Interrupts and System Calls

Don Porter
CSE 306

Lecture goal
• Understand how system calls work
  – As well as how exceptions (e.g., divide by zero) work
• 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.

Background: Control Flow

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

Irregular control flow: exceptions, system calls, 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)
Asynchronous Interrupt Example

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

Disk_handler()

User * Kernel

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

```
0  31  47  255
```

Device IRQs

Reserved for the CPU

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.

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`

Hardware interrupts (low level):
- 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 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

- `0  31  47  ...
  255`

x86 interrupt table

- `idtr`
- Linear Address of Interrupt Table

- `0  31  47  ...
  255`

- `Code Segment: Kernel Code`
- `Segment Offset: page_fault_handler //linear addr`
- `Ring: 0 // kernel`
- `Present: 1`
- `Gate Type: Exception`
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

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
  – Sort of like exception handlers in Java
  – But separated from the control flow of the program
• The IDT stores a pointer to the right handler routine

What is a system call?

• A function provided to applications by the OS kernel
  – Generally to use a hardware abstraction (file, socket)
  – Or OS-provided software abstraction (IPC, scheduling)
• Why not put these directly in the application?
  – Protection of the OS/hardware from buggy/malicious programs
  – Applications are not allowed to directly interact with hardware, or access kernel data structures
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

How many system calls?
- Linux exports about 350 system calls
- Windows exports about 400 system calls for core APIs, and another 800 for GUI methods

But why use interrupts?
- Also protection
- Forces applications to call well-defined “public” functions
  - Rather than calling arbitrary internal kernel functions
- Example:
  public foo()
  {
    if (!permission_ok()) return -EPERM;
    return _foo(); // no permission check
  }

Calling _foo() directly would circumvent permission check

Summary
- System calls are the “public” OS APIs
- Kernel leverages interrupts to restrict applications to specific functions
- Lab 1 hint: How to issue a Linux system call?
  - int $0x80, with system call number in eax register

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!

Summary

• Interrupt handlers are specified in the IDT
• Understand how system calls are executed
  – Why interrupts?
  – Why special system call instructions?