# Input / Output

W4118 Operating Systems I

columbia-os.github.io

# The OS and I/O

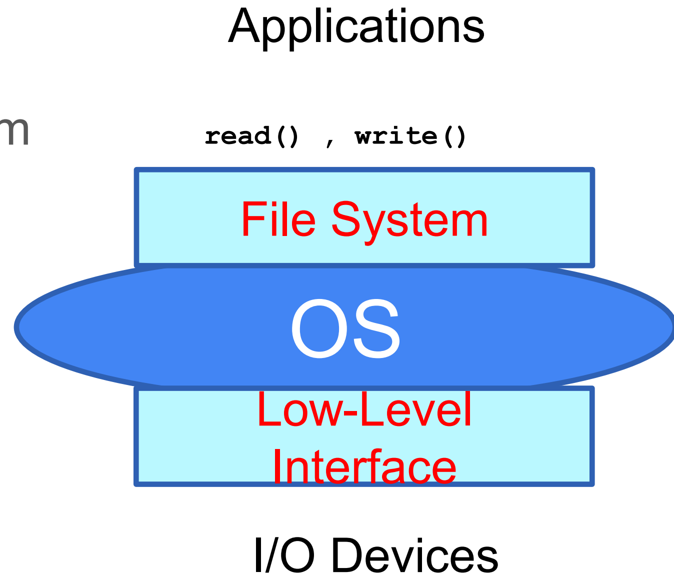
- The OS controls all I/O devices
  - Issue commands to devices
  - Catch interrupts
  - Handle errors

- Provides an interface between the devices and the rest of the system

- A few exceptions are processor built-in accelerators ( encryption, compression engines) that often can be invoked by applications.
  - But think of these less as I/O but as a different kind of functional units in the CPU

# The OS and IO: Simplified View

- Applications rarely communicate directly with Devices
- Would be too hardware specific
- OS responsible of translating filesystem requests (read/write) into low level device specific I/O requests

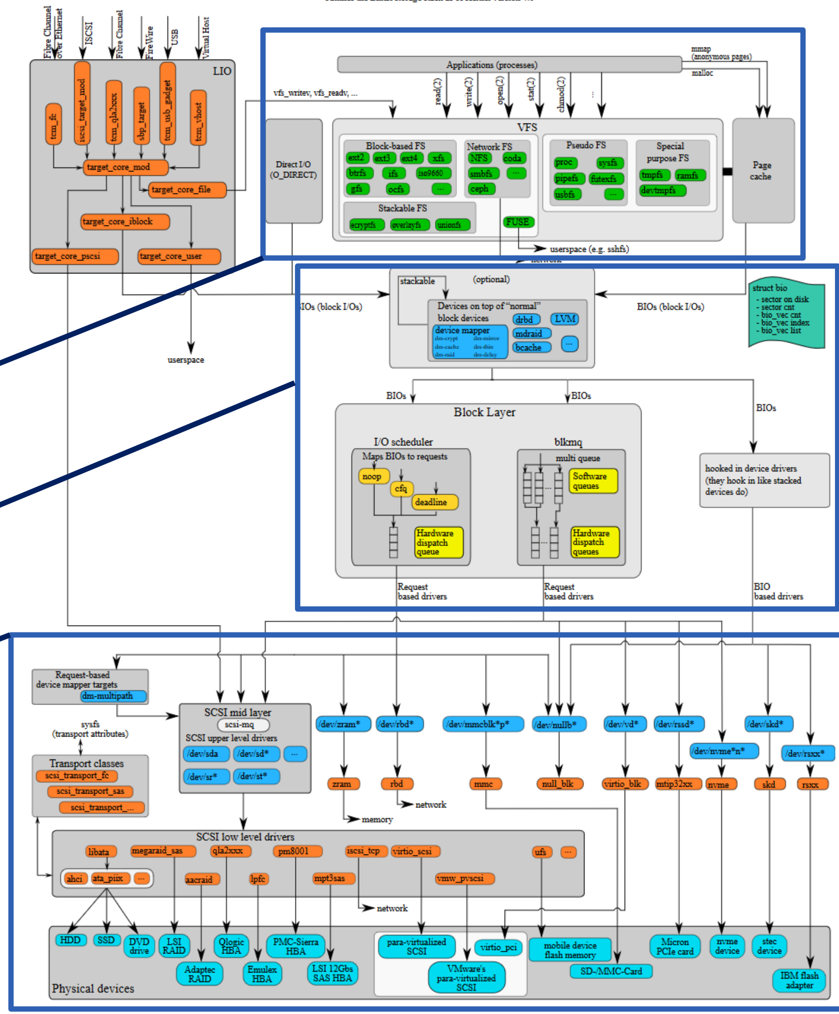


# Full Linux I/O Stack (4.10)

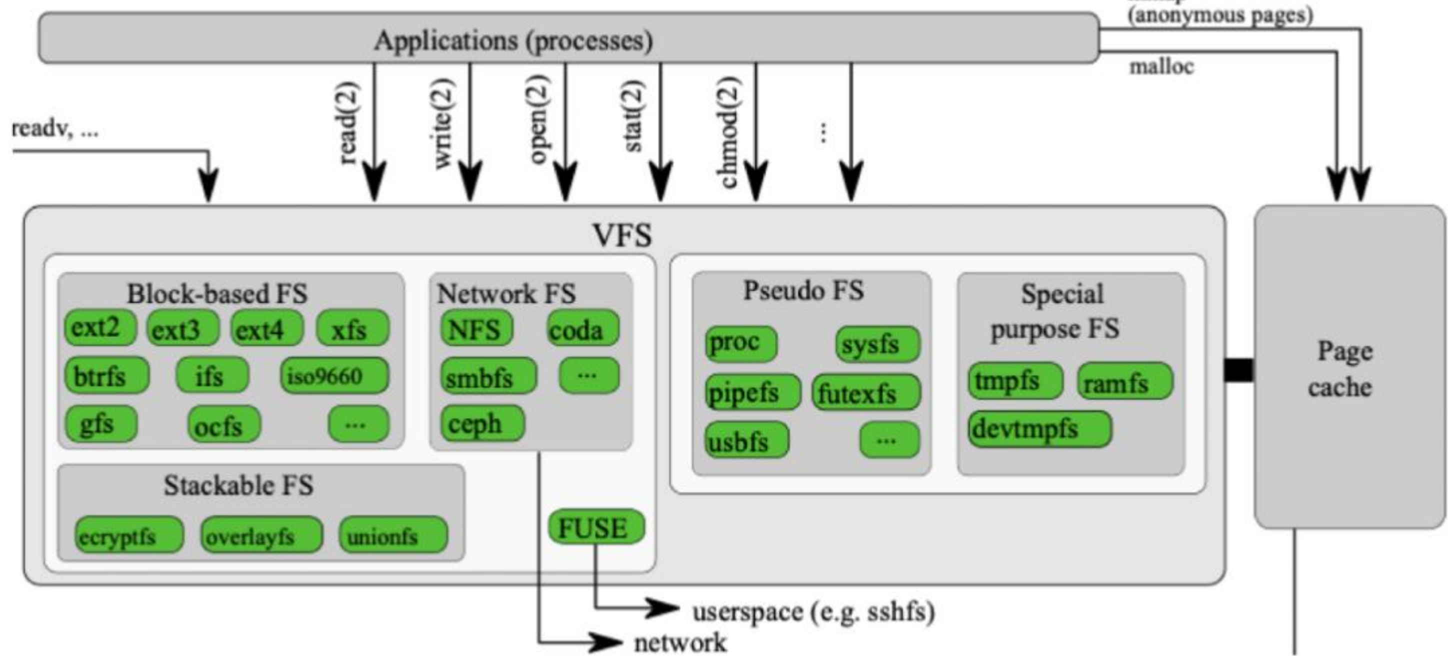
File System

Block I/O Layer

Device Layer



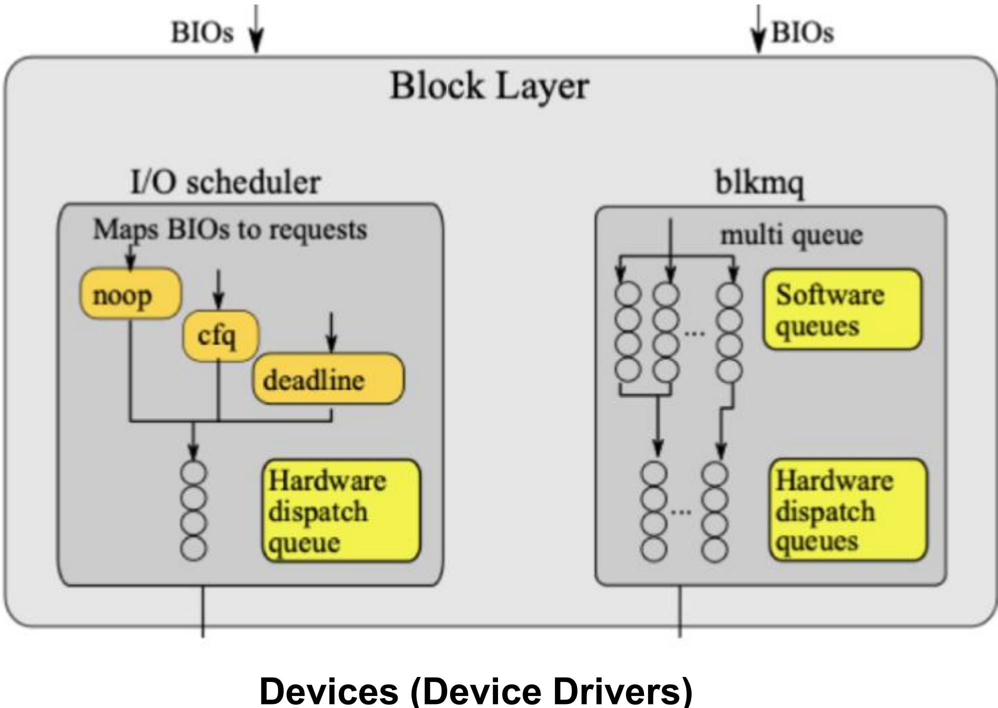
# Linux File System




BIOs

Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

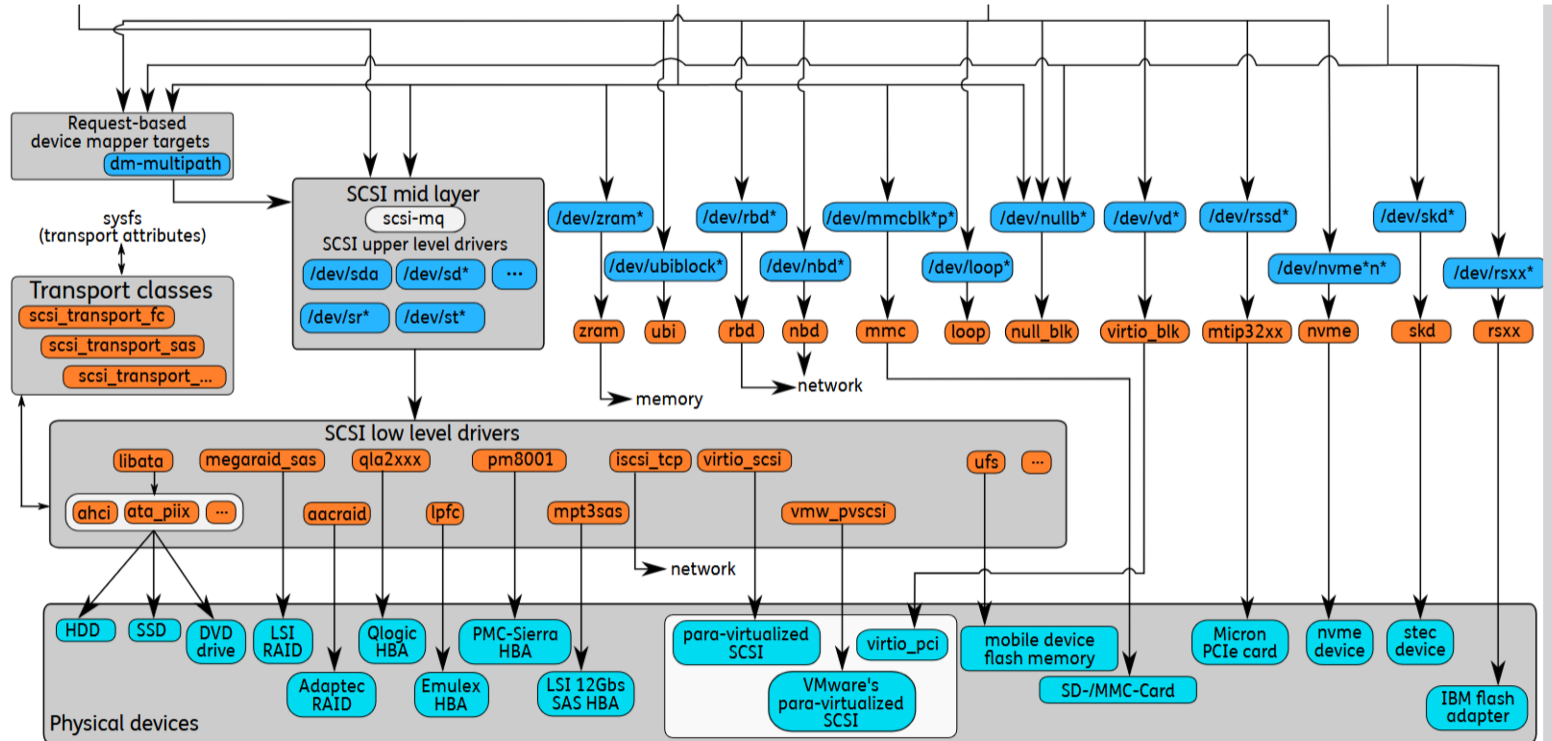
# Linux Block I/O Layer




**Devices (Device Drivers)**

Image Credit: Thomas-Krenn AG, *Linux Storage Stack Diagram*

# Linux Device Layer

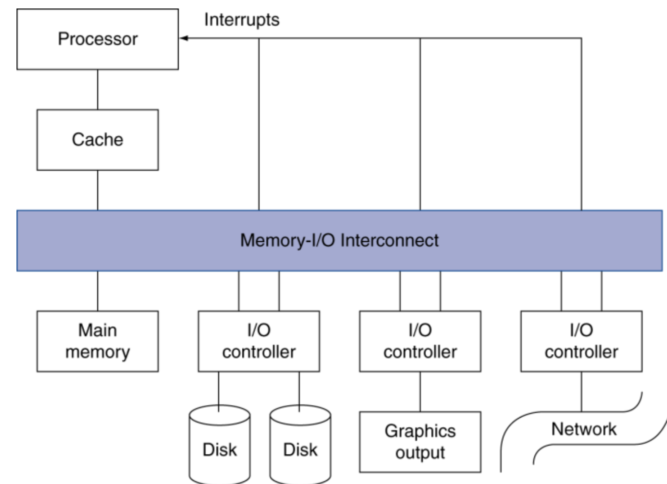


# I/O Devices: Challenges

- Very diverse devices
  - — behavior (i.e., input vs. output vs. storage)
  - — partner (who is at the other end?)
  - — data rate

- I/O Design affected by many factors
  expandability, resilience

- Performance:
  - access latency and throughput
  - connection between devices and the system
  - the memory hierarchy
  - the operating system



# I/O Devices

- Block device
  - Stores information in fixed-size blocks
  - Each block has its own address
  - Transfers in one or more blocks
  - Example: Hard-disks, CD-ROMs, USB sticks

- Character device
  - Delivers or accepts stream of character
  - Is not addressable
  - Example: mice, printers, network interfaces

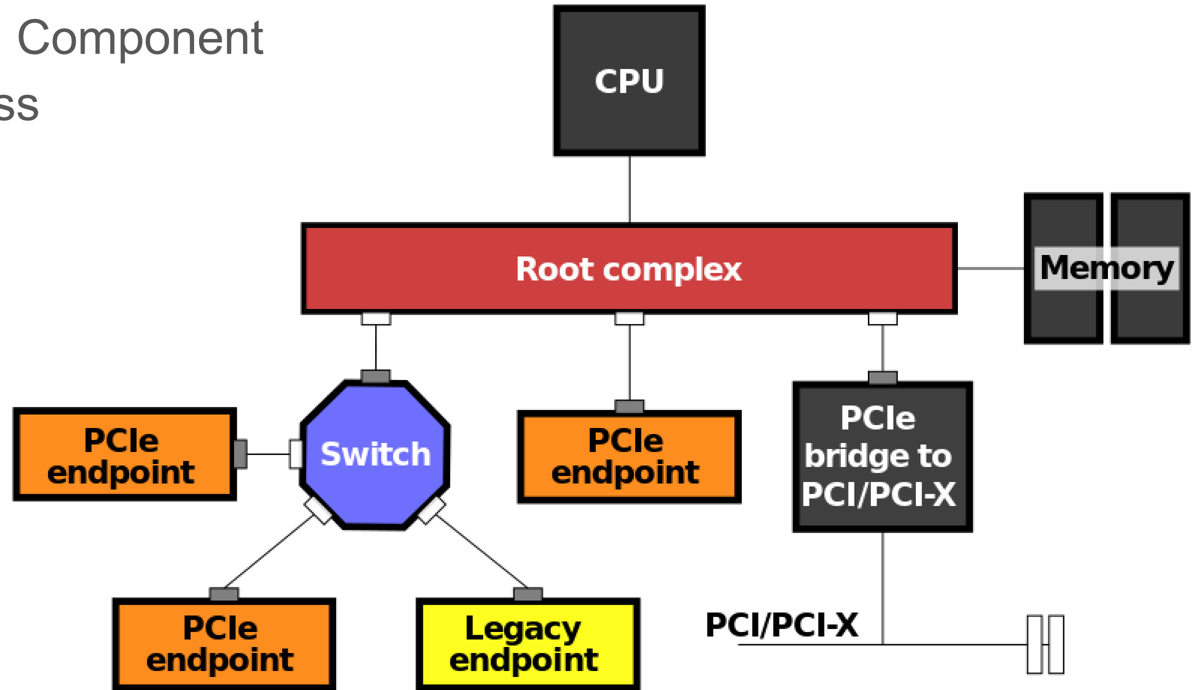
# Devices ( Feed and Speed) !!!

- High degree of diversity in devices.

| Device | Data rate |
| --- | --- |
| Keyboard | 10 bytes/sec |
| Mouse | 100 bytes/sec |
| 56K modem | 7 KB/sec |
| Bluetooth 5 BLE | 256 KB/sec |
| Scanner at 300 dpi | 1 MB/sec |
| Digital video recorder | 3.5 MB/sec |
| 802.11n Wireless | 37.5 MB/sec |
| USB 2.0 | 60 MB/sec |
| 16x Blu-ray disc | 72 MB/sec |
| Gigabit Ethernet | 125 MB/sec |
| SATA 3 disk drive | 600 MB/sec |
| USB 3.0 | 625 MB/sec |
| Single-lane PCIe 3.0 bus | 985 MB/sec |
| 802.11ax Wireless | 1.25 GB/sec |
| PCIe Gen 3.0 NVMe M.2 SSD (reading) | 3.5 GB/sec |
| USB 4.0 | 5 GB/sec |
| PCI Express 6.0 | 126 GB/sec |

# PCI-e

- StandardPeripheral Component Interconnect Express




Standard means to connect modern devices to the system

# Controller and Device

- Each controller on the device has a few registers used to communicate with CPU
- By writing/reading into/from those device registers, the OS can control the device.
- (1) cmd buffers to start/stop operations
- (2) There are also data buffers in the device that can be read from or written to by the OS
  - Location/address of buffer
  - Location/address of I/O "object"
  - Size of data to transfer

# CPU – Device Communication

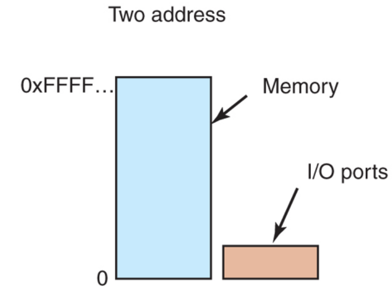
How does CPU communicate with device control registers and data buffers?

Two main approaches:

- I/O port space
- Memory-mapped I/O

# I/O Port Space

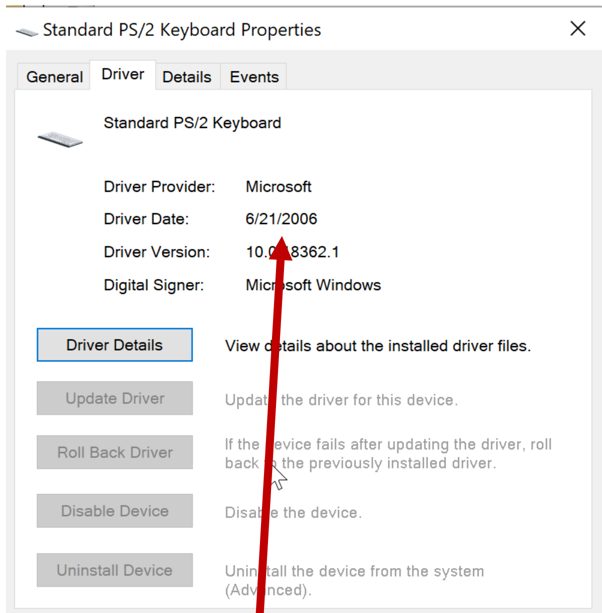
- Each control register is assigned an I/O port number
- The set of all I/O ports form the I/O port space
- I/O port space is protected → only kernel can access
  execute privileged instructions:
  - `in  portnum, target`
  - `out portnum, source`



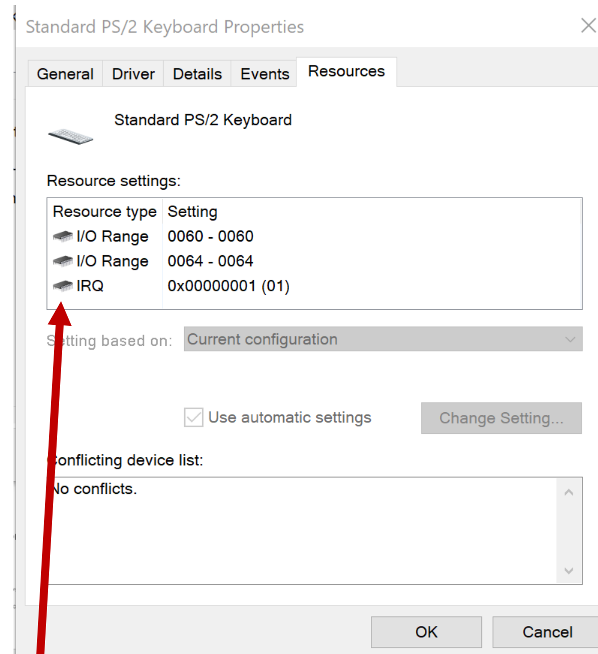

- Classical x86 way
- Largely done only for “legacy” devices

| Opcode | Mnemonic | Description |
|---|---|---|
| E4 ib | IN AL,imm8 | Input byte from imm8 I/O port address into AL. |
| E5 ib | IN AX,imm8 | Input byte from imm8 I/O port address into AX. |
| E5 ib | IN EAX,imm8 | Input byte from imm8 I/O port address into EAX. |
| EC | IN AL,DX | Input byte from I/O port in DX into AL. |
| ED | IN AX,DX | Input word from I/O port in DX into AX. |
| ED | IN EAX,DX | Input doubleword from I/O port in DX into EAX. |

# Example Win Keyboard




Talk about legacy software




two io-ports  and one interrupt

# Example: Serial and Parallel Ports

| COM Port # | IRQ | I/O Port Address |
| --- | --- | --- |
| 1 | 4 | 3F8-3FFh |
| 2 | 3 | 2F8-2FFh |
| 3 | 4 | 3E8-3EFh |
| 4 | 3 | 2E8-2EFh |

| LPT Port # | IRQ | I/O Port Address Range |
| --- | --- | --- |
| LPT1 | 7 | 378-37Fh or 3BC-38Fh |
| LPT2 | 5 | 278-27Fh or 378-37Fh |
| LPT3 | 5 | 278-27Fh |

# Memory-Mapped I/O

- Map control registers into the memory space
- Each control register is assigned a unique physical memory address offset
- These physaddr are mapped through the pagetable into the kernel address space and can now be operated on via memory load/store
- Load/stores to the memory are “snooped” on by the device and acted upon, PCI device is told the range to snoop on
- Every modern architecture basically does it this way

One address space

# Advantages of Memory-Mapped I/O

- Device drivers can be written entirely in C (since no special instructions are needed)

- No special protection is needed from OS, just refrain from putting that portion of the address space in any user's virtual address space

- Every instruction that can reference memory can also reference control registers

# PCI-Configuration Space

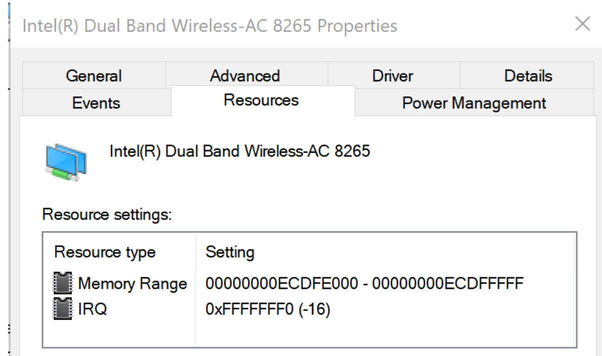
*Table 2-22:* **Type 0 PCI Configuration Space Header**

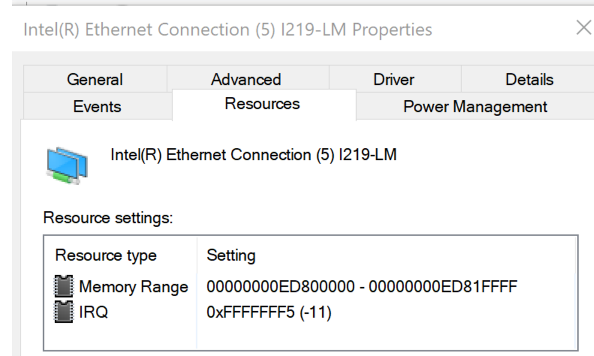
| 31 | 16 | 15 | 0 | |
|---|---|---|---|---|
| Device ID | | Vendor ID | | 00h |
| Status | | Command | | 04h |
| Class Code | | | Rev ID | 08h |
| BIST | Header | Lat Timer | Cache Ln | 0Ch |
| Base Address Register 0 | | | | 10h |
| Base Address Register 1 | | | | 14h |
| Base Address Register 2 | | | | 18h |
| Base Address Register 3 | | | | 1Ch |
| Base Address Register 4 | | | | 20h |
| Base Address Register 5 | | | | 24h |
| Cardbus CIS Pointer | | | | 28h |
| Subsystem ID | | Subsystem Vendor ID | | 2Ch |
| Expansion ROM Base Address | | | | 30h |
| Reserved | | | CapPtr | 34h |
| Reserved | | | | 38h |
| Max Lat | Min Gnt | Intr Pin | Intr Line | 3Ch |

Standard PCI Control Space

```
struct pci_config {
      uint16 devid;
      uint16 vendorid;
      :
      uint32 bar0;
      :
      uint32 bar1;
      :
      uint8 maxlat;
      :
      uint8 intr;
};
```

# Examples for network devices




Range: 0x2000 == 8KB

```
struct wlac_8265 {
        struct pci_config pcie_config;
        <specific structure for 8265 dev>
};


struct wlac_8265 *wldev =
   (struct wlac_8265*) kern_addr(0xECDFE000);
wldev->memberelem =  …..
```

Range: 0x20000 == 128KB

```
struct lan_I219 {
        struct pci_config pcie_config;
        <specific structure for I219ev>
};


struct lan_I219 *landev =
   (struct lan_I219*) kern_addr(0xED800000);
landev->memberelem = …..
```

virtual kernel address

physical bus address

# Inspecting your PCI subsystem

What devices are connected to my system ?

```
[frankeh@linserv1 ~]$ lspci | grep -v "^00"
01:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
01:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
02:00.0 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
02:00.1 Ethernet controller: Broadcom Inc. and subsidiaries NetXtreme II BCM5709 Gigabit Ethernet (rev 20)
03:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:00.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:01.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:04.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
04:05.0 PCI bridge: PLX Technology, Inc. PEX 8624 24-lane, 6-Port PCI Express Gen 2 (5.0 GT/s) Switch [ExpressLane] (rev bb)
05:00.0 RAID bus controller: Broadcom / LSI MegaRAID SAS 2108 [Liberator] (rev 05)
0a:03.0 VGA compatible controller: Matrox Electronics Systems Ltd. MGA G200eW WPCM450 (rev 0a)
20:00.0 Host bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890 Northbridge only dual slot (2x8) PCI-e GFX Hydra part (rev 02)
20:00.2 IOMMU: Advanced Micro Devices, Inc. [AMD/ATI] RD890S/RD990 I/O Memory Management Unit (IOMMU)
20:02.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD9x0/RX980 PCI to PCI bridge (PCI Express GFX port 0)
20:03.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD9x0 PCI to PCI bridge (PCI Express GFX port 1)
20:0b.0 PCI bridge: Advanced Micro Devices, Inc. [AMD/ATI] RD890/RD990 PCI to PCI bridge (PCI Express GFX2 port 0)
```

```
[frankeh@linserv1 ~]$ lspci -n | grep -v "^00"
01:00.0 0200: 14e4:1639 (rev 20)
01:00.1 0200: 14e4:1639 (rev 20)
02:00.0 0200: 14e4:1639 (rev 20)
02:00.1 0200: 14e4:1639 (rev 20)
03:00.0 0604: 10b5:8624 (rev bb)
04:00.0 0604: 10b5:8624 (rev bb)
04:01.0 0604: 10b5:8624 (rev bb)
04:04.0 0604: 10b5:8624 (rev bb)
04:05.0 0604: 10b5:8624 (rev bb)
05:00.0 0104: 1000:0079 (rev 05)
0a:03.0 0300: 102b:0532 (rev 0a)
20:00.0 0600: 1002:5a12 (rev 02)
20:00.2 0806: 1002:5a23
20:02.0 0604: 1002:5a16
20:03.0 0604: 1002:5a17
20:0b.0 0604: 1002:5a1f
```

Add item
Discuss
Help
ID syntax

**The PCI ID Repository**

The home of the pci.ids file

Log in

Main -> PCI Devices -> **Vendor 14e4**

**Name: Broadcom Inc. and subsidiaries**

## Discussion

**Name: Broadcom Corporation**

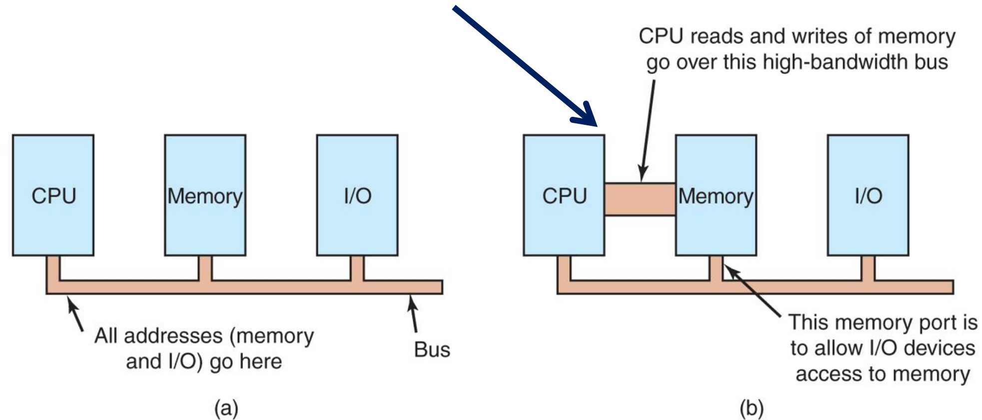
*mj*

*2002-08-14 18:16:25*

# Something to be conscious about with Memory-Mapped I/O

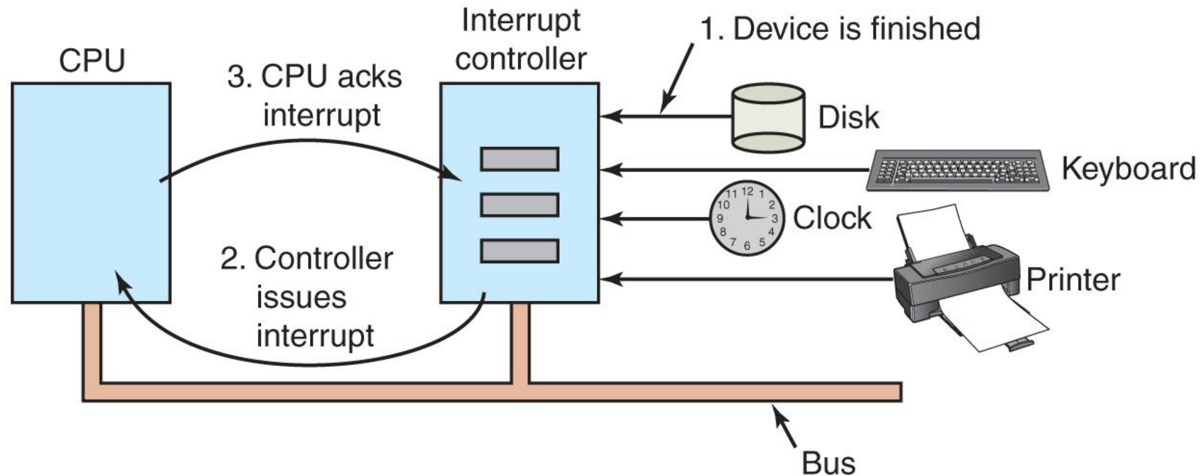
- Caching a device control register can be disastrous, after all it is just a “memory load”

- PTE has a “cache disabled bit

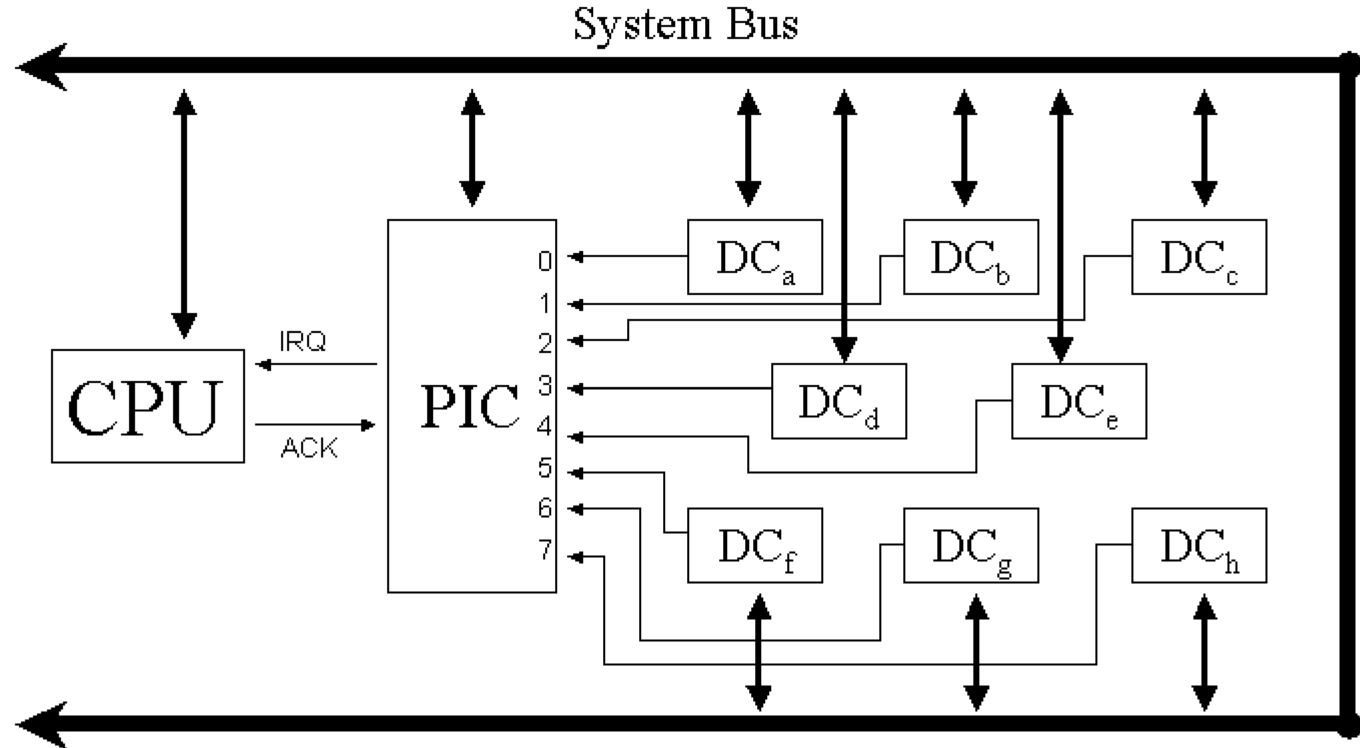


# Interrupts

- When an I/O device has finished the work given to it, it causes an interrupt.
- More generally, anytime a device wants attention it causes an interrupt.



# (Advanced) Programmable Interrupt Controllers [ APIC ]



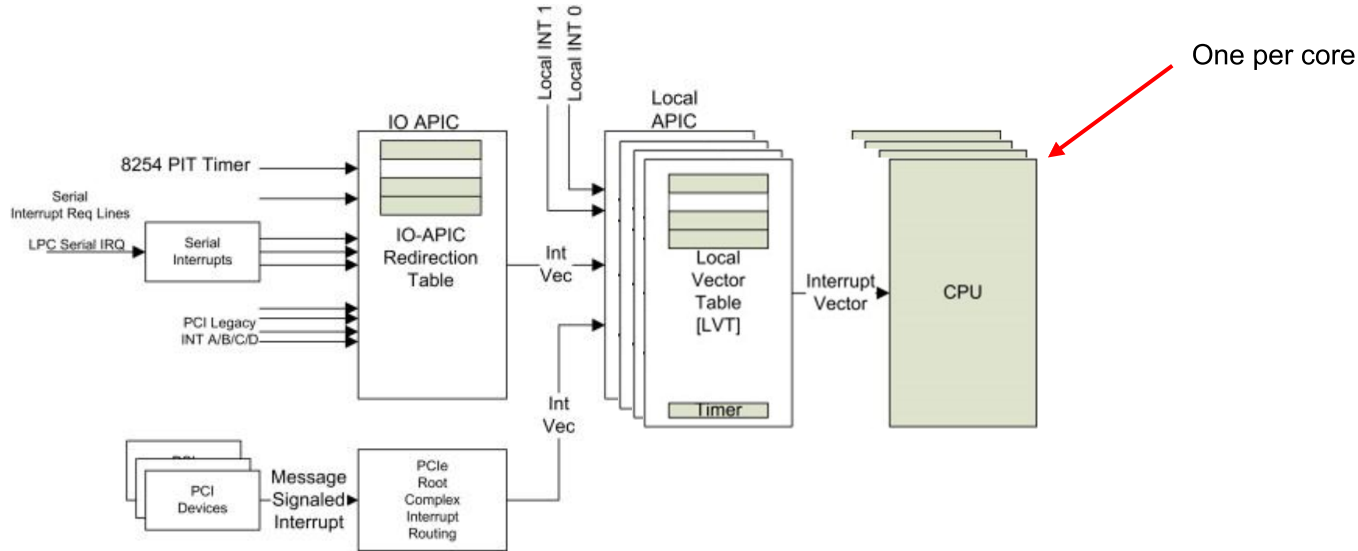

DC == Device Controller

# Message Signaled Interrupts

- Hardware lines from device to APIC too limited in numbers → pin count restrictions
- MSI allows the device to write a small amount of interrupt-describing data to a special memory-mapped I/O address
- Writing this “value” to a particular memory address triggers an interrupt.
- The PCI chipset delivers the corresponding interrupt to a processor.
- PCI-2.2 defined MSI with upto 32 interrupts
- PCI-3.0 defines MSI-X with upto 2048 interrupts
- Highly programmable

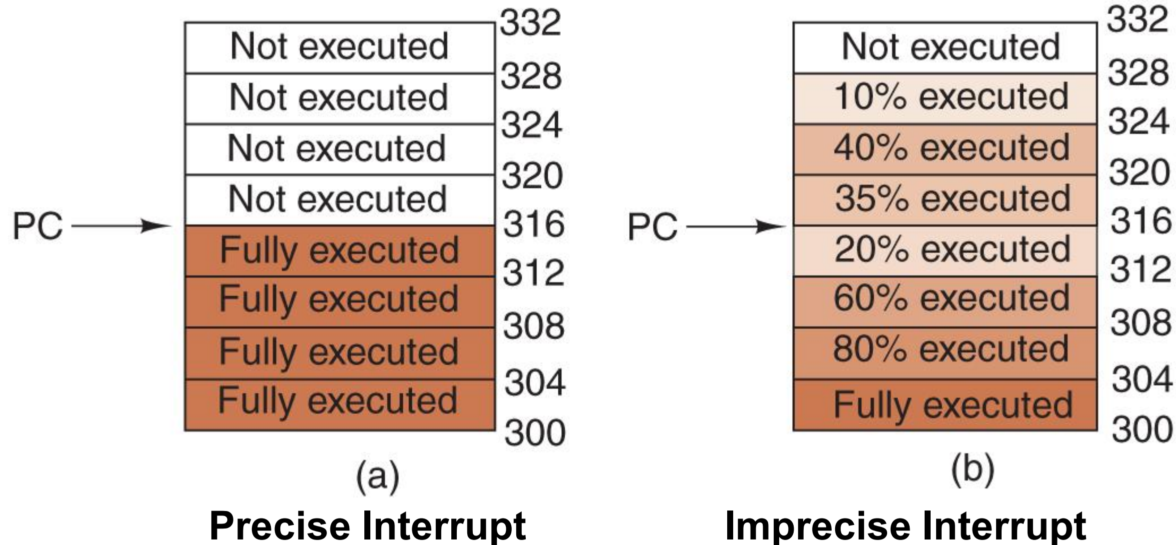
# Complete Interrupt System

- Split Architecture
  - Global: IO APIC and PCI-based MSI Interrupt Routing
  - Local: per core Local APIC



# Issues with Interrupt processing

- Architectures are pipelined and OOO
- What does it then mean to “switch the next instruction to enter the kernel at __entry ?



# Precise Interrupts

- Makes handling interrupts much simpler

- Has 4 properties
  - The program counter (PC) is saved in known place (typically a special register only accessible through kernel mode)
  - All instructions before the one pointed by PC have fully executed
  - No instruction beyond the one pointed by PC has been executed (or has any sideeffects)
  - The execution state of the instruction pointed to by the PC is known

- Hugely important with out of order / superscalar processors

# I/O Software

- Device independence
- Uniform naming
- Error handling
    - Should be handled as close to the hardware as possible
- Synchronous vs asynchronous (interrupt-driven)
- Buffering
- Sharable versus dedicated devices

# Three Ways of Performing I/O (Control and Data)

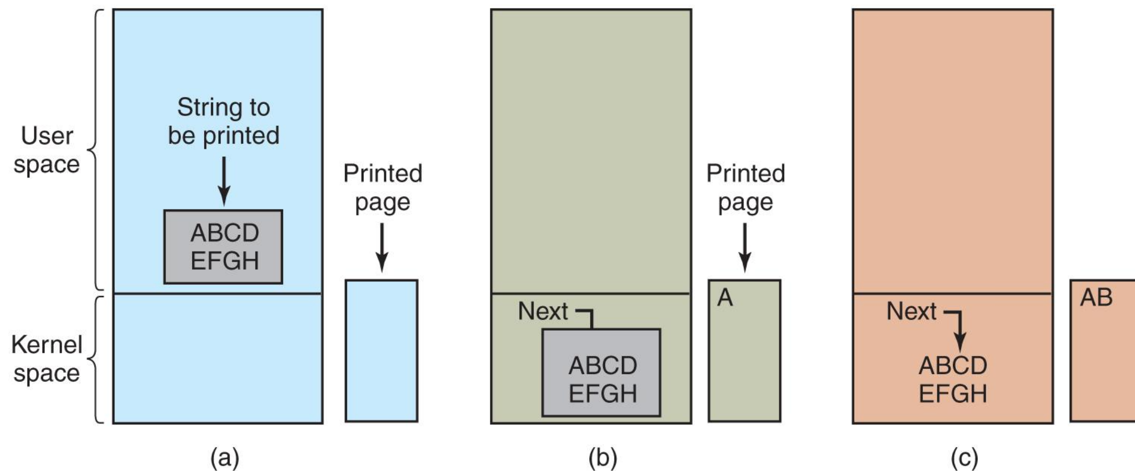
- Programmed I/O
- Interrupt-driven I/O
- I/O Using DMA

# Programmed I/O

- CPU does all the work
- Busy-waiting (polling)

Example:

```
copy_from_user(buffer, p, count);              /* p is the kernel buffer */
for (i = 0; i < count; i++) {                  /* loop on every character */
    while (*printer_status_reg != READY) ;     /* loop until ready */
    *printer_data_register = p[i];             /* output one character */
}
return_to_user( );
```

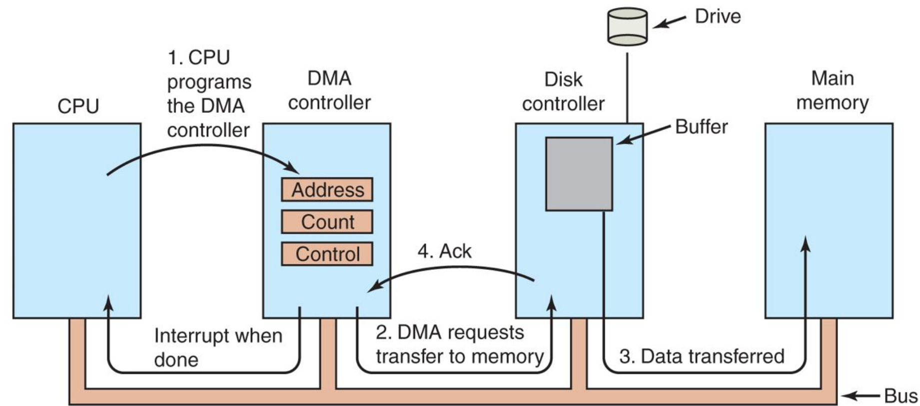


# Interrupt-Driven I/O

- Waiting for a device to be ready, the process is blocked and another process is scheduled.
- When the device is ready it raises an interrupt.
- Then data is copied by CPU as previous
  (avoids polling)
- Upon completion of the transfer the blocked process can be made ready again.

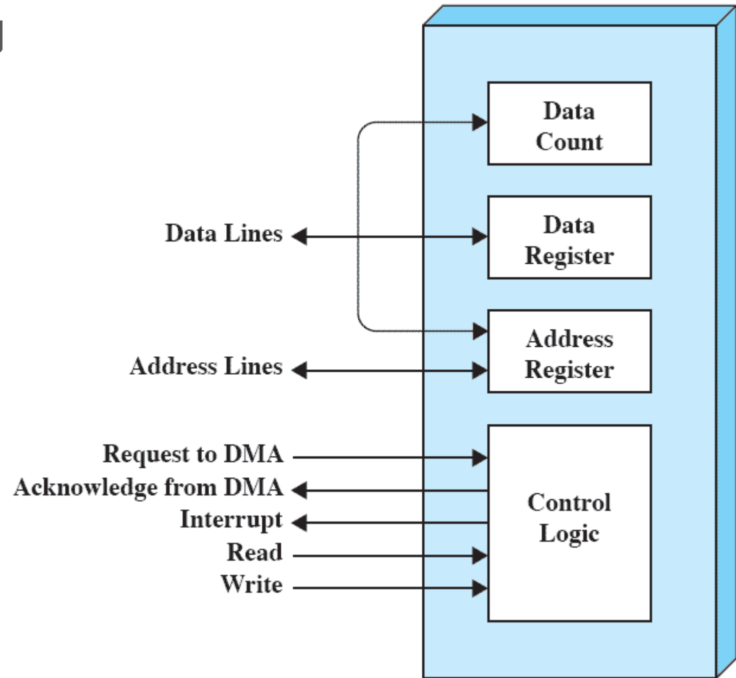
# Direct Memory Access (DMA)

- It is not efficient for the CPU to request data from I/O one byte/word at a time
- DMA controller has access to the system bus independent of the CPU



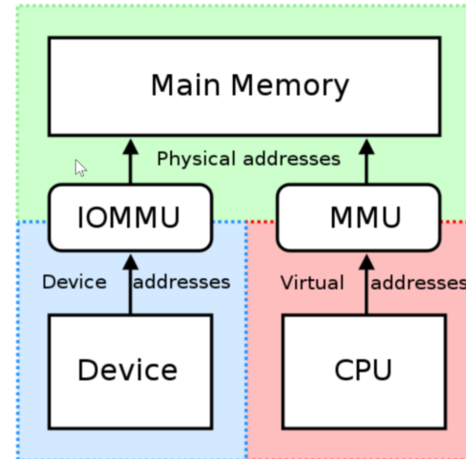
# I/O Using DMA

- DMA does the work instead of the CPU
- Think of DMA engine as a simple "copy-core"
- Let the DMA do its work and then use interrupts to notify CPU



# DMA and Memory Protection

- Since DMA allows devices access to memory it can be a source of BUGs and security exposures (and considerable headaches)
- A System / Operating System protects itself with another translation table → IOMMU
- Enables OS to install only buffer addresses in IOMMU to which it allows DMA / device accesses
- If Device attempts DMA to an address and there is no translation, an interrupt/exception is raised
- It is similar how apps are restricted to what memory they can access using a pagetable (MMU).



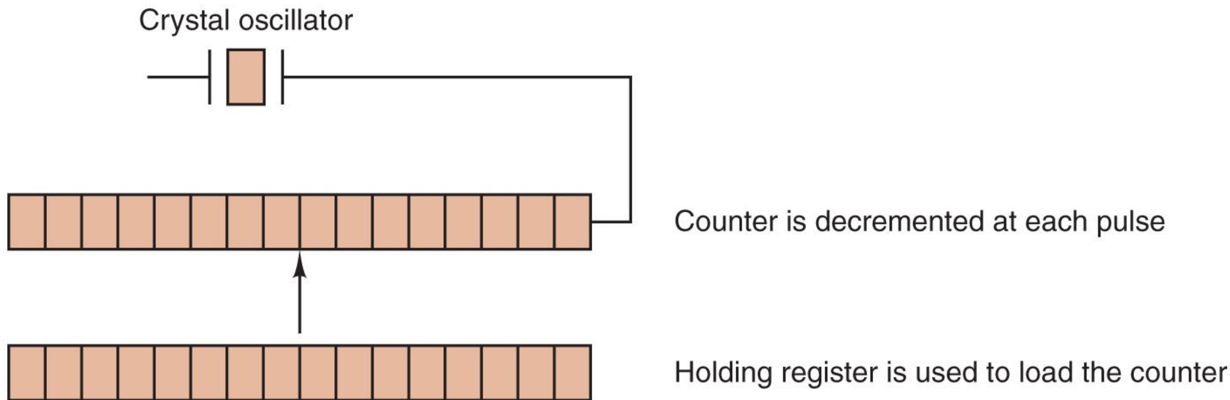
# Example of Devices

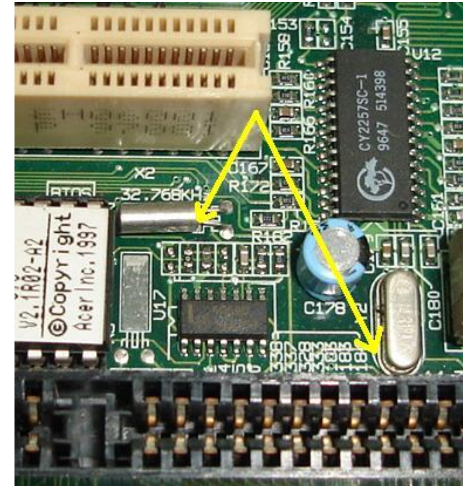
- Let’s look at two devices

- Clock which helps maintains the time

# Clock Hardware

- Old: tied to the power line and causes an interrupt on every voltage cycle
- New: Crystal oscillator + counter + holding register



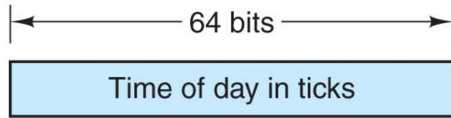



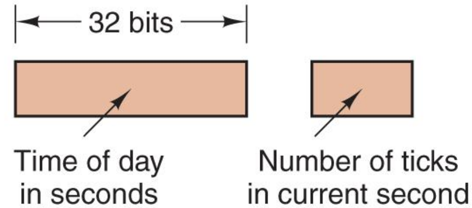
# Clock Software

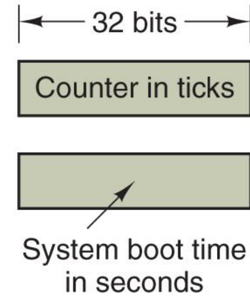
- Maintaining the time of the day
- Preventing processes from running longer than they are allowed to
- Accounting for CPU usage
- Handling alarm system call made by user processes
- Providing watchdog timers for parts of the system itself
- Doing profiling, monitoring, and statistics gathering

# Maintaining Time of Day

Three ways to maintain the time of day.



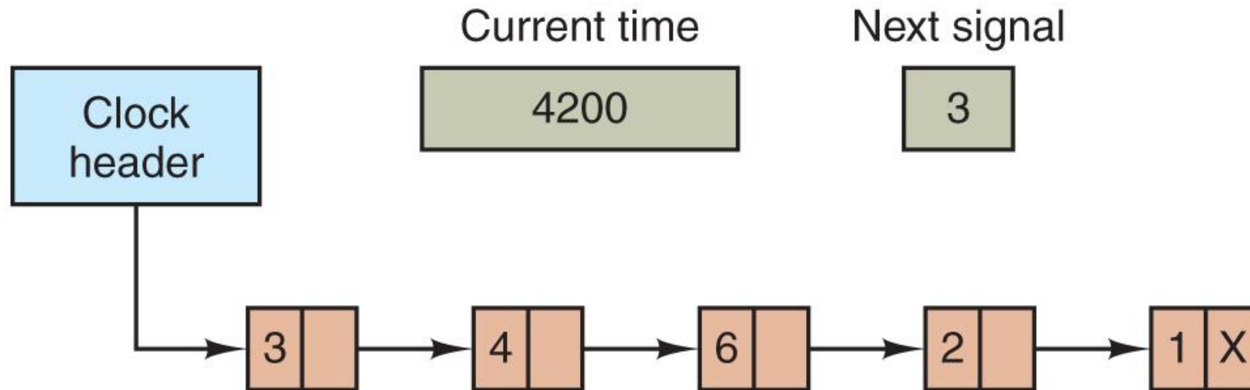




(a) (b) (c)

# Emulating Future Time Events

Three ways to maintain the time of day.



# Linux Networking Driver

- Post buffers for incoming packets for device to DMA
- Post Control Blocks to device
- Clear the incoming queue

- Throttle TCP/IP stack when rate is too high
- Throttle Device when too many packet arrive

- Multitude of tasks with high degree on concurrency at very high data rates