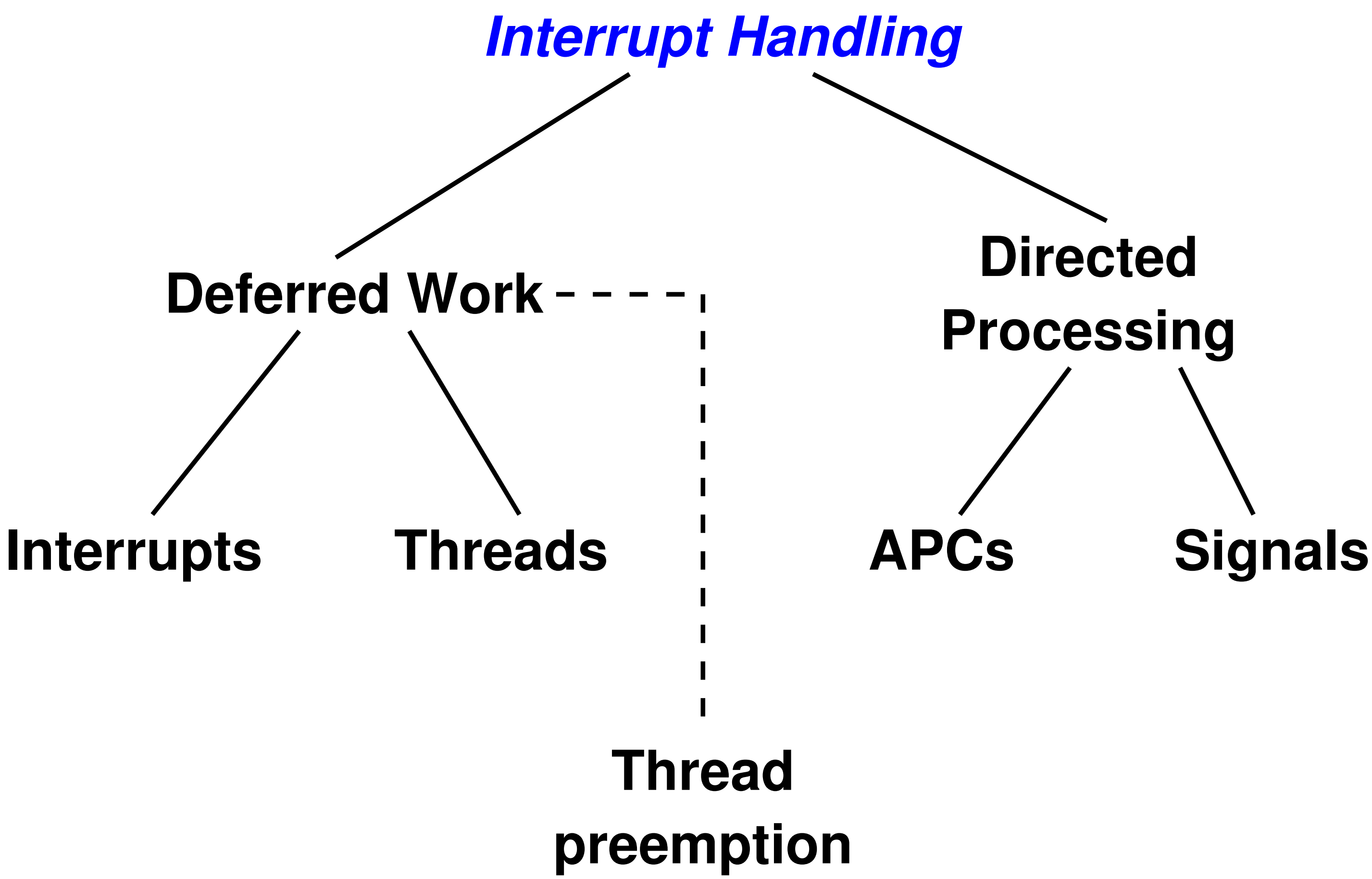
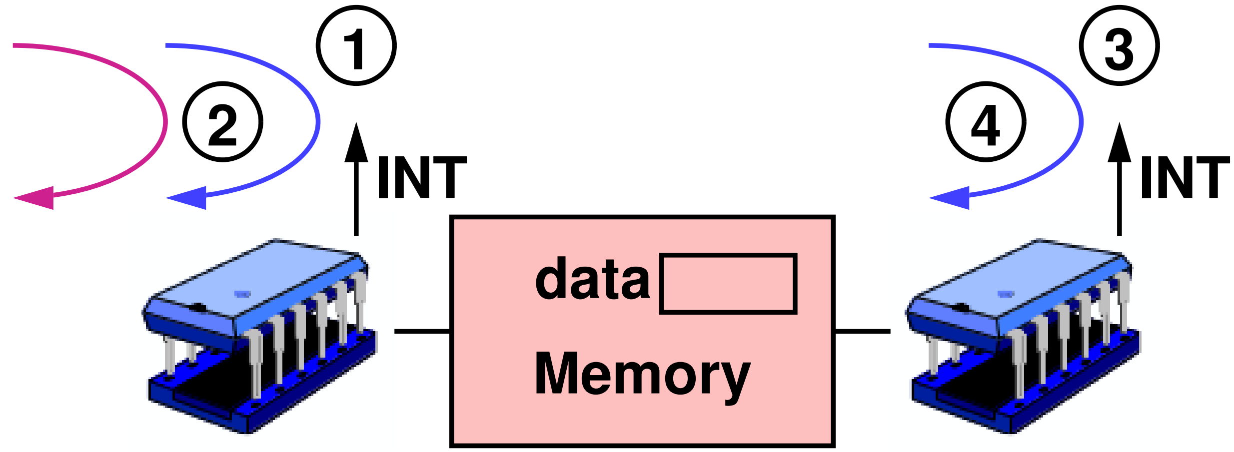
5.2 Interrupts

Interrupt Handling - Overview



Thread Synchronization

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



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

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*
    - 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*

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);
}
```

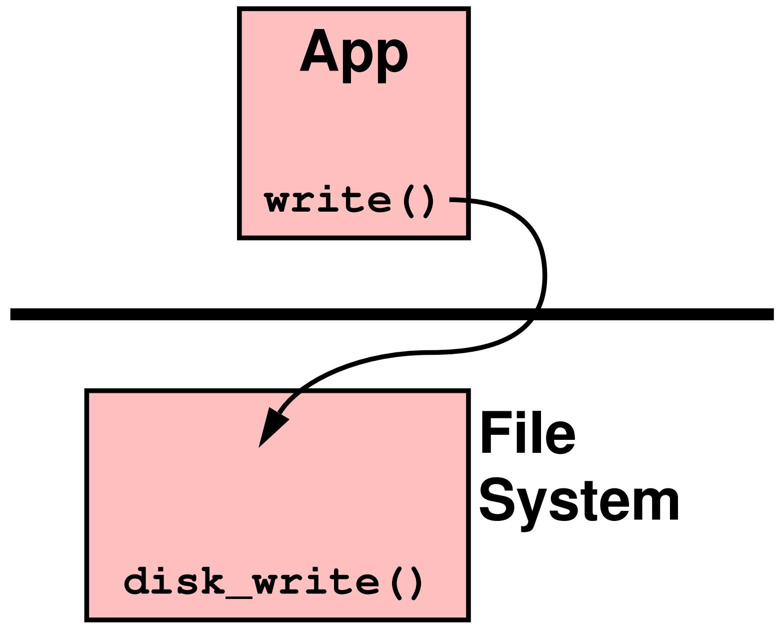
```
void AccessXInterrupt() {
  ...
  X = X+1;
  ...
}
```

- 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`

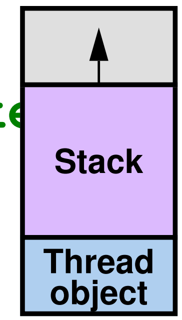
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by
    //     masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts, then check
    //     RunQueue
    setIPL(0);  // 0 means no interrupts 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`

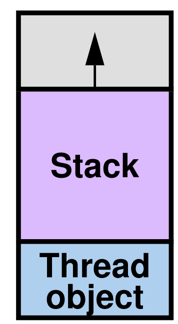
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to Run
    //      masking all inter
  while(queue_empty(RunQueue)
    // repeatedly allow interrupts, then check
    //      RunQueue
    setIPL(0);  // 0 means no
    setIPL(HIGH_IPL);
  }
  // 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*



Modified `thread_switch`

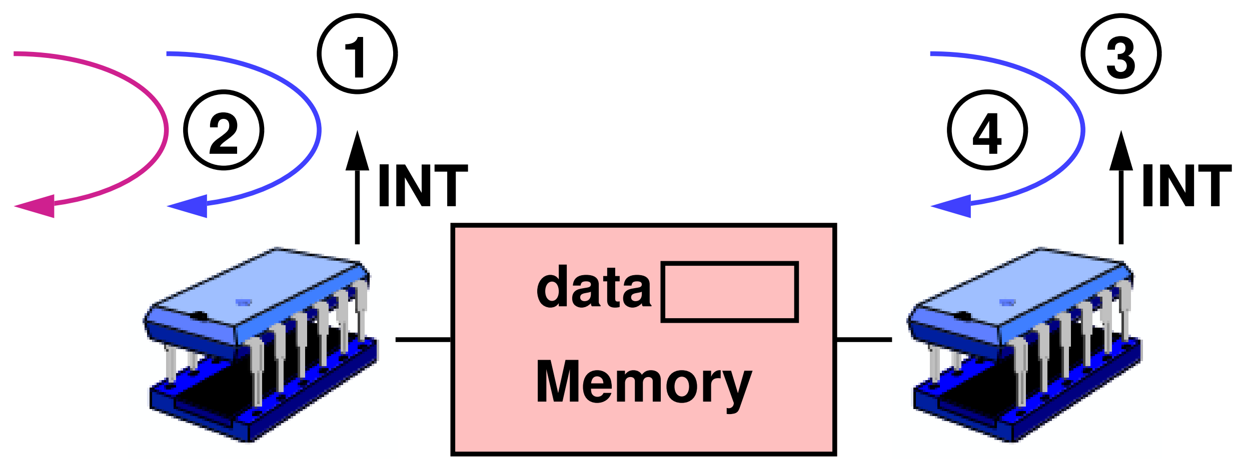
```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by
    //     masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts, the
    //     RunQueue
    setIPL(0);  // 0 means no interrupts
    HLT // should halt the 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*

- it does not "wait" properly
- the correct way to wait for an *asynchronous* event is:
  1) disable/block it
  2) enable/unblock and wait for it in one *atomic* operation

Preemptive Kernels

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

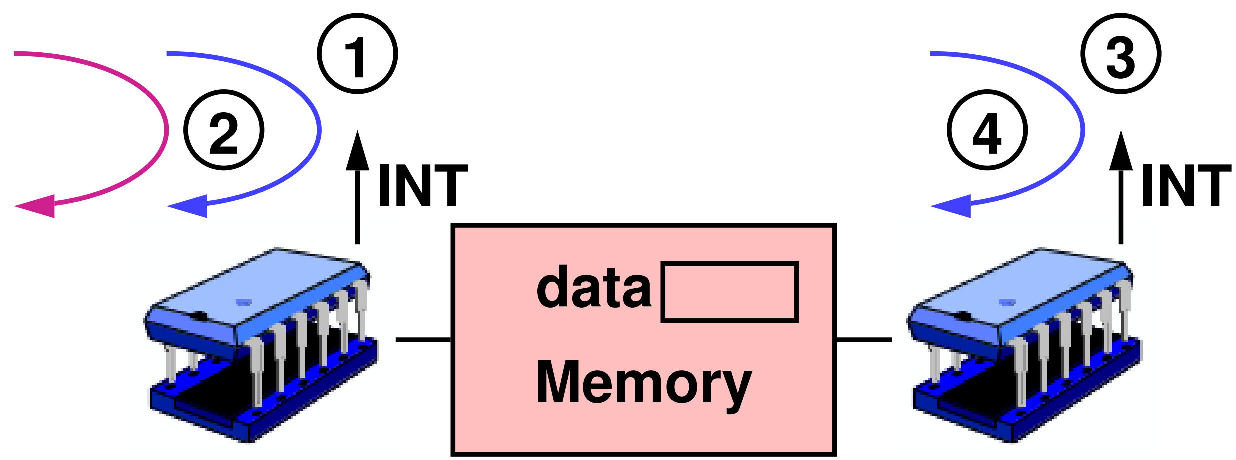
Solution ...

```
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?



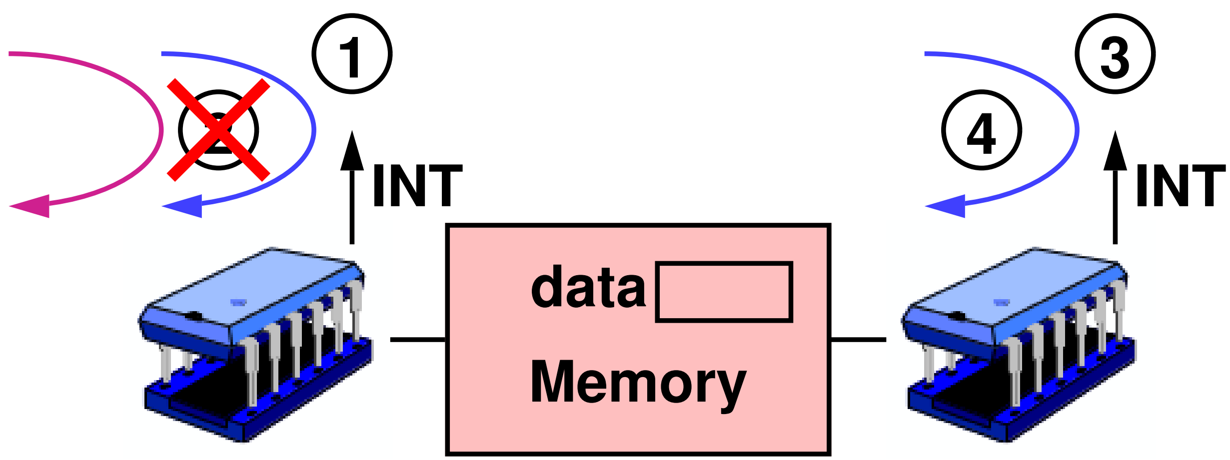
Solution ...

```
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?



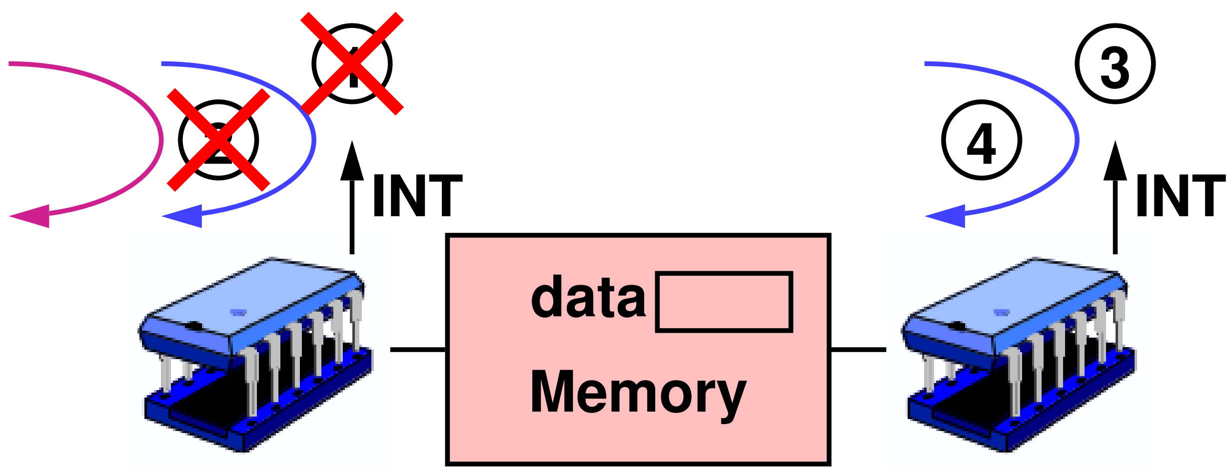
Solution ...

```
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?



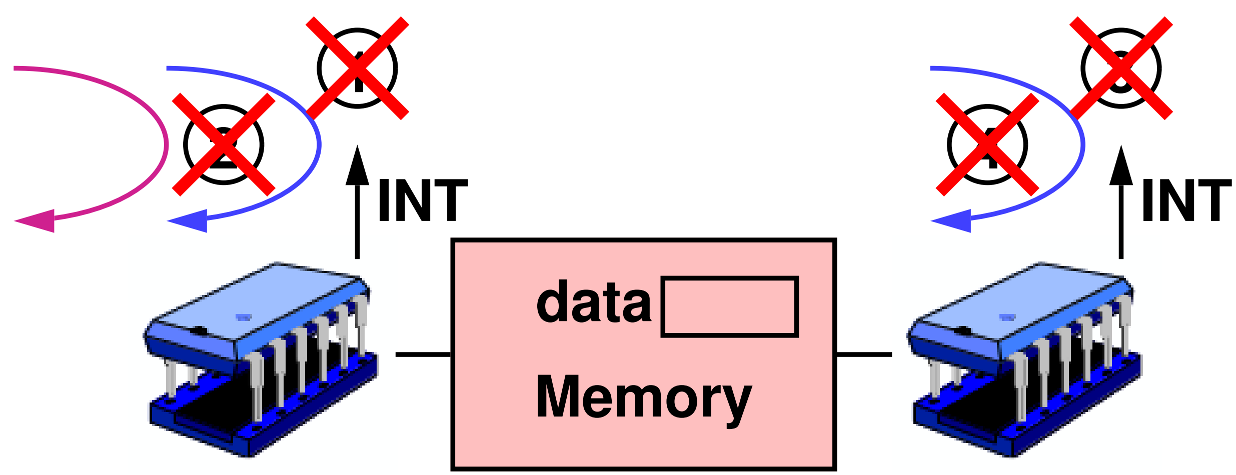
Solution ...

```
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?



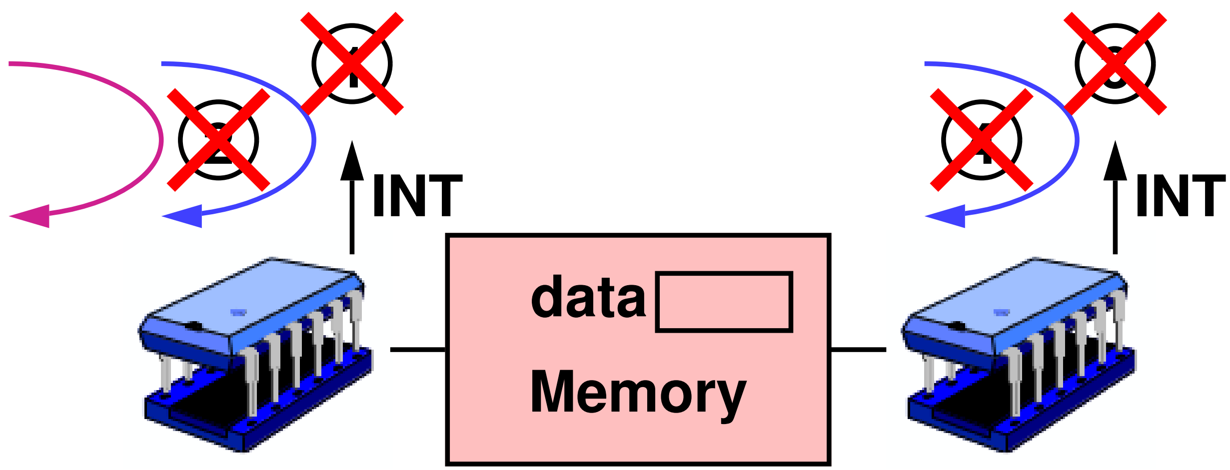
Solution ...

```
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



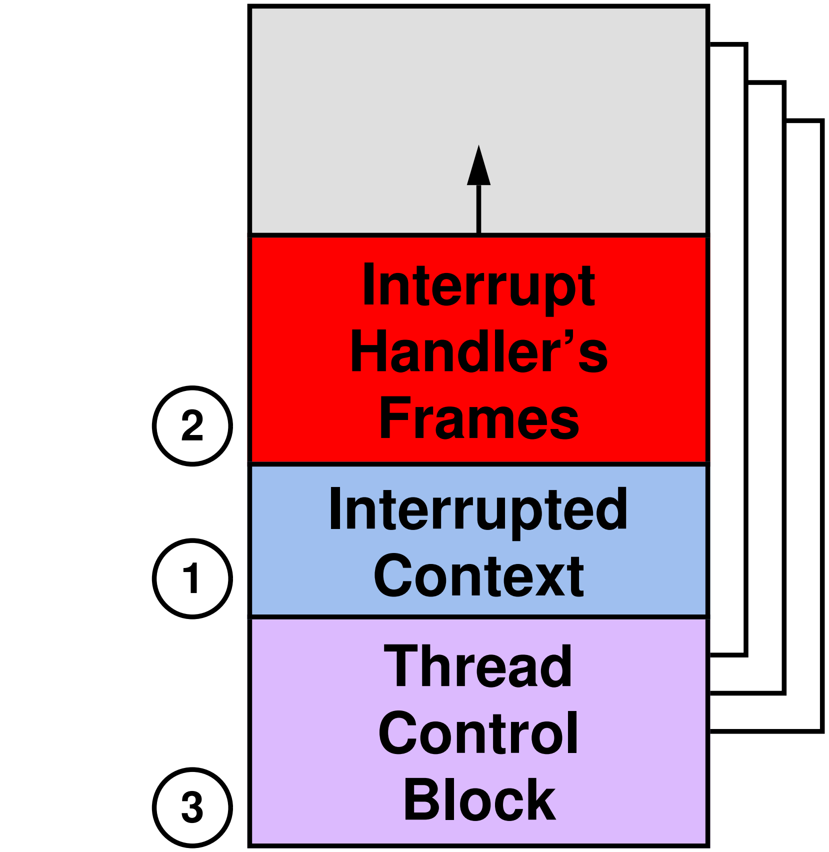
Interrupt Threads?

- 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?

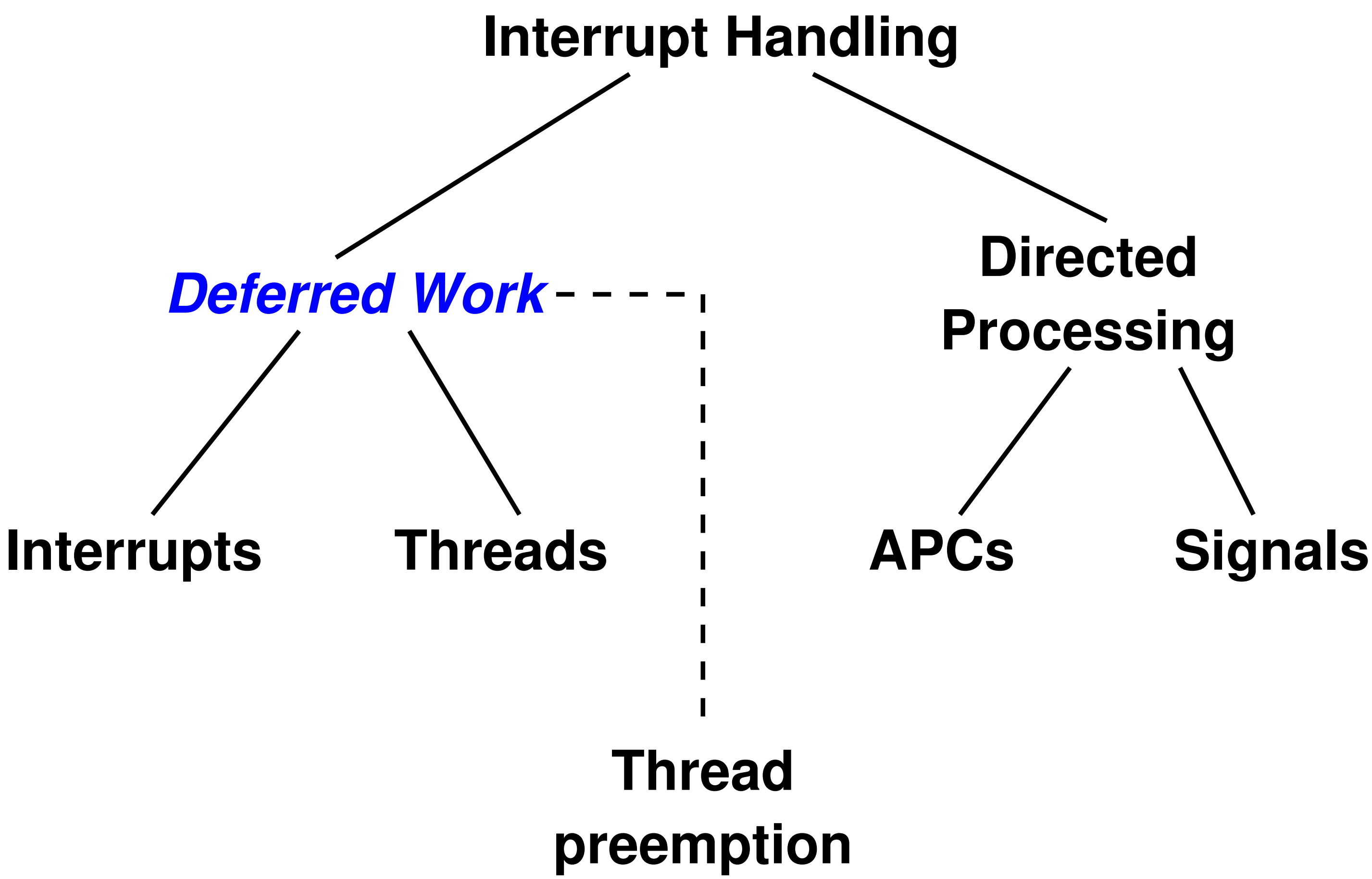
Interrupt Threads

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

Interrupt Threads In Action



Interrupt Handling - Overview



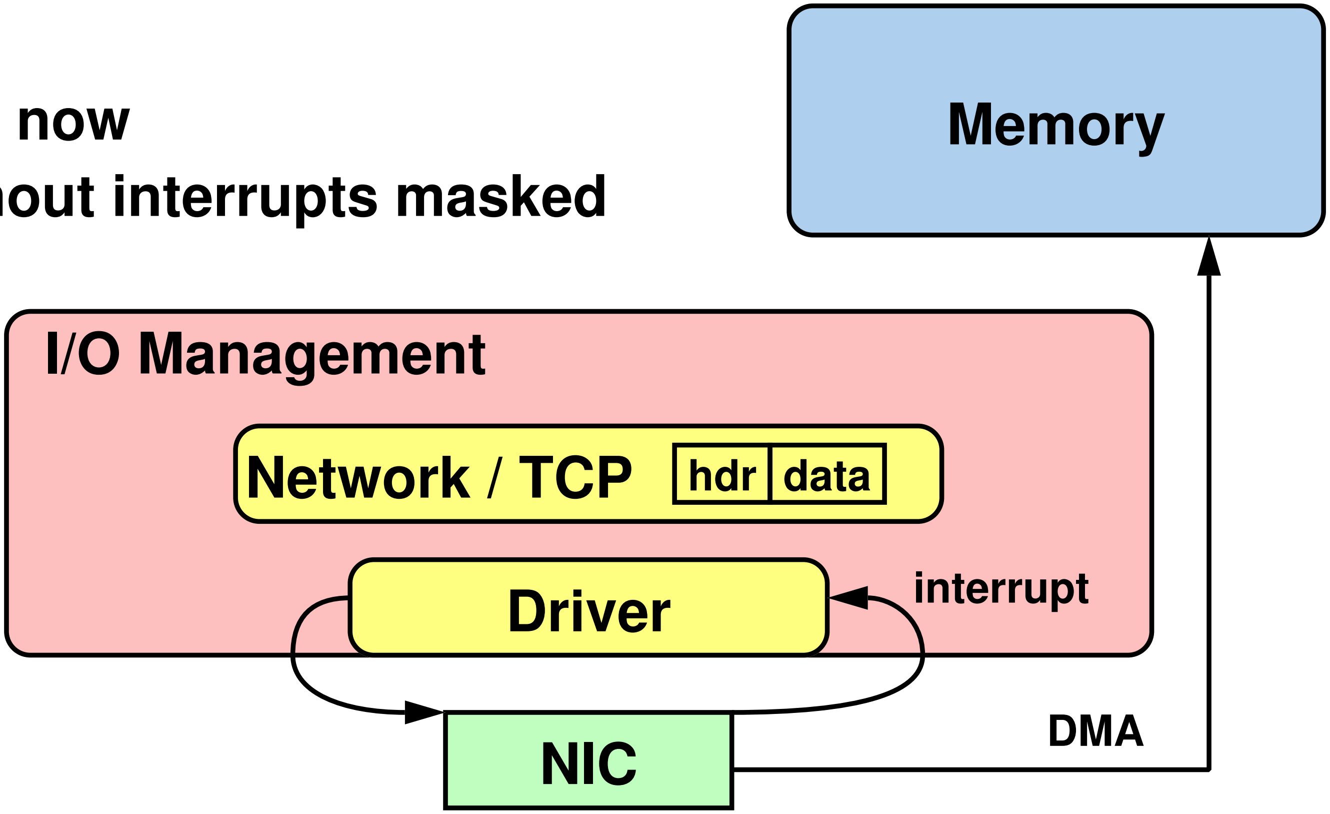
Deferred Work

- Interrupt handlers run with interrupts masked (up to its interrupt priority level)
  - both when executed in interrupt context or thread context
  - may interfere with handling of other interrupts
  - they must run to completion (but may be interrupted by a higher priority interrupt)
    - it must *complete quickly*

- What to do if an interrupt handler has a lot of work to be done?
  - only do what you must do inside the interrupt handler
  - *defer* most of the work to be done after the interrupt handler returns

Deferred Work

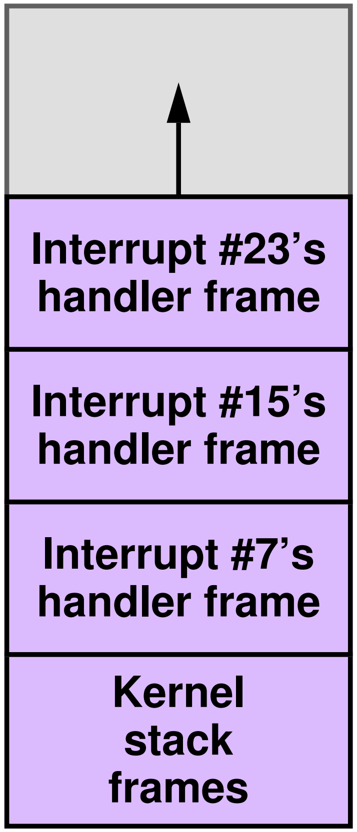
- Ex: network packet processing
  - TCP header processing can take a long time
    - not suitable to do them in a interrupt handler
- Solution
  - do minimal work now
  - do rest later without interrupts masked
- How?



Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt, do minimal work
  ...
  EnQueue(WorkQueue, MoreWork);
}
```



Current thread's kernel stack

Windows Interrupt Priority Levels

hardware	31	High
	30	Power fail
	29	Inter-processor
	28	Clock
	.	.
	.	.
	.	.
	4	Device 2
	3	Device 1
software	2	*DPC*
	1	APC
	0	Thread

Windows handles deferred work in a special interrupt context

- DPC (deferred procedure call) is a *software interrupt*

Deferred Procedure Calls

```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  QueueDPC(MoreWork);
  /* requests a DPC interrupt here */
}

void DPCHandler( ... ) {
  while(!Empty(DPCQueue)) {
    Work = DeQueue(DPCQueue);
    Work();
  }
}
```

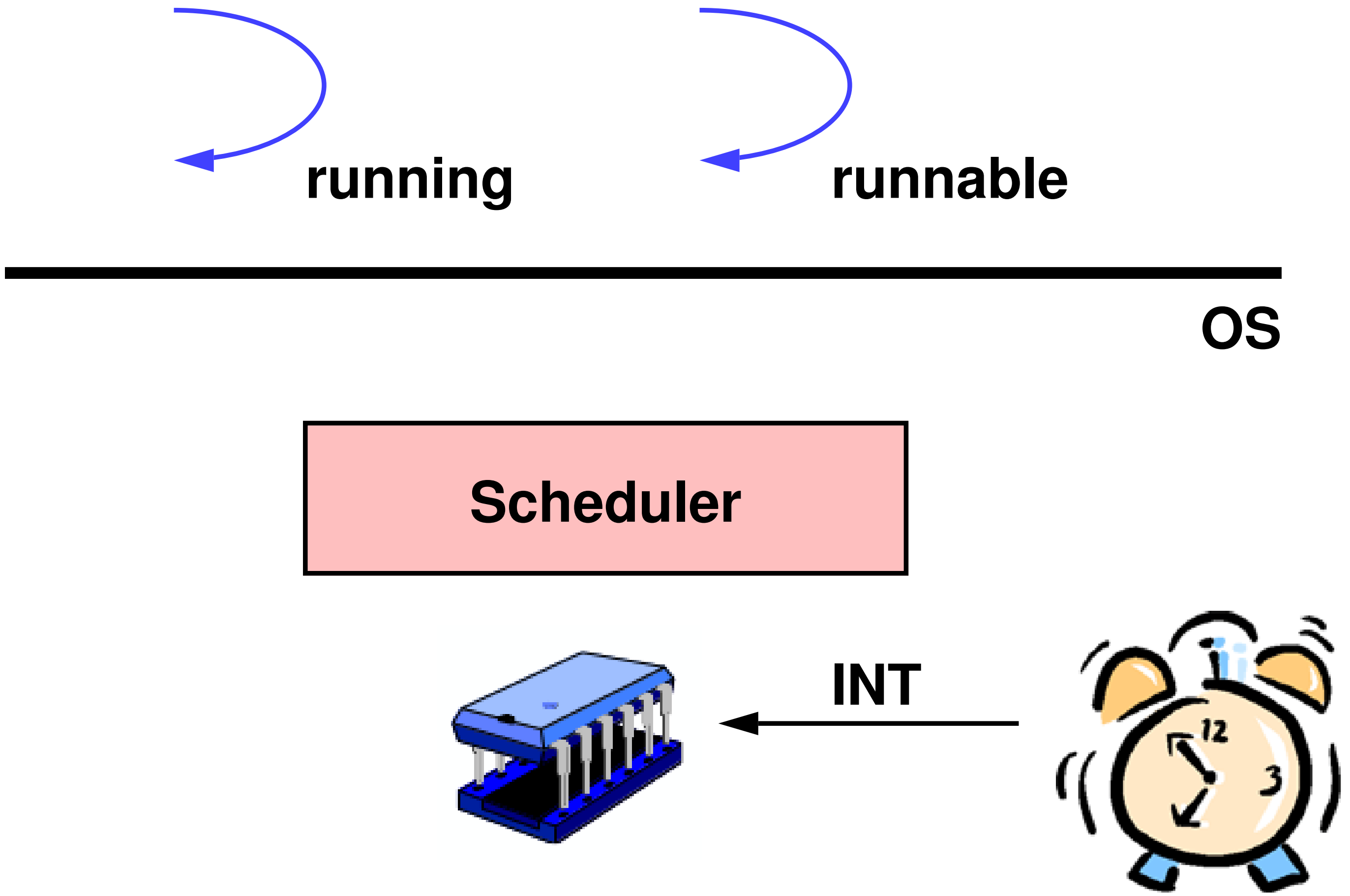
Software Interrupt Threads

- Linux handles deferred work in a special kernel thread
  - this kernel thread is scheduled like any other kernel thread

```
void InterruptHandler( ) {
  // deal with interrupt
  ...
  EnQueue(WorkQueue, MoreWork);
  SetEvent(Work);
}

void SoftwareInterruptThread() {
  while(TRUE) {
    WaitEvent(Work)
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}
```

Thread Preemption



Preemption: User-Level Only

- **Non-preemptive kernel**
  - **preempt only threads running in user mode**
  - **if clock-interrupt happens, just set a global flag**

```
void ClockHandler( ) {
  // deal with clock
  //     interrupt
  ...
  if (TimeSliceOver())
    ShouldReschedule = 1;
}
```

Preemption: User-Level Only

- If interrupted a user thread

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev]();
  if (PreviousMode == UserMode) {
    // the clock interrupted user-mode code
      if (ShouldReschedule)
        Reschedule();
  }
}
```

- If interrupted a kernel thread

```
void TopLevelTrapHandler(...) {
  SpecificTrapHandler();
  if (ShouldReschedule) {
    /* the time slice expired
       while the thread was
       in kernel mode */
    Reschedule();
  }
}
```

`Reschedule()` puts the calling thread on the run queue

- then call `thread_switch()` to give up the processor

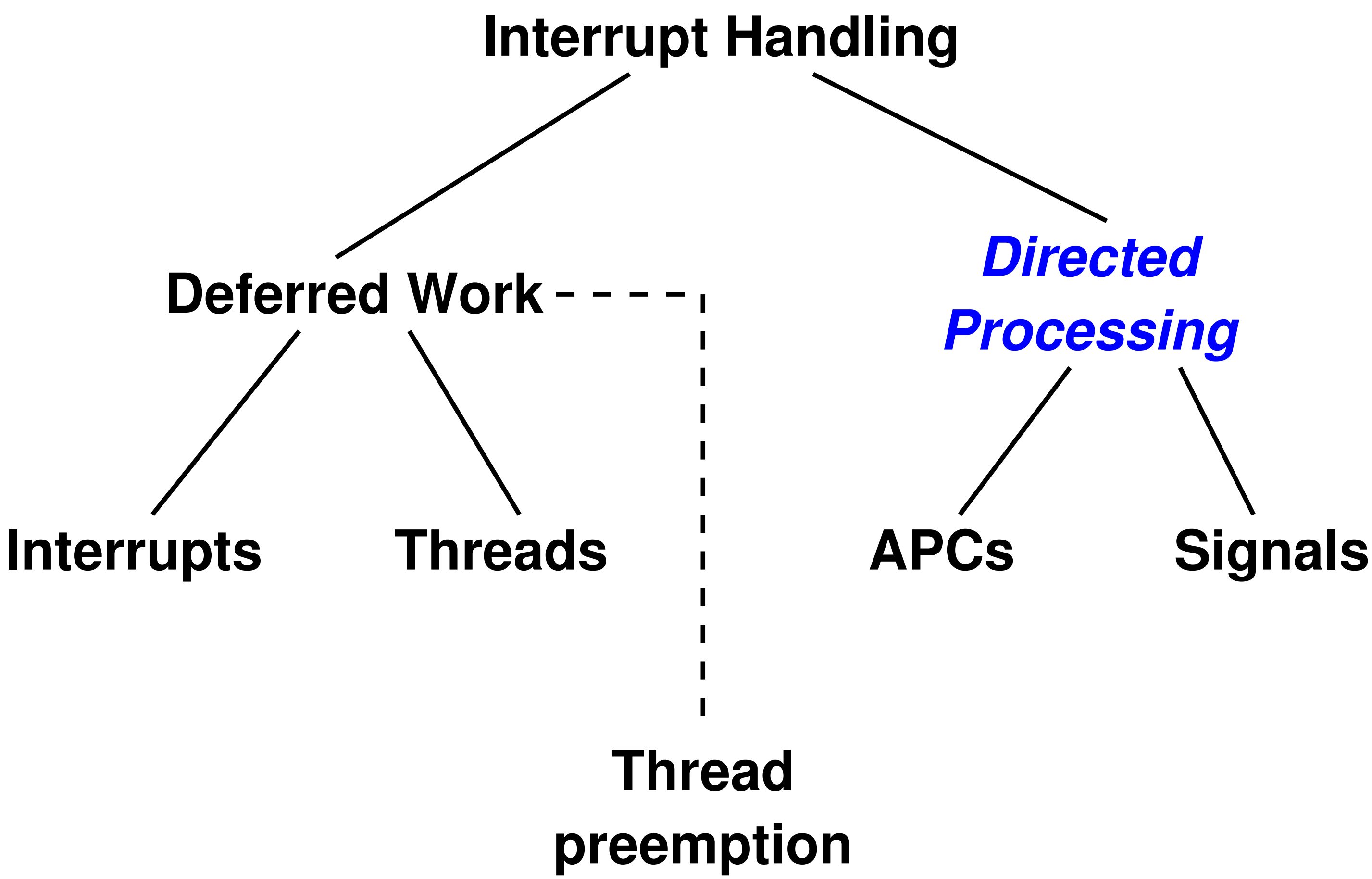
The work of *rescheduling* is *deferred*

Preemption: Full (i.e., Preemptive Kernel)

- Preemptive kernel
  - preemption can happen for a kernel thread
  - if clock-interrupt happens, setup the kernel thread to give up the processor when the processor is about to return to the thread's context
  - how?
    - e.g., add the `Reschedule()` function to the DPC queue

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  if (TimeSliceOver()) {
    QueueDPC(Reschedule);
    /* requests a DPC interrupt here */
  }
}
```

Interrupt Handling - Overview



Directed Processing

- Signals: Unix
  - perform given action in context of a particular thread in user mode
  - e.g., seg fault
    - generated by hardware and needs to be delievered to the user process to invoke a singal handler

- APC: Windows Asynchronous Procedure Calls
  - roughly same thing, but also may be done in ***kernel mode***
    - thus, the APC mechanism is ***more general*** than Unix signals

software

2	DPC
1	***APC***
0	Thread

Windows Interrupt Priority Levels

Invoking the Signal Handler

- Basic idea is to set up the user stack so that the handler is called as a subroutine and so that when it returns, normal execution of the thread may continue

- Complications:
  - saving and restoring registers
    - must first save *all* registers and later on restore all of them
  - signal mask
    - must block the signal and later on unblock the signal
  - therefore, when the signal handler returns, it needs to return to some code that restores all the registers, unblocks the signal, then return to the interrupted code

Invoking the Signal Handler (1)

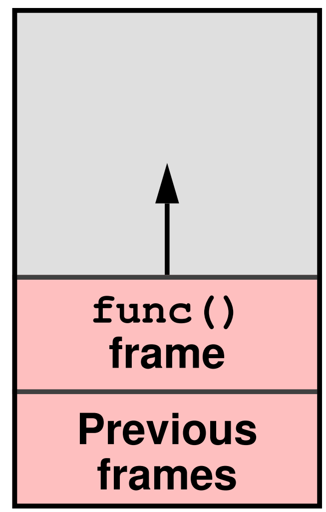
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

IP

Handler

```
sighandler(
  int sig) {
 ...
}
```



User Stack

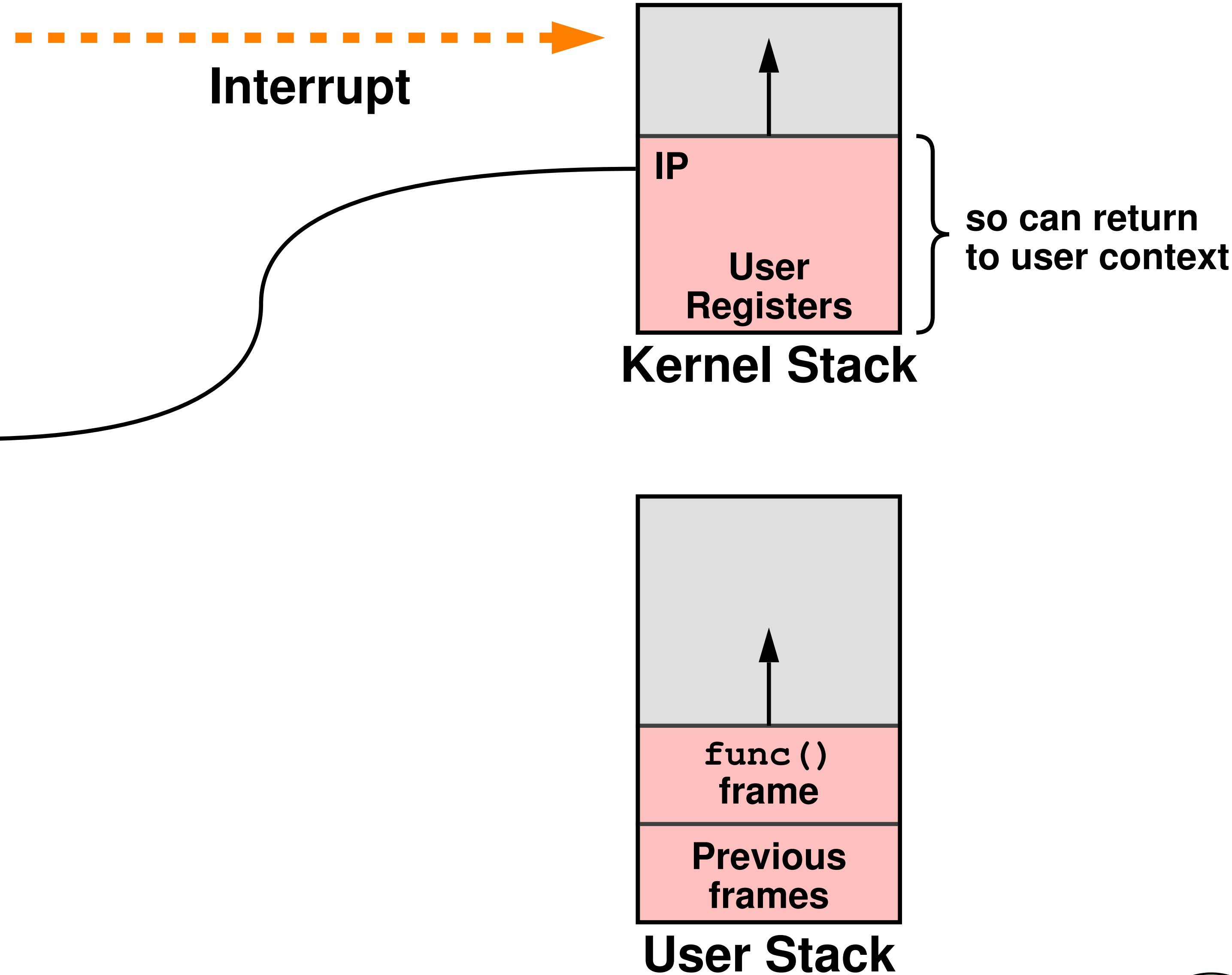
Invoking the Signal Handler (2)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



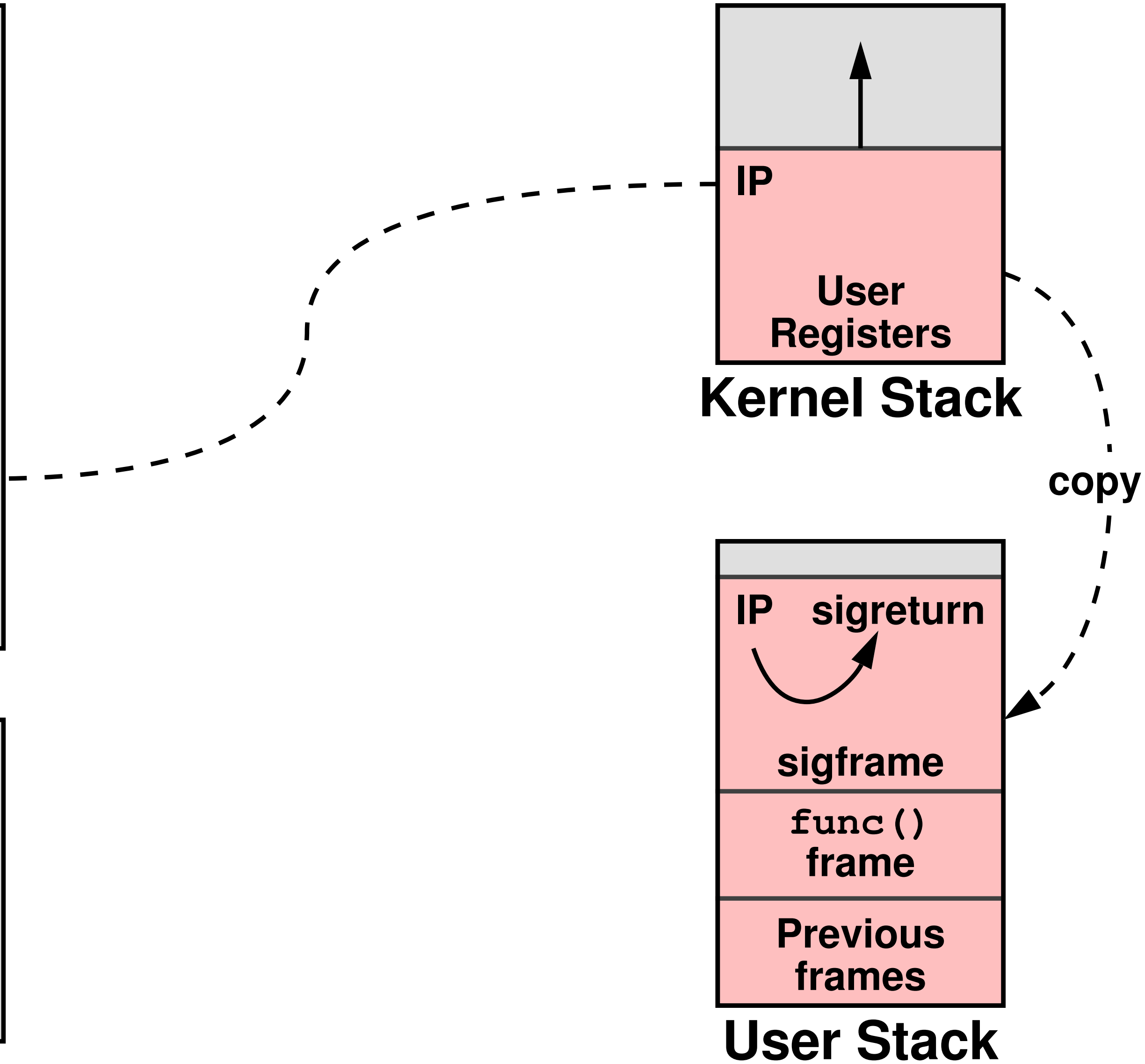
Invoking the Signal Handler (3)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



⇨ **Save user thread context in a sigframe on the user stack**

Invoking the Signal Handler (4)

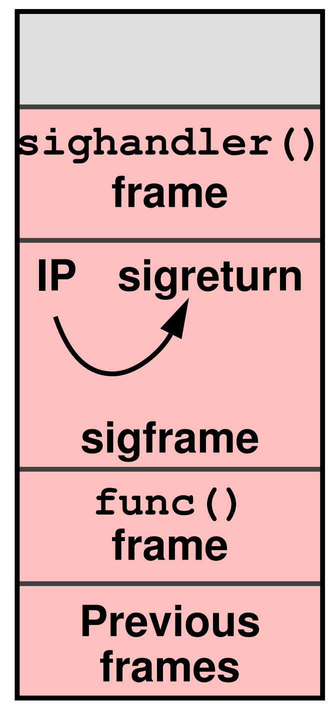
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```

IP



User Stack

- Signal handler executed on the user stack

Invoking the Signal Handler (5)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```

invoke `sigreturn()` system call on return

Kernel Stack

IP sigreturn

sigframe

`func()` frame

Previous frames

User Stack

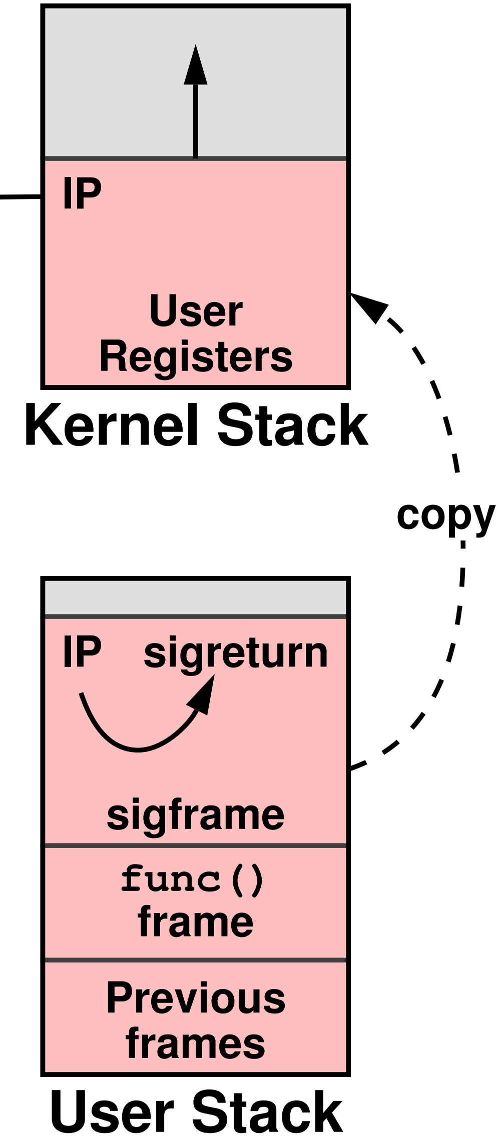
Invoking the Signal Handler (5)

Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

Handler

```
sighandler(
  int sig) {
 ...
}
```



- Copy context back into kernel stack and execute `iret`

Invoking the Signal Handler (6)

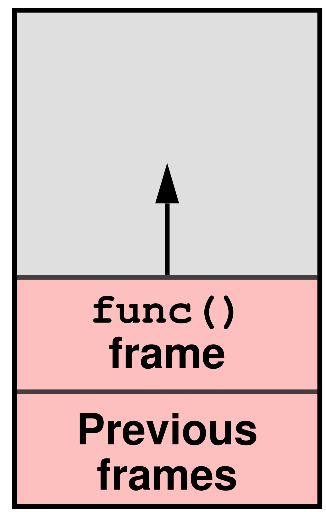
Main Line

```
func(int a1,
     int a2) {
  int i, j = 2;
  for (i=a1;
       i<a2;
       i++) {
    j = j*2;
    j = j/127;
    ...
  }
}
```

IP

Handler

```
signalhandler(
  int sig) {
 ...
}
```



User Stack

Extra Slides

Asynchronous Procedure Calls

- Two uses
  - kernel APC: release of kernel resources
  - user APC: notifying a thread of an external event

Kernel APC

- Release of kernel resources
  - interrupt handler cannot free storage for buffer and control blocks until info passed to process
  - can't be done unless in context of process
  - otherwise address space not mapped in
  - interrupt handler requests kernel APC to have thread, running in kernel mode, absorb info in buffer and control blocks and then free them

User APC

- Notifying thread of external event
  - example: asynchronous I/O
  - thread supplies *completion routine* when starting asynchronous I/O request
  - called in thread's context when I/O completes
    - similar to a Unix signal
    - called only when thread is in *alertable wait state*
      - an option in certain blocking system calls

APC Implementation

- Per-thread list of pending APCs
  - on notification, thread executes them
- User APC
  - thread in alertable state is woken up and executes pending APCs when it returns to user mode
- Kernel APC
  - running thread interrupted by APC interrupt (lowest priority interrupt)
  - waiting thread is "unwaited"
  - execute pending kernel APCs