CSE 506: Operating Systems

Device Drivers
Talking to Devices

• 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
    • Hardware can be smart enough to remap RAM

• Win: Cast interface regions to a structure
  – Write updates to different areas using high-level languages
  – Still subject to timing, side-effect caveats
Optimizations

• How does compiler (and CPU) know about I/O?
  – Which regions have side-effects and other constraints?
    • It doesn’t: programmer must specify!
Optimizations (2)

- Recall: Common optimizations (compiler and CPU)
  - Out-of-order execution
  - Reorder writes
  - Cache values in registers
- When writing to device, write happens immediately
  - Do not keep it in a register
- CPU and compiler optimizations must be disabled
volatile keyword

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

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

• `__volatile__` means I really mean it!
Compiler barriers

• Inline assembly has a set of clobber registers
  – Hand-written assembly will clobber them
  – Compiler saves values to memory before inline asm
    • No caching of values in these registers

• “memory” clobber says to flush all registers
  – Compiler generates code for all writes to memory
  – No previously-read values can be used without re-read
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 IRQs
  – Older devices may only have a 16-bit chip
    • Can only access lower memory addresses
I/O Ports

• 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)
  – Port 0x1000 is not the same as address 0x1000
  – Different instructions – inb, inw, outl, etc...
More on ports

• A port maps onto input pins/registers on a device
• Unlike memory, writing to a port has side-effects
  – Write “launch” command to /dev/missiles
    • Reading ports can trigger hardware actions too!
  – Memory can safely duplicate operations/cache results
• Idiosyncrasy: composition doesn’t necessarily work
  – outw 0x1010 <port> != outb 0x10 <port>
  \[\text{outb 0x10 <port+1}\]
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
  - But could be remapped by the kernel
- Configuration 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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Vendor ID, Device ID</td>
<td>Configuration register with vendor and device</td>
</tr>
<tr>
<td>0x4</td>
<td>Command Reg.</td>
<td>Command and control register</td>
</tr>
<tr>
<td>0x6</td>
<td>Status Reg.</td>
<td>Status and control register</td>
</tr>
<tr>
<td>0x8</td>
<td>Revision ID</td>
<td>Revision identification register</td>
</tr>
<tr>
<td>0xa</td>
<td>Class Code</td>
<td>Class code for device class</td>
</tr>
<tr>
<td>0xc</td>
<td>Cache Line</td>
<td>Cache line for device</td>
</tr>
<tr>
<td>0x0d</td>
<td>Latency Timer</td>
<td>Latency timer for device</td>
</tr>
<tr>
<td>0xe</td>
<td>Header Type</td>
<td>Header type for device</td>
</tr>
<tr>
<td>0xf</td>
<td>BIST</td>
<td>Built-in self-test register (BIST)</td>
</tr>
<tr>
<td>0x10</td>
<td>Base Address 0</td>
<td>Physical memory address register</td>
</tr>
<tr>
<td>0x20</td>
<td>Base Address 4</td>
<td>Base Address for device</td>
</tr>
<tr>
<td>0x30</td>
<td>Expansion ROM Base Address</td>
<td>Expansion ROM base address for device</td>
</tr>
</tbody>
</table>

- **Required Register**: Must be implemented by the device driver
- **Optional Register**: May be implemented by the device driver

*From device driver book*

**Figure 12-2. The standardized PCI configuration registers**
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
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
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 demultiplex interrupts
  – Being able to “load balance” the IRQs is useful
Direct Memory Access (DMA)

• 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

- DMA buffers must be in physical memory
  - Like page tables and IDTs
- Some buses (SBus) can use virtual addresses
- Most (PCI) use physical (avoid page translation)
  - IOMMUs allow “virtual addresses” for devices
    - Can make random physical pages look contiguous to the device
    - Called “Bus address” for clarity
    - New to the x86 (called VT-d)
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
Advanced Host Controller Interface (AHCI)

- Hardware “wrapper” around SATA
  - Defines a standard way to talk to disk drives
- Sits on the PCI bus
  
  Bus 0, device 4, function 0:
  SATA controller: PCI device 8086:2922
  IRQ 11.
  BAR4: I/O at 0xc040 [0xc05f].
  BAR5: 32 bit memory at 0xfefb0000 [0xfefb0fff].
  id "ahci"

- BAR5 is BIOS-assigned *physical* address range
  - Called “ABAR” (AHCI BAR)
  - Note: in above example, it’s above 1GB
    - Can be moved/remapped as convenient
AHCI Memory Space

HBA Memory Space Usage

- Config Registers
- ABAR

HBA Registers

- Memory Registers
- Generic Host Control
- Reserved
- Vendor Specific
- Port 0
- Port 1
- Port 31

System Memory

- Command List (Port 0)
- Command List (Port 1)
- Command List (Port 31)

Received FIS Structure (Port 0)

Received FIS Structure (Port 1)

Received FIS Structure (Port 31)

Ports 1 - 31 optional

From OSdev.org
AHCI Setup

• Initialized via memory-mapped addresses
  – Not ports!
  – Can just cast the pointer to struct and write to it

• Must set CLB and FBU
  – CLB points to buffer of command headers
    • Called “slots”
  – FBU points to buffer for received data
AHCI Operation (1/2)

• Pick a free command slot (on port to read)
  – Can send separate command in each slot
  – Slot points to a Command Table

• Write command into Command Table
  – Frame Information Structure (FIS) to dictate command
    • Determines what operation to perform
  – Physical Region Descriptor Table (PRDT) to dictate target
    • Determines what data to do it with

• Where are these things?
  – Headers are memory mapped, tables are in host memory
    • Device reads host memory as needed
AHCI Operation (2/2)

• Command written to FIS tells device what to do
  – Command itself: “READ DMA”
  – LBA to start reading from
  – Count of bytes to read

• When ready, write “issue command” to port
  – Memory-mapped write triggers operation

• When command is done, interrupt arrives
  – Iterate through sent commands
    • Find the one that’s done
Handling Interrupts

• IRQs masked while in interrupt
  – 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

• Bottom half is a kernel thread
  – Can be one that is waiting for interrupt
    • e.g., thread that sent a disk request on block cache miss
  – Can be separate kernel thread
    • *softirq* in Linux
      • Very high priority thread - first to be scheduled on IRET
Network Drivers

• Mostly very dumb devices
  – Get packet from net, DMA to host, raise interrupt
  – On send, DMA from host, send packet, raise interrupt

• Nearly all implementations use *ring buffers*
  – Give list of pages to NIC
  – NIC uses first available buffer for received packet
  – NIC sends out packets from ready buffers

• Controlling each NIC vastly different
  – No AHCI-like wrapper across vendors
Intel E1000 NIC

- BAR0 used to talk to NIC
- NIC smart enough to understand ring buffer
  - Follow circular linked list on their own
- Each buffer has a header
  - Header used for flow control
    - Report Status (RS) used to request progress
    - Descriptor Done (DD) indicates buffer can be reused
      - If buffer is full, NIC drops incoming packets
      - If buffer is full, host probably drops outgoing packets
        - Could try to increase buffer size
- Write to Transmit Descriptor Tail (TDT) to initiate xfer