (4) now (kernel only)

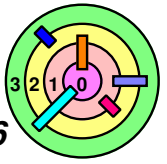
Interrupt Handling

- We are focusing on dealing with synchronization/concurrency issues
- What to do if you have *non-preemption kernels*?
  - in these systems, a kernel thread can *never* be preempted *by another thread*
    - may switch to interrupt context, but return to *same thread*
    - threads running in privileged mode yield the processor only voluntarily
    - this makes the kernel a lot easier to implement!
      - because don't have to implement *locking inside the kernel* for every shared data structure (although sometimes, mutex is still needed to synchronize kernel threads)
    - done in early Unix systems
    - done in weenix
      - this is like your kernel 1 with `DRIVERS=1` in `Config.mk`
  - use *interrupt masking* to access variables shared with ISRs



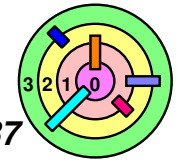
Interrupt Handling

- 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*
  - for multiple CPUs, use a *spin lock* to lock out other CPUs
- How do you disable preemption?
  - just use a global variable
    - before you preempt a kernel thread, check this global flag
      - if the flag says preemption is disabled, don't preempt CurrentThread



Interrupt Masking

- Unmask interrupts interrupt current processing (if interrupt is pending)
- What causes interrupts to be masked?
  - the occurrence of a particular class of interrupts masks further occurences
    - i.e., inside an interrupt service routine (ISR)
  - explicit programmatic action
  - some architectures impose a hierarchy of interrupt levels
    - e.g., Intel architectures use APIC (Advanced Programmable Interrupt Controller)

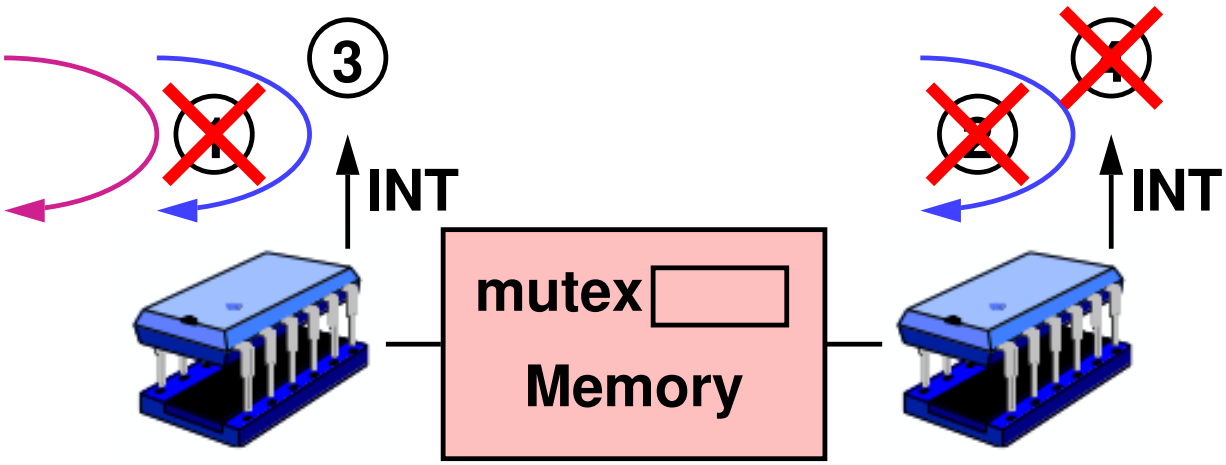


Non-Preemptive Kernel Synchronization

```
int X = 0;
```

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

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



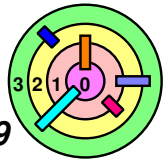
Non-Preemptive Kernel Synchronization

```
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
  - analogous to cannot use mutex inside a signal handler



Non-Preemptive Kernel Synchronization

```
int X = 0;
```

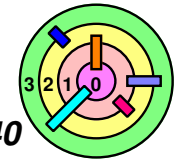
```
void AccessXThread() {
  int oldIPL;
  oldIPL = setIPL(IHLevel);
  X = X+1;
  setIPL(oldIPL);
}
```

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

- Solution is to mask the interrupt
  - this is analogous to the solution in Ch 2 to mask signals
  - seems to work well in a non-preemptive kernel
    - let's look at something more realistic

      Q: what *data structures* do you really share between *thread* code and an *interrupt service routine*?

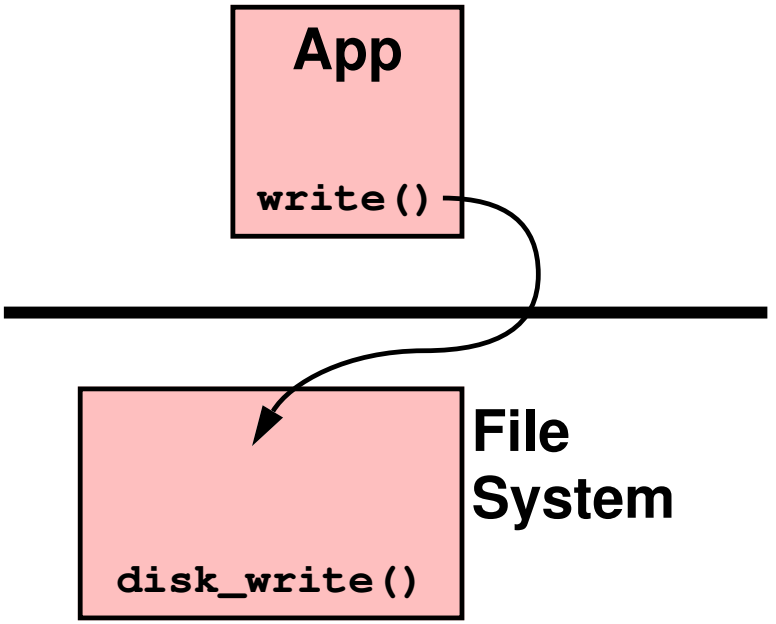
      A: an *I/O queue* and the *RunQueue*

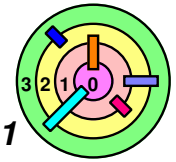


More Relistic Example: Disk I/O

```
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
   }
   ...
}
```





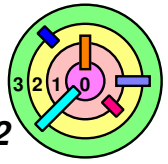
More Relistic Example: Disk I/O

```
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*

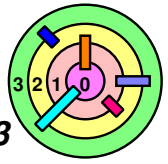


Improved Disk I/O

```
int disk_write(...) {
   ...
   int 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

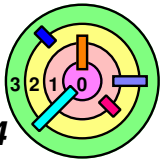
```
int disk_write(...) {
   ...
   int oldIPL = setIPL(diskIPL);
   startIO();  // start disk operati
   ...
   enqueue(disk_waitq, CurrentThread
   thread_switch();
     // wait for disk operation to c
   setIPL(oldIPL);
   ...
}
```

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`



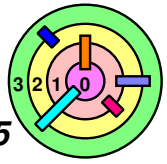
Modified `thread_switch`

```
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);  // unmask all interrupts
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context,
              CurrentThread->context);
  setIPL(oldIPL);
}
```

IMPORTANT: must only access `RunQueue` when *all interrupts are blocked*

- because `RunQueue` is accessed in *all* interrupt handlers

⇨ **You can (and should) use this in kernel 1**



Modified `thread_switch`

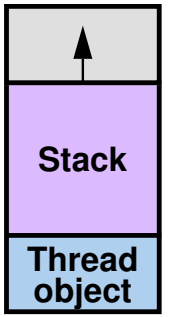
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQ
    //      masking all interr
  while(queue_empty(RunQueue)
    // repeatedly allow interrupts,
    //      then check RunQue
    setIPL(0);  // unmask all
    setIPL(HIGH_IPL);
  }
  // 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!

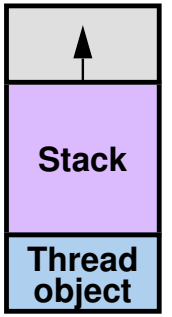
Let's say that another thread calls `thread_switch()`

- it's not doing I/O
- its `oldIPL` is set to 0

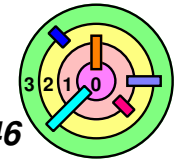


Now we call `thread_switch()`

- our `oldIPL` set to `diskIPL`
- then we switch to this other thread and set IPL to 0 (disk interrupt enabled)



You can use this in kernel 1 & 2



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);  // unmask all interrupts
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context,
              CurrentThread->context);
  setIPL(oldIPL);
}
```

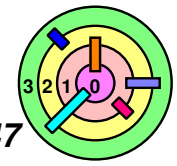
Problem - *busy waiting*

- can take a long time to get an interrupt
- the correct way to wait is to wait while sleeping
- since there's nothing to run, should halt the CPU

Note:

- it's okay to do this in kernel 1 and kernel 2
- this *wont' work for kernel 3*

You can use this in kernel 1 & 2



Modified thread_switch

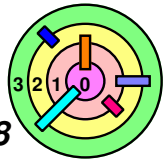
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by
    //      masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts,
    //      then check RunQueue
    setIPL(0);  // unmask all interrupts
    HLT // enable interrupt & halt CPU
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context,
              CurrentThread->context);
  setIPL(oldIPL);
}
```

If you decide to halt the CPU in `weenix`, need to watch out for a *race condition*

- this code does not "wait" properly
- the correct way to wait for an *asynchronous* event is:
  1) disable/block it
  2) if event hasn't ocurred, enable/unblock and wait for it in one *atomic* operation

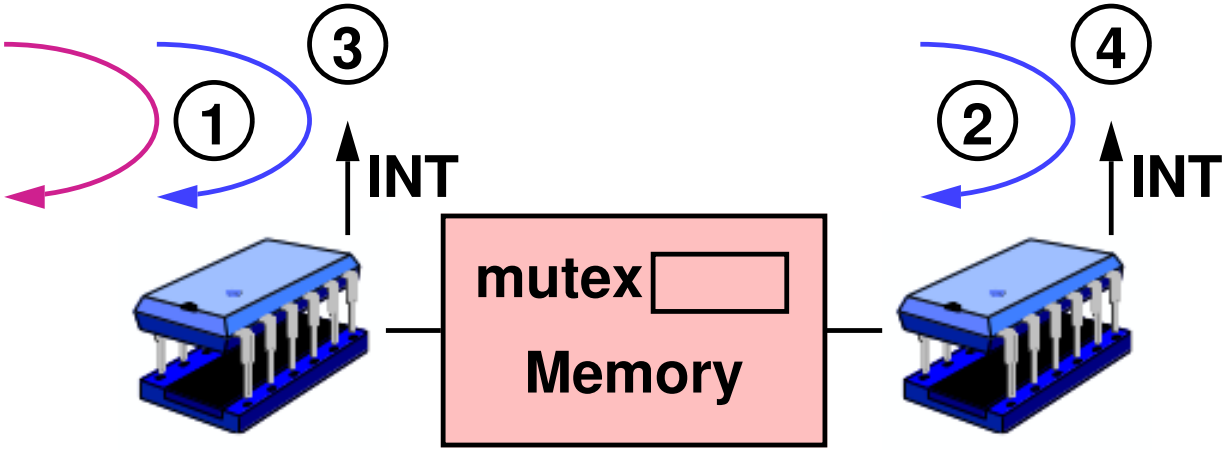
Note:

- you have to do it this way (and *fix* the race condition) for kernel 3
  - or live with the possibility of system freezing



Preemptive Kernels & Multiple CPUs

- What's different?
- Recall asynchronous activies that may require concurrency control
  1) *another thread* running on the *same processor* may *preempt* this thread and accesses the same data structure
  2) *another thread* running on *another processor* might access the same data structure
  3) an *interrupt handler* running on the *same processor* that accesses the same data structure
  4) an *interrupt handler* running on *another processor* might access the same data structure



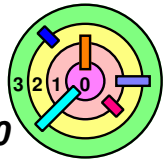
Solution?

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

- Does it work?
  - no, can deadlock in `AccessXInterrupt()` in case (1) when thread code and interrupt handler runs on the *same processor*



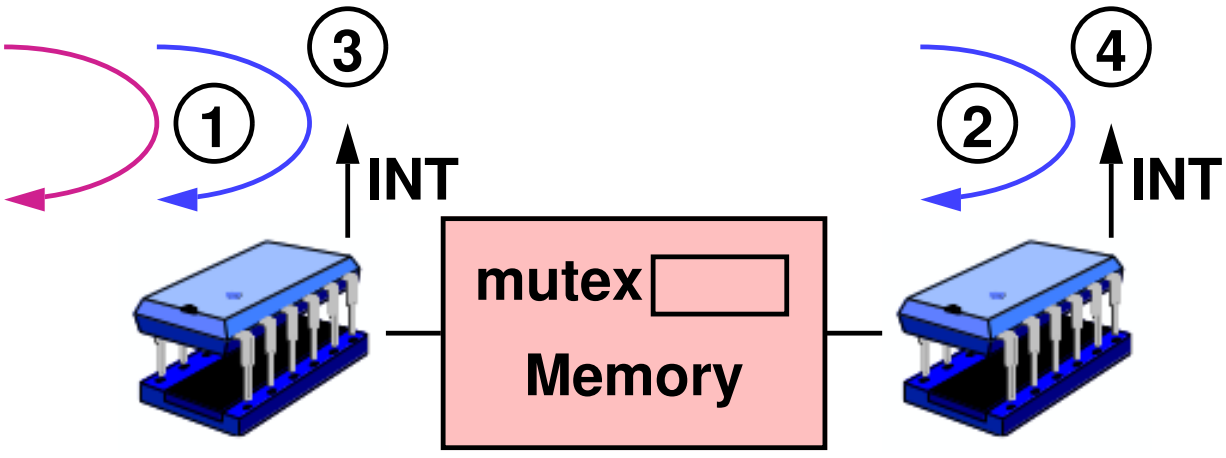
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

Does it work?



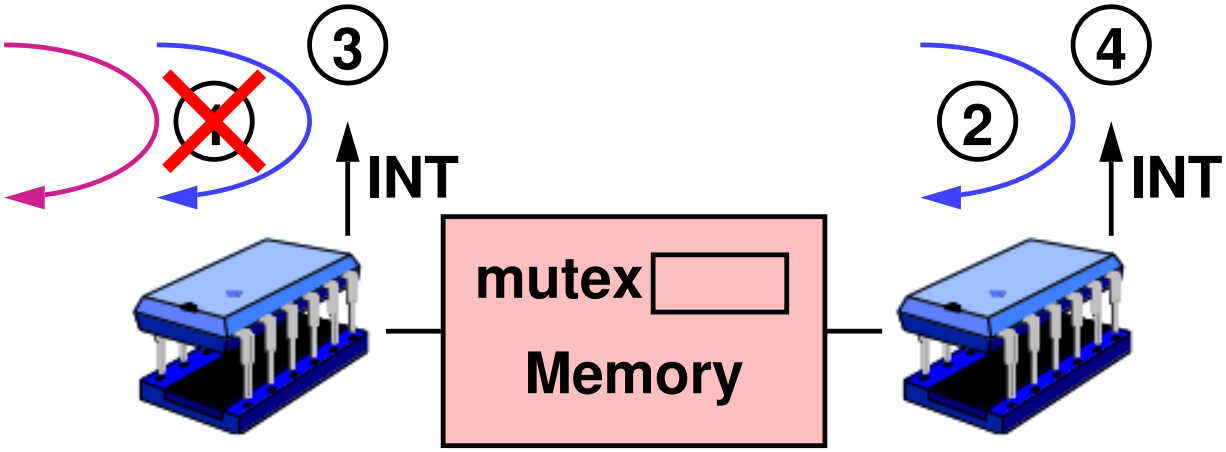
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

Does it work?



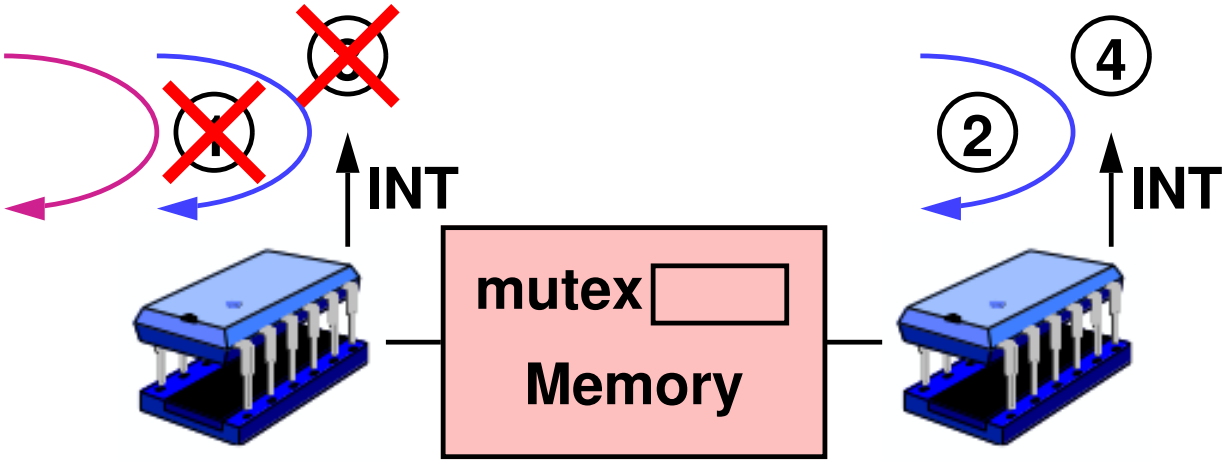
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

Does it work?



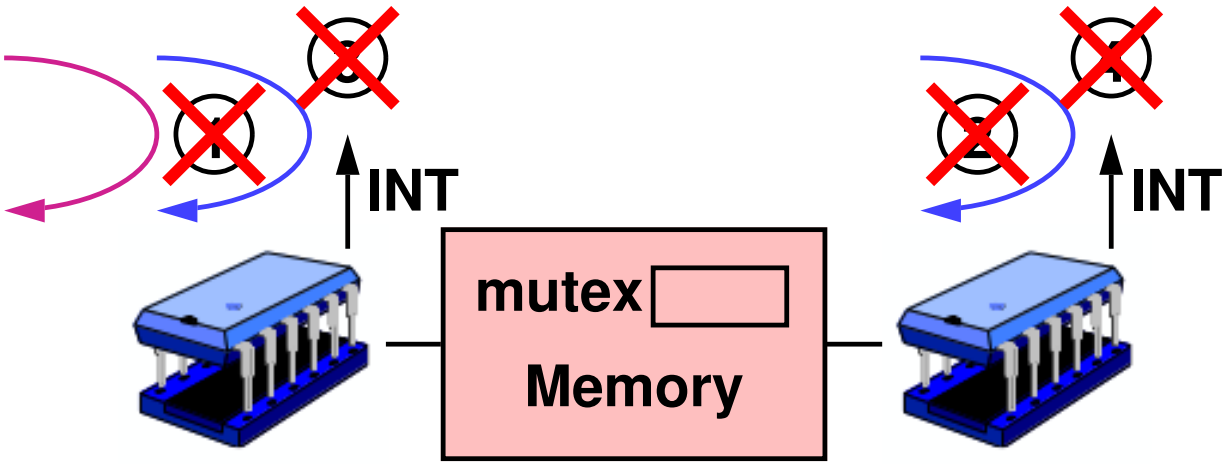
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

⇨ Does it work?



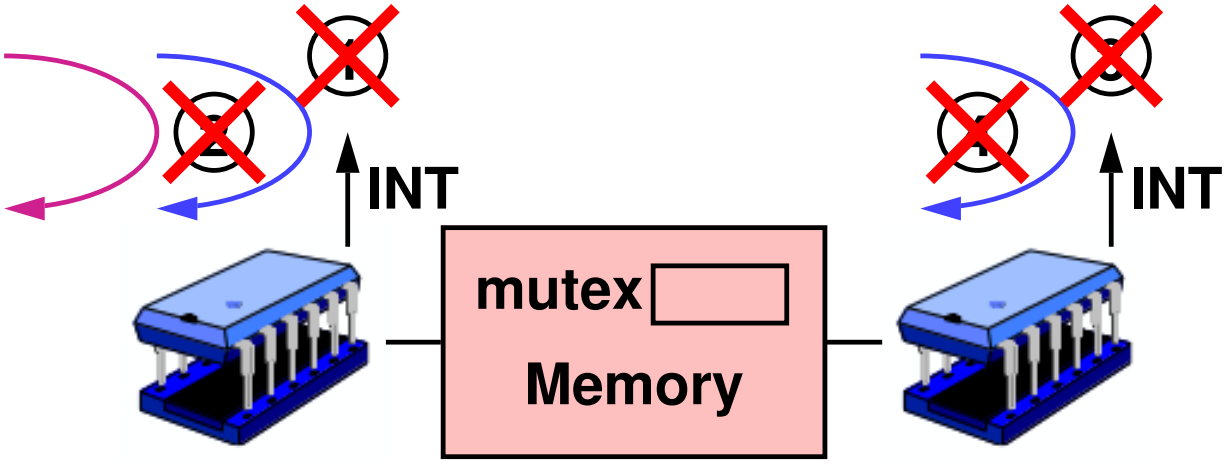
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

- Does it work?
  - yes
    - order is important



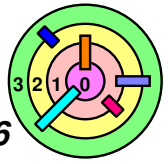
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

- Can we switch the order of `DisablePreemption()` and `MaskInterrupts()`?
  - if you mask interrupt first, what would happen if you switch to a thread what would mess with interrupt masking?



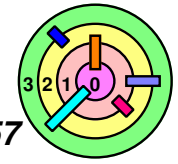
Solution ...

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  DisablePreemption();
  MaskInterrupts();
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  UnMaskInterrupts();
  EnablePreemption();
}
```

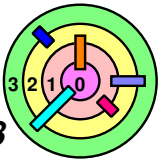
```
void AccessXInterrupt() {
  ...
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
  ...
}
```

- What X are we really talking about?
  - it may be the RunQueue (use *highest* IPL)
  - it may be the I/O queue (use *appropriate* IPL)
  - it may be a *futex queue* (no need to mask interrupt)



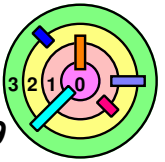
Interrupt Threads?

- Solaris allows interrupts to be handled as threads
- Does it make sense to handle interrupts with threads?
  - perhaps similar to using `sigwait` for handling signals with threads
  - what would be the advantages?
  - what would be the disadvantages?



Interrupt Threads

```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  if (!MoreWork)
    return;
  else
    BecomeThread( );
  ...
  P(Semaphore); // sleep!
  ...
}
```



Interrupt Threads In Action