Device Programming

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)
Talking to Devices

• Device interface consists of *registers* and *memories*
  – plus interrupts for some (most) devices
  – Ex. of registers: command, control and status
  – Ex. of memory: frame buffer in video card

• How to access device register and memory?

• Two ways:
  – Port-mapped I/O (only x86 these days)
  – Memory-mapped I/O

• Many devices use both at the same time
  – Port-mapped for registers
  – Memory-mapped for memory
Port-Mapped I/O

• Initial x86 model: separate memory and I/O space
  – Memory uses virtual addresses
  – Devices accessed via ports

• A port is just an address (like memory), but in a different space
  – Port 0x1000 is not the same as address 0x1000

• Goal: not wasting *limited* memory space on I/O
  – Memory space only used for RAM

• Can map both device registers and memory to ports
Programming Ports

• Different instructions to access
  – inb, inw, outl, etc.

• Unlike RAM, writing to a port has side-effects
  – “Launch” opcode to /dev/missiles
  – So can reading!
  – Memory can safely duplicate operations/cache results

• Idiosyncrasy: composition doesn’t necessarily work
  – outw 0x1010 <port> != outb 0x10 <port>
    outb 0x10 <port+1>
Memory-Mapped I/O

• Map devices onto regions of physical memory

• Hardware redirects accesses away from RAM
  – Points those addresses at devices
  – A bummer if you “lose” some RAM
    • Map devices to regions where there is no RAM
    • Not always possible – recall the ISA hole (640 KB-1 MB) from Lab 2

• Win: Cast interface regions to a struct types
  – Write updates to different areas using high-level languages

• Subject to same side-effect caveats as ports
Programming Mem-Mapped IO

• A memory-mapped device is accessed by normal mem. ops

• But, how does compiler know about I/O?
  – Which regions have side-effects and other constraints?
    • It doesn’t: programmer must specify!
Problem with Optimizations

• Recall: Common optimizations (compiler and CPU)
  – Compilers keep values in registers, eliminate redundant operations, etc.
  – CPUs have caches
  – CPUs do out-of-order execution and re-order instructions

• When reading/writing a device, it should happen immediately
  – Do not keep it in a register
  – Do not re-order it
  – Also, should not keep it in processor’s cache

• CPU *and* compiler optimizations must be disabled
volatile Keyword

• volatile variable cannot be bound to a register
  – Writes must go directly to memory/cache
  – Reads must always come from memory/cache

• volatile code blocks cannot be re-ordered
  – Must be executed precisely at this point in program
  – e.g., inline assembly

• __volatile__ means I really mean it!
Fence Operations

• Also known as Memory Barriers

• `volatile` does not force the CPU to execute instructions in order

```c
Write to <device register 1>
mb(); // fence
Read from <device register 2>
```

• Use a `fence` to force in-order execution
  – Linux example: `mb()`
  – Also used to enforce ordering between memory operations in multi-processor systems
Dealing with Caches

• Processor may cache memory locations

• Often, memory-mapped I/O should not be cached

• Solution: Mark ranges of memory used for I/O as non-cacheable
Configuration

• Where does all of this come from?
  – Who sets up port mapping and I/O memory mappings?
  – Who maps device interrupts onto IRQ lines?

• Generally, the BIOS
  – Sometimes constrained by device limitations
  – Older devices hard-coded port addresses and IRQs
  – Older devices only have 16-bit addresses
    • Can only access lower memory addresses
Buses

• Buses are “plumbing” between major components

• There is a bus between RAM and CPUs

• There is often another bus between devices
  – Buses tend to have standard specifications
    • Ex: PCI, ISA, AGP
PCI

• PCI (memory and I/O ports) is configurable
  – Generally by the BIOS
    • Mainly at boot time
  – But could be remapped by the kernel

• Configuration space
  – A new space in addition to port space and memory space
  – 256 bytes per device (4k per device in PCIe)
  – Standard layout per device, including unique ID
  – Big win: standard way to figure out hardware
PCI Configuration Layout

Figure 12-2. The standardized PCI configuration registers

From *Linux Device Drivers, 3rd Ed*
PCI Overview

- Most desktop systems have 2+ PCI buses
  - Joined by a bridge device
  - Forms a tree structure (bridges have children)
PCI Layout

Figure 12-1. Layout of a typical PCI system

From Linux Device Drivers, 3rd Ed
PCI Addressing

• Each peripheral listed by:
  – Bus Number (up to 256 per domain or host)
    • A large system can have multiple domains
  – Device Number (32 per bus)
  – Function Number (8 per device)
    • Function, as in type of device
      • Audio function, video function, storage function, ...

• Devices addressed by a 16-bit number

• Linux command `lspci` shows all the PCI devices + lots of information on them
PCI Interrupts

• Each PCI slot has 4 interrupt pins

• Device does not worry about mapping to IRQ lines
  – APIC or other intermediate chip does this mapping

• Bonus: flexibility!
  – Sharing limited IRQ lines is a hassle. Why?
    • Trap handler must de-multiplex interrupts
  – Being able to “load balance” the IRQs is useful
Direct Memory Access (DMA)

• Simple read/write model bounces all I/O through the CPU
  – Fine for small data, totally awful for huge data

• Idea: tell device where you want data to go (or come from)
  – Let device do data transfers to/from memory
    • No CPU intervention
    – Interrupt CPU on I/O completion

• DMA buffers must be in physical memory
  – Like page tables and IDTs
Ring Buffers

- Many devices pre-allocate a “ring” of buffers
  - Think network card

- Device writes into ring; CPU reads behind

- If ring is well-sized to the load:
  - No dynamic buffer allocation
  - No stalls

- Trade-off between device stalls (or dropped packets) and memory overheads
IOMMU

• It is a pain to allocate physically contiguous regions

• Idea: “virtual addresses” for devices
  – We can take random physical pages and make them look contiguous to the device
  – Called “Bus address” for clarity

• New to the x86 (called VT-d)
  – Until very recently, x86 kernels just suffered

• But why does x86 suddenly care about IOMMUs?
  – Next slide
IOMMU and Virtual Machines

• Scenario: system with 4 NICs, 4 VMs
  – Want to give each VM its own NIC
  – VM1 can write to a NIC’s control register and tell it to DMA to VM2’s memory – BAD !!!

• Without IOMMU: Hypervisor must mediate all network traffic

• With IOMMU: Each VM can have a different virtual bus address space
  – Looks like a single NIC; can only issue DMAs for its own memory (not other VM’s memory)
  – No Hypervisor mediation needed!
Recall: Handling Interrupts

• Interrupts disabled while in interrupt handler
  – Need to avoid spending much time in there

• Split interrupt processing into two steps
  – *Top half*: acknowledge interrupt, queue work
  – *Bottom half*: take work from queue and do it