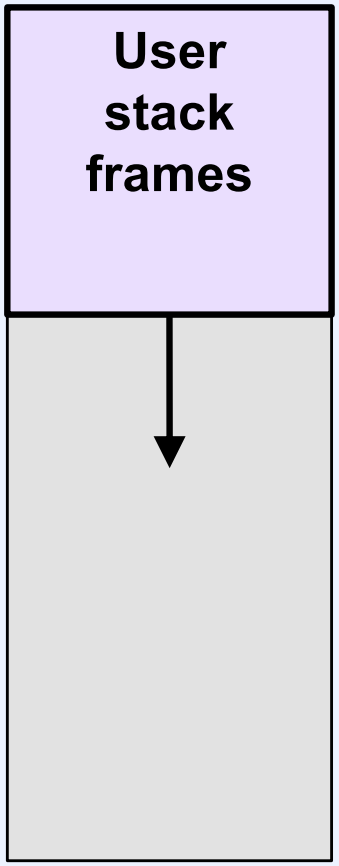
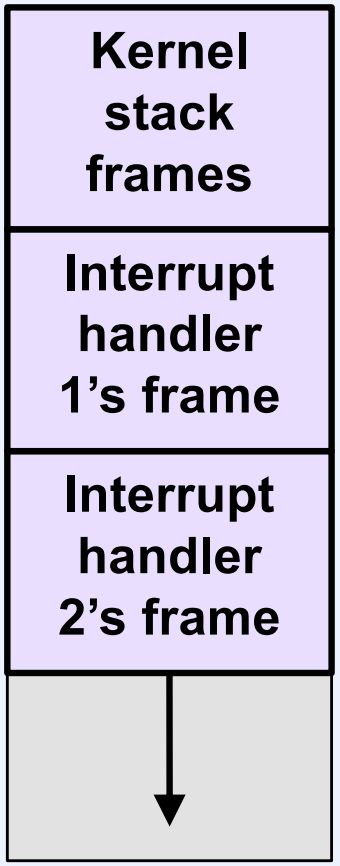
Interrupts, Etc.

Interrupts

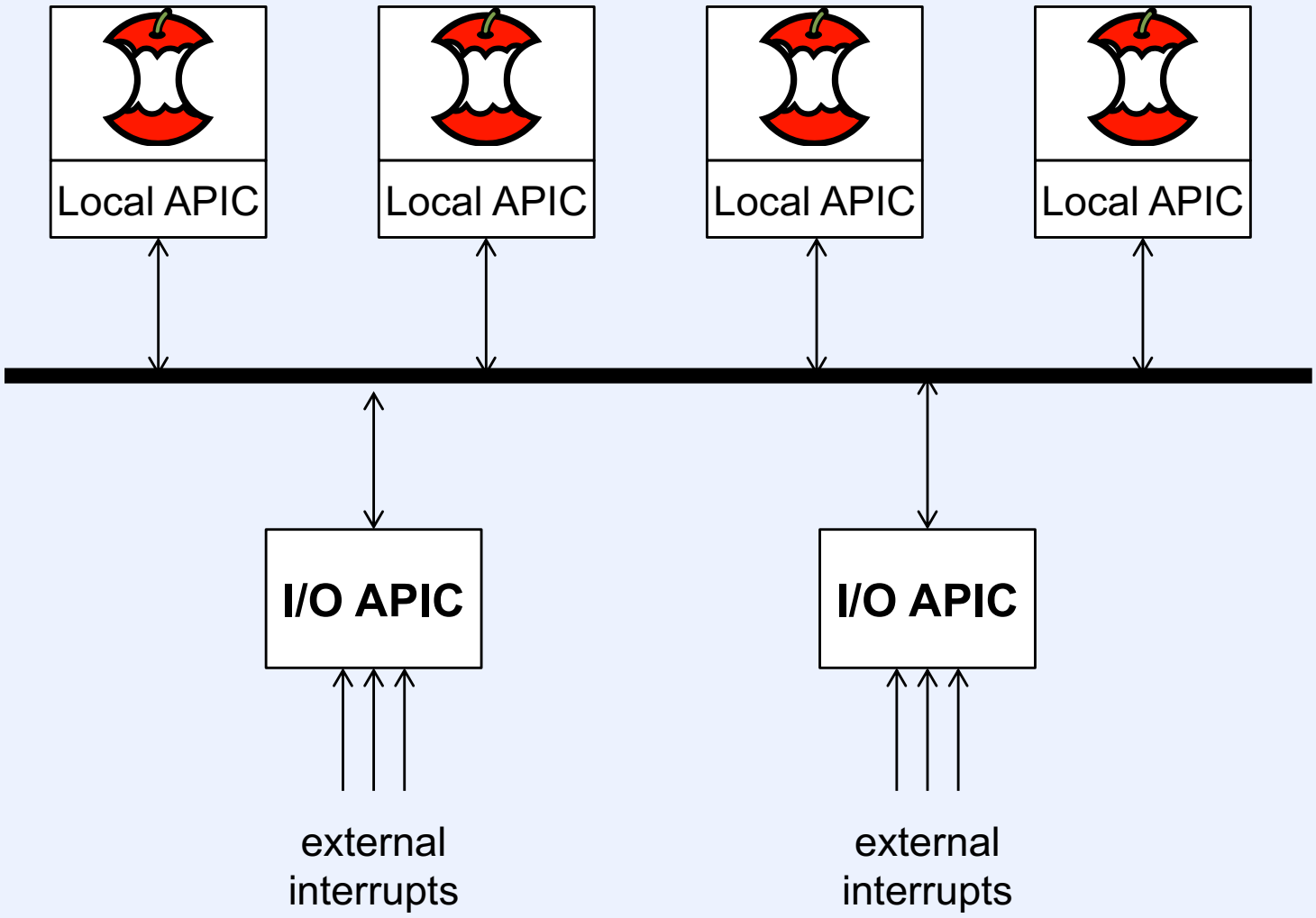


Current thread's
user stack



Current thread's
kernel stack

x86 Interrupt Architecture



Dealing with Interrupts

- Interrupt comes from external source
- Must execute code to handle it
- Which stack?
  - use current (kernel) stack

    or
  - use separate stack

Use the Current (Kernel) Stack

- Borrowed from the current thread
  - requires all threads to have sufficiently large kernel stacks
- Interrupted thread may not concurrently execute
  - would corrupt interrupt handler's stack frames
- Interrupt handler should not block
  - may be waiting on interrupted thread
    - deadlock
  - threads must mask interrupts to protect data structures shared with interrupt handlers

Hierarchical Interrupt Masking

- Interrupts assigned priorities 1 − n
  - current interrupt priority level *i*
    - interrupts with priorities 1 − *i* masked
  - IPL set to i
    - when interrupt of priority *i* is handled
    - when IPL set explicitly to *i*
- Raising the IPL
  - protects data structures
    - good!
  - delays response to lower priority interrupts
    - bad!

Separate Interrupt Stack

- **Single stack used strictly by interrupt handlers**
  - **hardware saves current register state on interrupt stack**
  - **no interrupt-handling space required for kernel stacks**
    - **all can be smaller**
  - **in principle, interrupted thread may continue to execute**
    - **won't corrupt interrupt handler's frames**

Quiz 1

Is interrupt-masking still required?

a) yes
b) no

(Non-Quiz) Questions

1) Is interrupt masking still required? *Done*
2) May interrupt handler block?
3) Does it work on multiprocessors?

Answers

1) Masking is still required
   - if nothing else, to protect data structures shared by multiple interrupt handlers
2) Interrupt handlers should not block
   - would have to block with raised IPL
   - results in lengthy time with interrupts masked
     - delayed response
3) Yes, but each processor has its own interrupt stack

Interrupt Threads

- Give each interrupt instance its own stack
  - handlers effectively execute as separate threads
  - interrupted thread continues to run
  - but interrupts remain masked while interrupt thread is processing interrupt

Effect of Interrupts

- Normally don't directly affect current thread
  - thread is interrupted
  - interrupt is dealt with
  - thread is resumed
- I/O-completion interrupts
  - may result in waking up higher-priority thread
- Clock interrupts
  - may trigger end of time slice

Synchronization and Interrupts

- Non-preemptive kernels
  - threads running in privileged mode yield the processor only voluntarily
  - involuntary thread switches happen only to threads in user mode
    - end of time slice
    - higher-priority thread is made runnable
  - inter-thread synchronization is easy
- Preemptive kernels
  - threads running in privileged mode may be forced to yield the processor
  - inter-thread sync is not easy

Non-Preemptive Kernel Sync.

```
int X = 0;
```

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

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

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
  }
  …
}
```

Improved Disk I/O

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

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);  // IPL == 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);
}
```

Preemptive Kernels on MP

- What's different?
- A thread accesses a shared data structure:
  1. it might be *interrupted* by an interrupt handler (running on its processor) that accesses the same data structure
  2. *another thread* running on another processor might access the same data structure
  3. it might be forced to *give up its processor* to another thread, either because its time slice has expired or it has been preempted by a higher-priority thread
  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);
    …
}
```

Solution …

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

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

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

Quiz 2

We have a single-core system with a preemptible kernel. We're concerned about data structure *X*, which is accessed by kernel threads as well as by an interrupt handler.

a) It's sufficient for threads to mask interrupts while accessing *X*

b) In addition, threads must lock (blocking) mutexes before accessing *X*

c) Instead of b, threads must lock spinlocks before accessing *X*

d) In addition to c, the interrupt handler must lock a spinlock before accessing *X*

Deferred Work

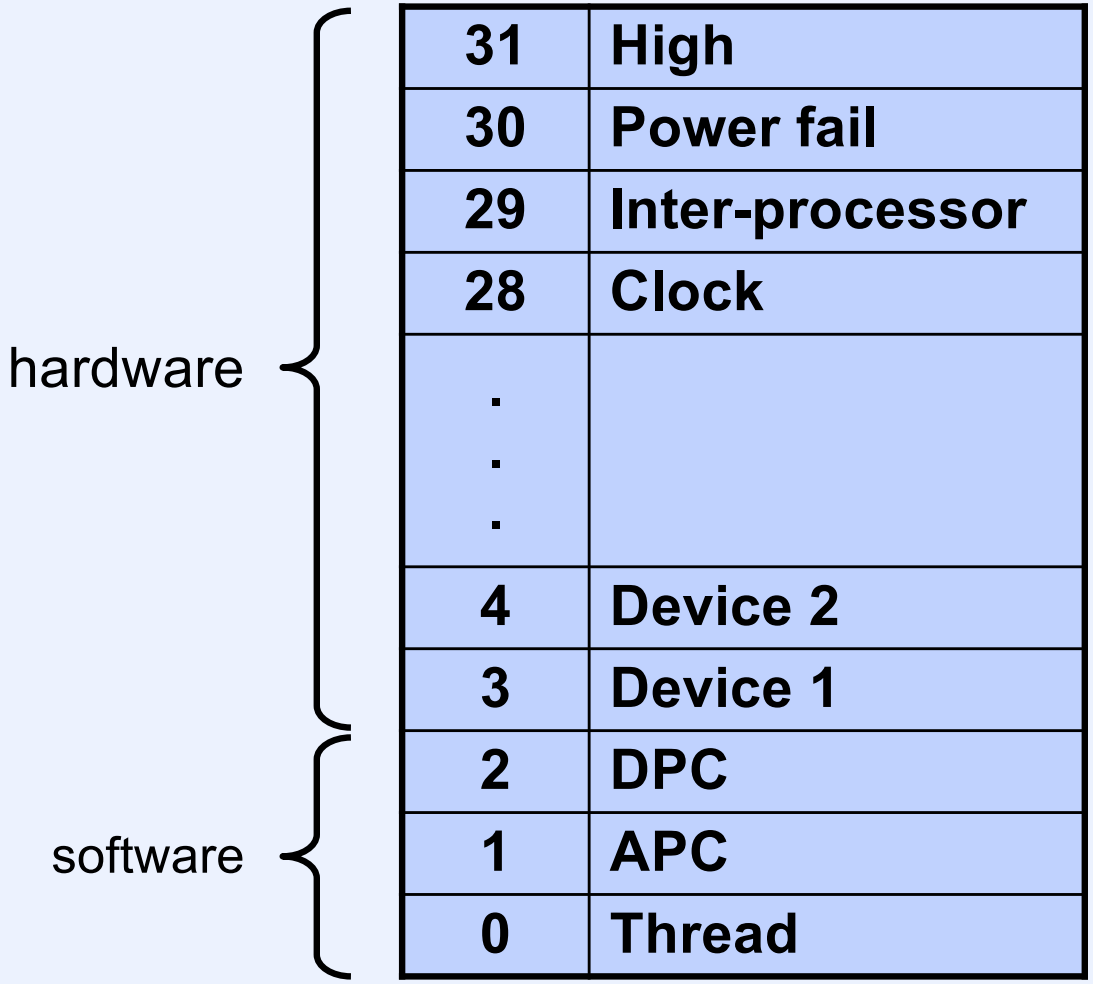
- Interrupt handlers run with interrupts masked
  - both when executed in interrupt context or thread context
  - may interfere with handling of other interrupts
- Solution
  - do minimal work now
  - do rest later without interrupts masked

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
  …
  EnQueue(WorkQueue, MoreWork);
}
```

Windows Interrupt Priority Levels



Deferred Procedure Calls

```
void InterruptHandler( ) {
  // deal with interrupt
  …
  QueueDPC(MoreWork, arg);
  /* enqueues MoreWork on
     the DPC queue and
     requests a DPC
     interrupt
  */
}
```

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

Software Interrupt Threads

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

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

Preemption: User-Level Only

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

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

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

Preemption: Full

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  …
  if (TimeSliceOver)
    QueueDPC(Reschedule);
}
```

Directed Processing

- Signals: Unix
  - perform given action in context of a particular thread in user mode
- APC: Windows asynchronous procedure calls
  - roughly same thing, but also may be done in kernel mode

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 can't 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