Interrupts and System Calls

Don Porter
CSE 306

Background: Control Flow

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

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

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)

Asynchronous Example

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

Disk Interrupt!
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

- 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 often for device interrupts (IRQs)
- 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.
- You could change it to use 0x81!
- 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 gdtr and ldtr
  - Entry 0 configures interrupt 0, and so on

x86 interrupt table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Interrupt</th>
<th>Code Segment</th>
<th>Segment Offset</th>
<th>Present</th>
<th>Gate Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-15</td>
<td>Kernel Code</td>
<td>&amp;page_fault_handler</td>
<td>1</td>
<td>Exception</td>
</tr>
<tr>
<td>14</td>
<td>16-255</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
- Interrupts can be sent directly to user code. Why?
- Present bit – disable unused interrupts
- Gate type (interrupt or trap/exception) – more in a bit

x86 interrupt table

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>31</th>
<th>...</th>
<th>47</th>
<th>...</th>
<th>255</th>
</tr>
</thead>
</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

How it works (HW)

- How does HW know what to execute?
- Interrupt descriptor table specifies what code to run and at what privilege
- 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 magic control block
- Pointed to by special task register (tr)
- Actually stored in the segment table (more on segmentation later)
- 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
- Optimization (for a simple uniprocessor OS):
  1) Why not just share one TSS and kernel stack per-process?
- Linux generalization:
  1) One TSS per CPU
  2) 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

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 handlers

- Just plain old code in the kernel
- 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

Lecture outline

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

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
  }
  ```

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/sysreturn

- These instructions use MSRs (machine specific registers) to store:
  - Syscall entry point and code segment
  - Kernel stack
  - Drop-in replacement for int 0x80
  - Longer saga with Intel variant

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 Lab 1

- You will use the int instruction to implement system calls
- You are welcome to use syscall if you prefer

Summary

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