Interrupts and System Calls

Don Porter
CSE 506
Housekeeping

• Welcome Tas Amogh Akshintala and Yizheng Jiao (1/2 time) – Office Hours posted
• Next Thursday’s class has a reading assignment
• Lab 1 due Monday 9/8
• All students should have VMs and private repos soon
  – Email Don if you don’t have one by tonight
  – Unless you just got in the class since Wed night (sigh)
Today's Lecture
Background: Control Flow

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

void printf(va_args)
{
    //...
}

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

```c
// x = 0 ... 
true
if (y) {
  2 /= x;
  printf(x);
} //...

void handle_divzero() {
  x = 2;
}
```

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

Device IRQs

- 0 to 31: Reserved for the CPU
- 32 to 47: Software Configurable
- 48 = JOS System Call
- 128 = Linux System Call

Interrupts from 0 to 255 can be assigned to devices.
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 memory
  – Pointed to by special register (idtr)
    • c.f., segment registers and gdtr and ldtr

• Entry 0 configures interrupt 0, and so on
x86 interrupt table

idtr

Linear Address of Interrupt Table
x86 interrupt table

idtr

Code Segment: Kernel Code
Segment Offset: &page_fault_handler //linear addr
Ring: 0 // kernel
Present: 1
Gate Type: Exception
Interrupt Descriptor

• Code segment selector
  – Almost always the same (kernel code segment)
  – Recall, this was designed before paging on x86!

• Segment offset of the code to run
  – Kernel segment is “flat”, so this is just the linear address

• Privilege Level (ring)
  – Ring that can raise this interrupt with an `int` instruction

• Present bit – disable unused interrupts

• Gate type (interrupt or trap/exception) – more in a bit
x86 interrupt table

Code Segment: Kernel Code
Segment Offset: &breakpoint_handler //linear addr
Ring: 3 // user
Present: 1
Gate Type: Exception
Interrupt Descriptors, ctd.

• In-memory layout is a bit confusing
  – Like a lot of the x86 architecture, many interfaces were later deprecated

• Worth comparing Ch 9.5 of the i386 manual with inc/mmuc.h in the JOS source code
How it works (HW)

• How does HW know what to execute?
  – Interrupt descriptor table specifies what code to run
    • And at what privilege (via code segment)
      – This can be set up once during boot for the whole system

• Where does the HW dump the registers; what does it use as the interrupt handler’s stack?
  – Specified in the Task State Segment
Task State Segment (TSS)

- Another segment, just like the code and data segment
  - A descriptor created in the GDT (cannot be in LDT)
  - Selected by special task register (tr)
  - Unlike others, has a hardware-specified layout

- Lots of fields for rarely-used features

- Two features we care about in a modern OS:
  - 1) Location of kernel stack (fields ss0/esp0)
  - 2) I/O Port privileges (more in a later lecture)
TSS, cont.

• Simple model: specify a TSS for each process
  – Note: Only $2^{13}$ entries in the GDT

• 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
Example

if (x) {
    printf("Boo");
    ...
}
printf(va_args...)

Disk_handler (){
    ...
}

Disk Interrupt!
Complication:

• What happens if I’m in an interrupt handler, and another interrupt comes in?
  – Note: kernel stack only changes on privilege level change
  – Nested interrupts just push the next frame on the stack

• What could go wrong?
  – Violate code invariants
  – Deadlock
  – Exhaust the stack (if too many fire at once)
Example

Stack

if (x) {
    printf("Boo\n"
...  
    printf(va_args...)

Network Interrupt!

Will Hang Forever! Already Locked!!!

Stack

User \ Kernel

disk_handler (){
    lock_kernel();
    ...
    unlock_kernel()
 }

net_handler (){
    lock_kernel();
    ...
}
Bottom Line:

• Interrupt service routines must be reentrant or synchronize
• Period.
Hardware interrupt sync.

• While a CPU is servicing an interrupt on a given IRQ line, the same IRQ won’t raise another interrupt until the routine completes
  – Bottom-line: device interrupt handler doesn’t have to worry about being interrupted by itself

• A different device can interrupt the handler
  – Problematic if they share data structures
  – Like a list of free physical pages...
  – What if both try to grab a lock for the free list?
Disabling interrupts

• An x86 CPU can disable I/O interrupts
  – Clear bit 9 of the EFLAGS register (IF Flag)
  – `cli` and `sti` instructions clear and set this flag

• Before touching a shared data structure (or grabbing a lock), an interrupt handler should disable I/O interrupts
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?
AMD: syscall/sysret

• These instructions use MSRs (machine specific registers) to store:
  – Syscall entry point and code segment
  – Kernel stack
• A drop-in replacement for int 0x80
• Everyone loved it and adopted it wholesale
  – Even Intel!
Aftermath

• 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 systenter/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. int 80