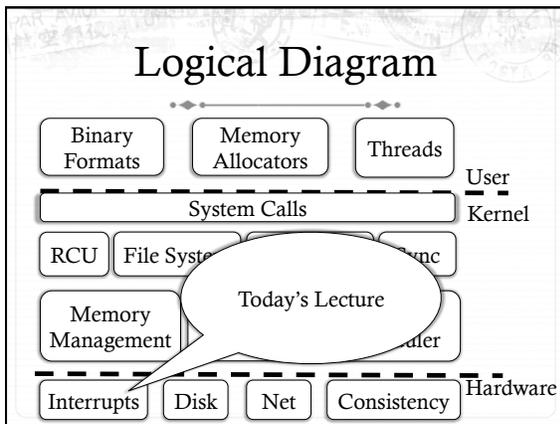


# Interrupts and System Calls

Don Porter  
CSE 506

- ## Housekeeping
- ✦ Welcome TA Amit Arya – Office Hours posted
  - ✦ Next Thursday's class has a reading assignment
  - ✦ Lab 1 due Friday
  - ✦ All students should have VMs at this point
    - ✦ Email Don if you don't have one
  - ✦ Private git repositories should be set-up



## Background: Control Flow

```

pc // x = 2, y = true  void printf(va_args)
    if (y) {          {
                        2 /= x;           //...
                        printf(x);       }
    } //...
    
```

Regular control flow: branches and calls  
(logically follows source code)

## Background: Control Flow

```

pc // x = 0, y = true  void handle_divzero()
    if (y) {          {
        Divide by zero!
        Program can't make
        progress!
        2 /= x;       x = 2;
        printf(x);   }
    } //...
    
```

Irregular control flow: exceptions, system calls, etc.

- ## Lecture goal
- ✦ Understand the hardware tools available for irregular control flow.
    - ✦ I.e., things other than a branch in a running program
  - ✦ Building blocks for context switching, device management, etc.

## Two types of interrupts

- ✦ Synchronous: will happen every time an instruction executes (with a given program state)
  - ✦ Divide by zero
  - ✦ System call
  - ✦ Bad pointer dereference
- ✦ Asynchronous: caused by an external event
  - ✦ Usually device I/O
  - ✦ Timer ticks (well, clocks can be considered a device)

## Intel nomenclature

- ✦ Interrupt – only refers to asynchronous interrupts
- ✦ Exception – synchronous control transfer

✦ Note: from the programmer’s perspective, these are handled with the same abstractions

## Lecture outline

- ✦ Overview
- ✦ How interrupts work in hardware
- ✦ How interrupt handlers work in software
- ✦ How system calls work
- ✦ New system call hardware on x86

## Interrupt overview

- ✦ Each interrupt or exception includes a number indicating its type
- ✦ E.g., 14 is a page fault, 3 is a debug breakpoint
- ✦ This number is the index into an interrupt table

## x86 interrupt table

The diagram shows a horizontal bar representing the interrupt table with indices 0, 31, 47, and 255 marked. A bracket under indices 0-31 is labeled 'Reserved for the CPU'. A bracket under indices 32-47 is labeled 'Device IRQs'. A bracket under indices 48-127 is labeled 'Software Configurable', with callouts indicating '48 = JOS System Call' and '128 = Linux System Call'. A bracket under indices 128-255 is also labeled 'Software Configurable'.

## x86 interrupt overview

- ✦ Each type of interrupt is assigned an index from 0–255.
- ✦ 0–31 are for processor interrupts; generally fixed by Intel
  - ✦ E.g., 14 is always for page faults
- ✦ 32–255 are software configured
  - ✦ 32–47 are for device interrupts (IRQs) in JOS
  - ✦ Most device’s IRQ line can be configured
  - ✦ Look up APICs for more info (Ch 4 of Bovet and Cesati)
  - ✦ 0x80 issues system call in Linux (more on this later)

## Software interrupts

- ✦ The `int <num>` instruction allows software to raise an interrupt
  - ✦ `0x80` is just a Linux convention. JOS uses `0x30`.
- ✦ There are a lot of spare indices
  - ✦ You could have multiple system call tables for different purposes or types of processes!
    - ✦ Windows does: one for the kernel and one for win32k

## Software interrupts, cont

- ✦ OS sets ring level required to raise an interrupt
  - ✦ Generally, user programs can't issue an `int 14` (page fault manually)
  - ✦ An unauthorized `int` instruction causes a general protection fault
    - ✦ Interrupt 13

## What happens (generally):

- ✦ Control jumps to the kernel
  - ✦ At a prescribed address (the interrupt handler)
- ✦ The register state of the program is dumped on the kernel's stack
  - ✦ Sometimes, extra info is loaded into CPU registers
    - ✦ E.g., page faults store the address that caused the fault in the `cr2` register
- ✦ Kernel code runs and handles the interrupt
- ✦ When handler completes, resume program (see `iret` instr.)

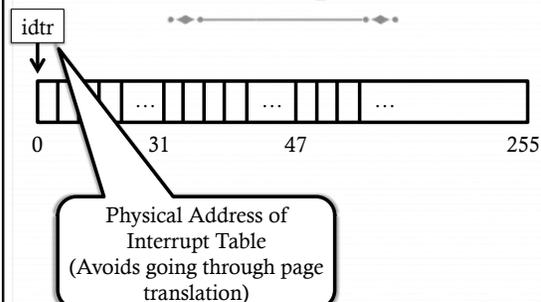
## How it works (HW)

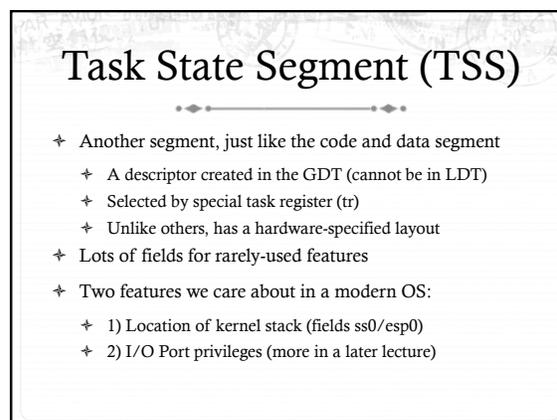
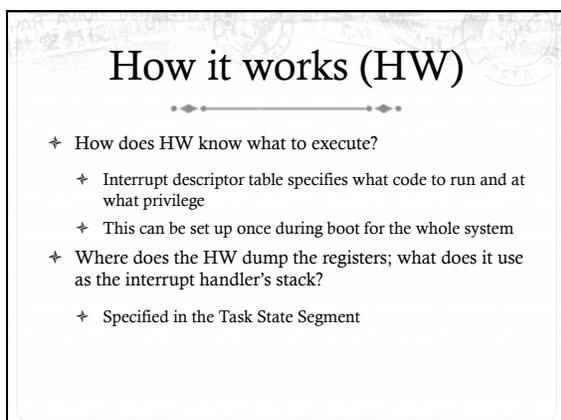
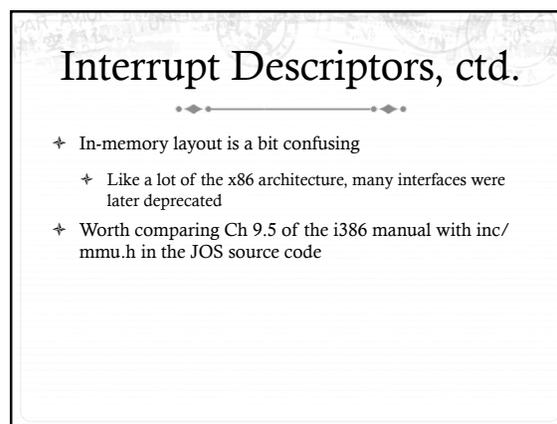
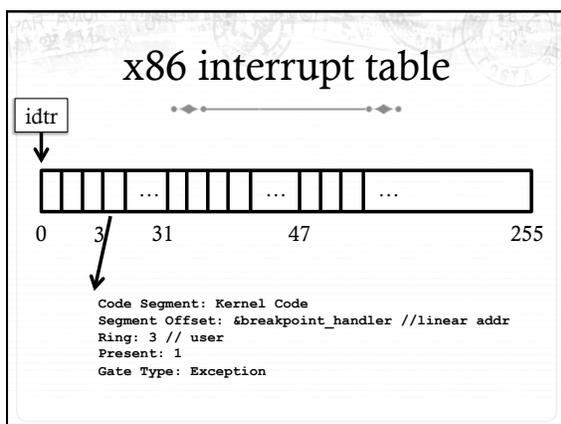
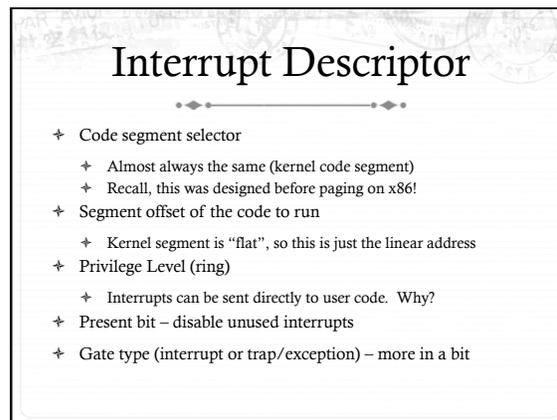
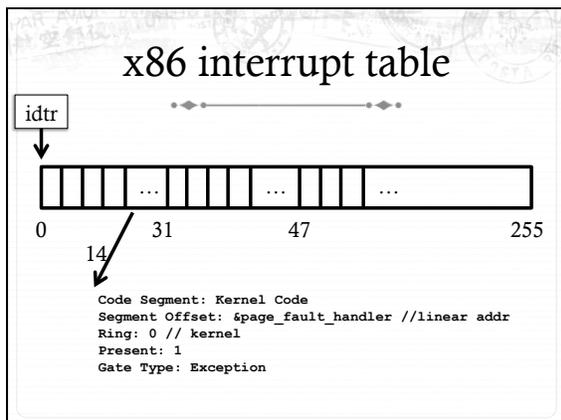
- ✦ How does HW know what to execute?
- ✦ Where does the HW dump the registers; what does it use as the interrupt handler's stack?

## How is this configured?

- ✦ Kernel creates an array of Interrupt descriptors in memory, called Interrupt Descriptor Table, or IDT
  - ✦ Can be anywhere in physical memory
  - ✦ Pointed to by special register (`idtr`)
    - ✦ c.f., segment registers and `gdt` and `ldtr`
- ✦ Entry 0 configures interrupt 0, and so on

## x86 interrupt table





## TSS, cont.

- ✦ Simple model: specify a TSS for each process
- ✦ Optimization (JOS):
  - ✦ Our kernel is pretty simple (uniprocessor only)
  - ✦ Why not just share one TSS and kernel stack per-process?
- ✦ Linux generalization:
  - ✦ One TSS per CPU
  - ✦ Modify TSS fields as part of context switching

## Summary

- ✦ Most interrupt handling hardware state set during boot
- ✦ Each interrupt has an IDT entry specifying:
  - ✦ What code to execute, privilege level to raise the interrupt
- ✦ Stack to use specified in the TSS

## Comment

- ✦ Again, segmentation rears its head
- ✦ You can't program OS-level code on x86 without getting your hands dirty with it
- ✦ Helps to know which features are important when reading the manuals

## Lecture outline

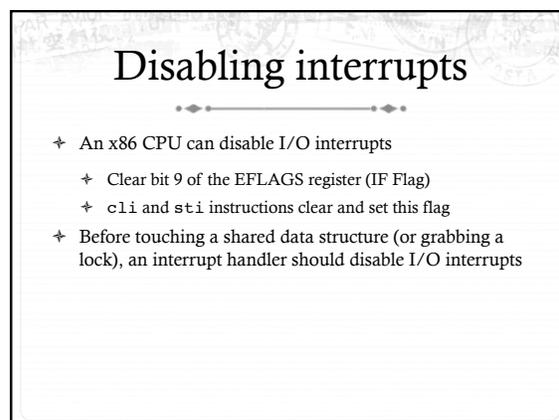
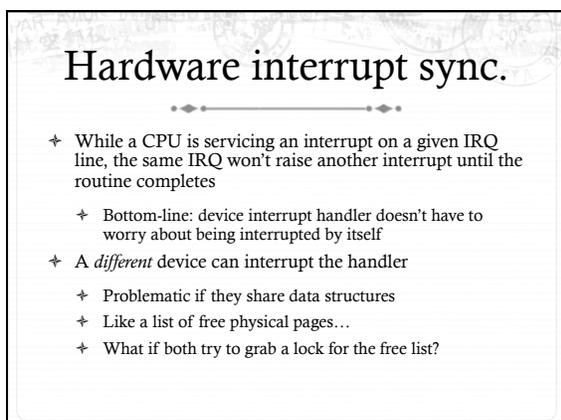
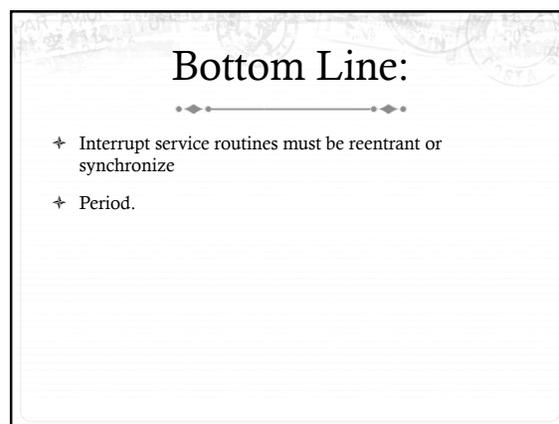
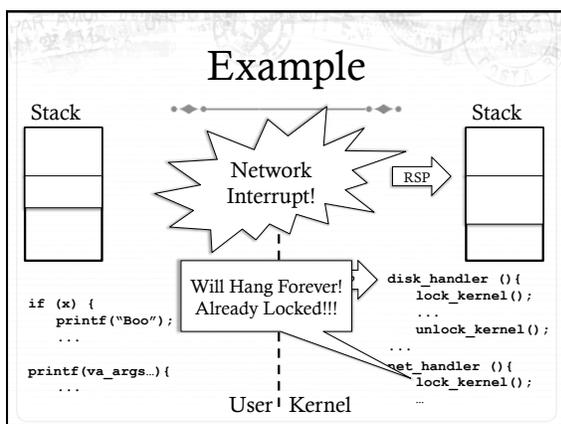
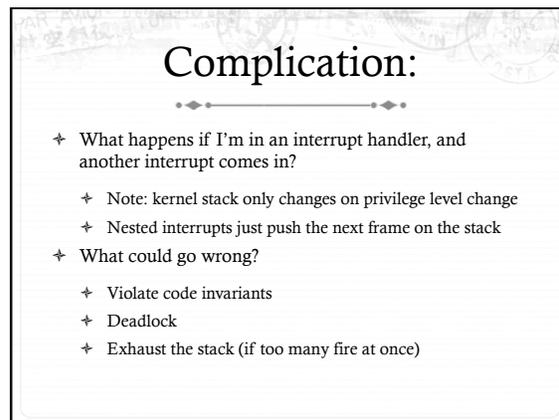
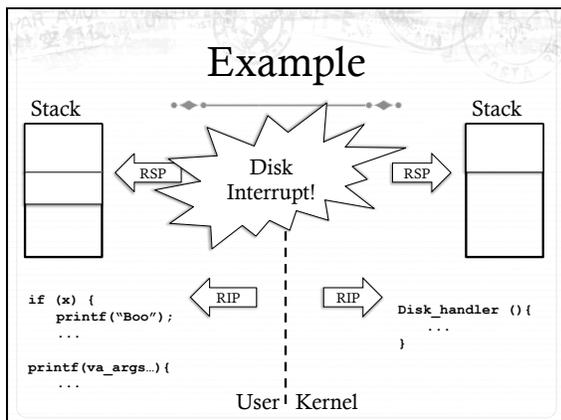
- ✦ Overview
- ✦ How interrupts work in hardware
- ✦ **How interrupt handlers work in software**
- ✦ How system calls work
- ✦ New system call hardware on x86

## High-level goal

- ✦ Respond to some event, return control to the appropriate process
- ✦ What to do on:
  - ✦ Network packet arrives
  - ✦ Disk read completion
  - ✦ Divide by zero
  - ✦ System call

## Interrupt Handlers

- ✦ Just plain old kernel code



## Gate types

- ✦ Recall: an IDT entry can be an interrupt or an exception gate
- ✦ Difference?
  - ✦ An interrupt gate automatically disables all other interrupts (i.e., clears and sets IF on enter/exit)
  - ✦ An exception gate doesn't
- ✦ This is just a programmer convenience: you could do the same thing in software

## Exceptions

- ✦ You can't mask exceptions
  - ✦ Why not?
    - ✦ Can't make progress after a divide-by-zero
  - ✦ Double and Triple faults detect faults in the kernel
- ✦ Do exception handlers need to be reentrant?
  - ✦ Not if your kernel has no bugs (or system calls in itself)
  - ✦ In certain cases, Linux allows nested page faults
    - ✦ E.g., to detect errors copying user-provided buffers

## Summary

- ✦ Interrupt handlers need to synchronize, both with locks (multi-processor) and by disabling interrupts (same CPU)
- ✦ Exception handlers can't be masked
  - ✦ Nested exceptions generally avoided

## Lecture outline

- ✦ Overview
- ✦ How interrupts work in hardware
- ✦ How interrupt handlers work in software
- ✦ **How system calls work**
- ✦ New system call hardware on x86

## System call "interrupt"

- ✦ Originally, system calls issued using `int` instruction
- ✦ Dispatch routine was just an interrupt handler
- ✦ Like interrupts, system calls are arranged in a table
  - ✦ See `arch/x86/kernel/syscall_table*.S` in Linux source
- ✦ Program selects the one it wants by placing index in `eax` register
  - ✦ Arguments go in the other registers by calling convention
  - ✦ Return value goes in `eax`

## Lecture outline

- ✦ Overview
- ✦ How interrupts work in hardware
- ✦ How interrupt handlers work in software
- ✦ How system calls work
- ✦ **New system call hardware on x86**

## Around P4 era...

- ✦ Processors got very deeply pipelined
  - ✦ Pipeline stalls/flushes became very expensive
  - ✦ Cache misses can cause pipeline stalls
- ✦ System calls took twice as long from P3 to P4
  - ✦ Why?
    - ✦ IDT entry may not be in the cache
    - ✦ Different permissions constrain instruction reordering

## Idea

- ✦ What if we cache the IDT entry for a system call in a special CPU register?
  - ✦ No more cache misses for the IDT!
  - ✦ Maybe we can also do more optimizations
- ✦ Assumption: system calls are frequent enough to be worth the transistor budget to implement this
  - ✦ What else could you do with extra transistors that helps performance?

## Intel: sysenter/sysexit

- ✦ These instructions use MSRs (machine specific registers) to store:
  - ✦ Syscall entry point and code segment
  - ✦ Kernel stack
  - ✦ Syscall return address
- ✦ Implication: system calls must be issued from a few kernel-approved addresses
  - ✦ i.e., in libc

## Pros and cons of fixed return point

- ✦ Pros:
  - ✦ Indeed faster than `int` instruction
  - ✦ Security arguments:
    - ✦ Easier to sandbox a program (prevent illegal system calls)
    - ✦ Limits ability of a program to issue errant system calls
- ✦ Cons: Programmer inconvenience
  - ✦ Can't just drop an `int 0x80` in my program anymore
  - ✦ Tighter contract between program and kernel
  - ✦ Also, not all x86 CPUs have this instruction

## More on compatibility

- ✦ Not all CPUs have `sysenter`
- ✦ We don't want every program to have to encode knowledge about every x86 CPU model
- ✦ And we don't want to break backwards-compatibility

## Linus's "disgusting" solution

- ✦ Kernel can support both `sysenter` and `int` (for legacy programs)
- ✦ Kernel figures out what CPU supports (since it has to anyway)
- ✦ Creates a page with the optimal system call instruction (and a standard function call preamble and epilogue)
  - ✦ Always mapped at a fixed address in programs
  - ✦ Replace `int 0x80` with a `call <addr>`

## vdso

- ✦ This page is called the Virtual Dynamic Shared Object (vdso)
- ✦ Libc and other programs reserve this address in their link tables
- ✦ Kernel is responsible for mapping it in during `exec`
- ✦ Solves part of the compatibility problem

## AMD: syscall/sysret

- ✦ Same basic idea as `sysenter/sysexit`, but without a fixed return point
  - ✦ Programmers suffered with the fixed return point for the performance win, but didn't like it
- ✦ More of a drop-in replacement for `int 0x80`
  - ✦ Trade a bit of the performance win for a big convenience win
- ✦ Everyone loved it and adopted it wholesale
  - ✦ Even Intel!

## Aftermath (pt 1)

- ✦ If every recent x86 CPU has `syscall`, why bother with `sysenter`?
  - ✦ Good question. Most don't!
- ✦ All 64-bit CPUs have `syscall`
  - ✦ Only really need vdso for 32-bit programs

## Aftermath (pt. 2)

- ✦ `Getpid()` on my desktop machine (recent AMD 6-core):
  - ✦ `int 80`: 371 cycles
  - ✦ `syscall`: 231 cycles
- ✦ So system calls are definitely faster as a result!

## In JOS

- ✦ You will use the `int` instruction to implement system calls
- ✦ There is a challenge problem in lab 3 (i.e., extra credit) to use `sysenter/sysexit`
  - ✦ Note that there are some more details about register saving to deal with
  - ✦ `syscall/sysret` is a bit too trivial for extra credit
    - ✦ But still cool if you get it working!

## Summary

- ✦ Interrupt handlers are specified in the IDT
- ✦ Understand when nested interrupts can happen
  - ✦ And how to prevent them when unsafe
- ✦ Understand optimized system call instructions
  - ✦ Be able to explain vdso, `syscall` vs. `sysenter` vs. `int 80`