Device Programming

Nima Honarmand
Device Interface Components:

- Device registers
- Device Memory
- DMA buffers
- Interrupt lines
Device Register and Memory

• Device registers: small (2, 4, 8 bytes)
• Device memory: larger sizes

• Don’t think of them as storage: reads and writes have side effects
  • Unless, explicitly specified otherwise
  • E.g., writing to an IDE controller register can start a disk read/write process (as in JOS’ IDE driver)

• Example of device registers: command, control and status registers
• Example of device 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
Accessing Device Register & Memory

• Two methods
  • **PIO**: Programmed I/O (or Port I/O)
    • Only x86 these days
  • **MMIO**: Memory-mapped I/O
    • Determined by device designer (not programmer)

• Some devices may use both at the same time
  • Programmed I/O for device registers
  • Memory-mapped for device memory

• Newer devices just use memory-mapped
  • E.g., PCI and PCIe
Programmed I/O

• Initial x86 model: separate memory and I/O space
  • Memory uses memory addresses
  • Devices accessed via I/O 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 with Ports

• Dedicated instructions to access ports
  • `inb`, `inw`, `outl`, etc.

• Unlike RAM, writing to a port has *side effects*
  • “Launch” opcode to `/dev/missiles`
  • So can reading!
    • Every port read can return a different result
    • Ex: reading disk data in JOS’ IDE driver
  • 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 device memory onto regions of *physical* memory address space

• Hardware redirects accesses away from RAM and to the device
  • 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 memory ops
  • E.g., the \texttt{mov} family in x86

• 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
  • Should not keep it in a processor register
  • Should not re-order it (neither compiler nor CPU)
  • 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** are not re-ordered by the compiler
  • Must be executed precisely at this point in program
  • E.g., inline assembly
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
  • Whether it’s DRAM or MMIO device register or memory

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

• Solution: Mark ranges of memory used for I/O as non-cacheable
  • Basically, disable caching for such memory ranges
Direct Memory Access (DMA)

- Reading/writing through device registers & memories bounces all I/O through the CPU
  - Uses CPU cycles
  - Fine for small data, totally awful for huge data

- Idea:
  - Tell device where you want data to go (or come from) in DRAM
  - Let device do data transfers to/from memory
    - *Direct Memory Access (DMA)*
    - No CPU intervention
    - Let know CPU on completion: interrupt CPU or let CPU poll later

- DMA buffers must be allocated in memory
  - Physical address is passed to the device
  - Like page tables and IDTs
Ring Buffers

• Many devices use pre-allocated “ring” of DMA buffers
  • E.g., network card use TX and RX rings (a.k.a. queues)

• Ring structured like a circular FIFO queue
  • Both ring and buffer allocated in DRAM by driver
  • Device registers for ring base, end, head and tail
    • Head: the first HW-owned (ready-to-consume) DMA buffer
    • Tail: location after the last HW-owned DMA buffer
  • Device advances head pointer to get the next valid buffer
  • Driver advances tail pointer to add a valid buffer

• No dynamic buffer allocation or device stalls if ring is well-sized to the load
  • Trade-off between device stalls (or dropped packets) & memory overheads
Interrupts & Doorbells (1)

• Ring buffers used for both sending and receiving

• **Receive**: device copies data into next empty buffer in the ring and advances head pointer
  - How would driver know about the new buffer?
    - Option 1: driver polls head pointer to see if changed
    - Option 2: Device sends an interrupt
  - How would device know when there is a new empty buffer?
    - When the driver writes to the tail register
    - Sometimes, referred to as *ringing the doorbell*
Interrupts & Doorbells (2)

• **Send**: driver prepares a full buffer & adds it to the ring tail
  • How would device know about the new buffer?
    • When the driver writes to the tail register (again a doorbell)
  • How would driver know there is room for new buffers in the ring?
    • Same options as before: driver polling or device interrupting
Review: 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
Device Configuration
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 have hard-coded port addresses and IRQs
  • Older devices only have 16-bit addresses
    • Can only access lower memory addresses
PCI

• PCI (memory and I/O ports) is configurable
  • Mainly at boot time by the BIOS
  • 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
PCI Tree Layout

Figure 12-1. Layout of a typical PCI system

Source: Linux Device Drivers, 3rd Ed
Software’s View of PCI Tree

• 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: 8 for bus#, 5 for device#, 3 for function#

• 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
  • BIOS and APIC do this mapping
  • Kernel can change this in runtime
    • E.g., to “load balance” the IRQs
Configuring & Enumerating PCI

• At boot time, BIOS configures PCI devices
  • Assigns a physical (MMIO) address to each BAR region for each PCI device
  • Assigns IRQ lines to PCI interrupts
  • Writes the configuration to each device’s config space
  • Kernel can change configuration later

• Kernel uses BIOS routines to enumerate configured devices
  • For each device, kernel reads its config space to identify its MMIO regions and interrupts
  • Maps the MMIO regions (physical addresses) to its virtual address space to be able to access the device
  • Uses vendor and device IDs to find and initialize the appropriate driver for the device
New Stuff: IOMMU and SR-IOV

IOMMU:
• So far, we assumed device can only DMA to memory using physical addresses
  • i.e., no address translation layer for device accesses
• IOMMU provides such a translation layer
  • Same way that MMU translates from CPU-virtual to physical, IOMMU translates from device-virtual to physical

SR-IOV:
• Single-Root IO Virtualization
  • Allows a single PCI device to expose many virtual devices to make kernel-based multiplexing unnecessary
  • Very useful in building high-performance virtual machines

• Will discuss both subjects extensively in virtual machine lectures