EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

What we covered last two lectures

- File systems in general
  - abstraction
  - files and directories
  - inodes
  - disk block management
- Linux ext2/ext3 File system, key concepts:
  - block groups - what are these?
  - superblocks, bitmap free blocks, inodes
  - inode structure & directory structure
  - journaling, extents

Input and Output

What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers & Port IO
- Direct Memory Access devices
- Interrupts and interrupt handling
- Precise & Imprecise interrupts
- IO software, specifically Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

I/O Devices

- The OS is responsible for managing I/O devices
  - Issuing requests
  - Managing corresponding interrupts
- The OS provides a high-level, easy-to-use interface to processes
- The interface, in principle, should be as uniform as possible
- The I/O subsystem is the part of the kernel responsible for managing I/O
- Composed of a number of *device drivers* that deal directly with the hardware

Drivers & I/O

- Code for handling interactions with hardware peripherals makes up a significant portion of most OSes
- Interacting with hardware is complicated and hard to get right
  - This is one reason we make OS/kernel do it!
  - Another reason is that hardware is a shared resource the OS needs to manage!

```
Totals grouped by language (dominant language first):
ansic:     12565237 (97.03%)

SLOC    Directory   SLOC-by-Language (Sorted)
7702364 drivers         ansic=7695559,perl=2839,yacc=1688,asm=1482,lex=779, sh=17
2004163 arch            ansic=1722641,asm=277802,perl=2564,sh=972,awk=478,etc.
817722  fs              ansic=817722
656828  sound           ansic=656645,asm=183
629822  net             ansic=629701,awk=121
422915  include         ansic=419514,cpp=3359,asm=42
155256  kernel          ansic=155239,asm=17
140305  tools           ansic=127661,perl=3977,sh=3913,python=2203,etc.
74125   lib             ansic=73993,perl=119,awk=13
69173   crypto          ansic=69173
68032   mm              ansic=68032
62853   Documentation   xml=50498,ansic=7239,perl=2542,sh=1183,etc.
50034   security        ansic=50034
48927   scripts         ansic=26753,perl=10866,python=4011,sh=2975,etc.
24151   block           ansic=24151
7436    virt            ansic=7436
6232    ipc             ansic=6232
4953    samples         ansic=4671,sh=282
2680    init            ansic=2680
1877    firmware        asm=1660,ansic=217
558     usr             ansic=544,asm=14
```

61% of all C code

< 1% of all C code

~0.5% of all C code

I/O HW

I/O Devices

Two categories:

- Block devices
  - Store information in blocks of a specified size
  - Block can be accessed (read or written) independently
  - Example: disk
- Character devices
  - Deal with a stream of characters without a predefined structure
  - Characters cannot be addressed independently
  - Example: mouse, printer, keyboard

I/O Devices

Classification not perfect

- Clocks/timers
- Graphics cards
- Tapes
- etc.

Why is an abstraction into block and character devices still useful?

Many things *can* be put into one of the two classes, and we can write interfaces that deal generically with them
FS deal just abstract block interfae

Device Data Rates

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

- **Huge variability in speeds!**
- **OS has to be able to support enormous range**
- **Performance constantly increasing, we now have 100Gbit Ethernet**

Device Controllers

I/O devices typically have two components

- Mechanical component
- Electronic component (e.g., connected to the mechanical component through a cable)

The electronic component is the *device controller*

- Often a PCI/ISA card installed on the motherboard (host adapter)
- May be able to handle multiple devices (e.g., daisy chained)
- May implement a standard interface (SCSI/EIDE/USB)

Controller's tasks

- Convert serial bit stream to block(s) of bytes (e.g., by internal buffering)
- Perform error correction as necessary
- Make data available to CPU/memory system

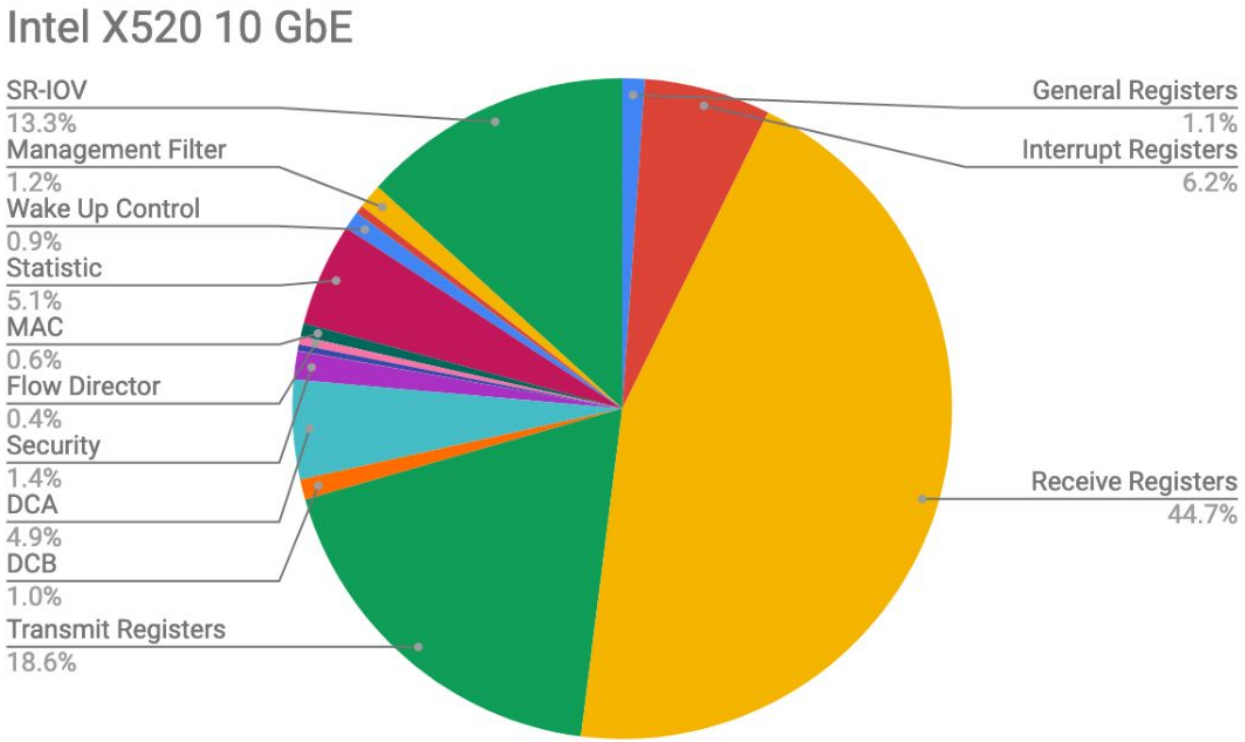
Aside: Controller as Computers

- The electronic portion can be quite complex
- In modern machines, can essentially be another computer
  - General purpose CPU (ARM, PowerPC, SPARC, …)
  - Some RAM - Need to cache data coming off of device; rather than negotiating for bus
  - Some permanent storage (usually flash)

Complexity of a simple Intel NIC

Intel 82599 Datasheet

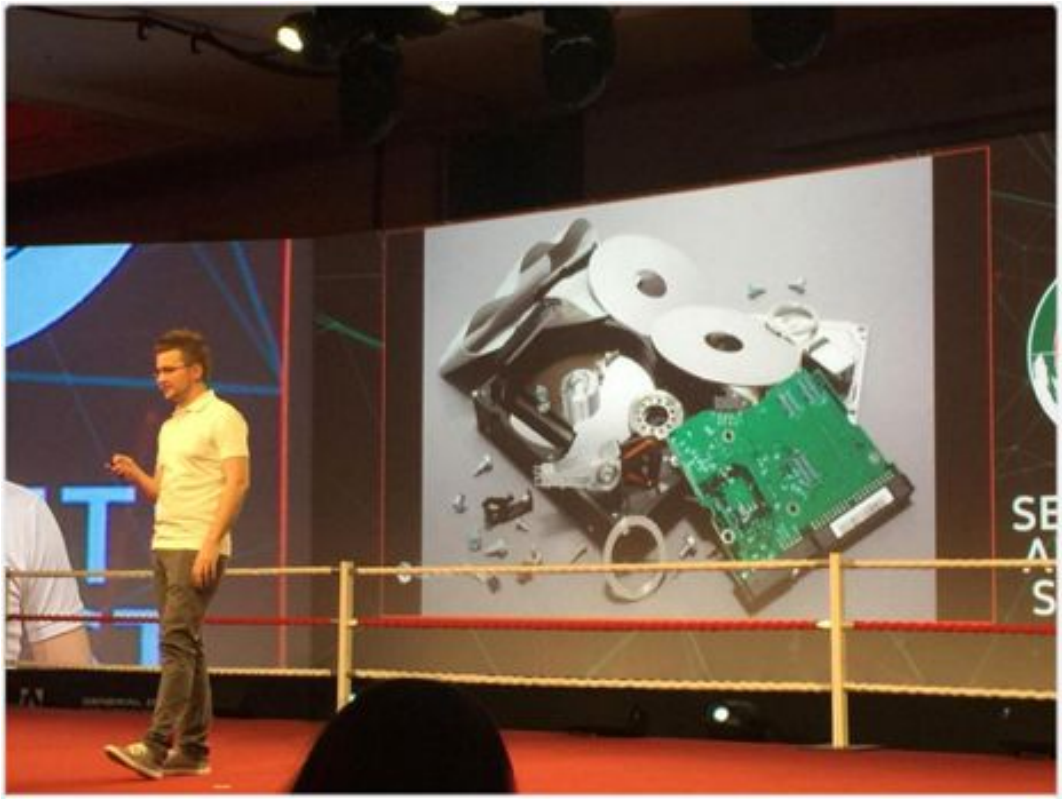
- Total of ~5600 32-bit registers
- Linux initializes ~1300 registers



Firmware

- The device controllers also run software and have their own operating systems (firmware)
- It turns out that this is often upgradable by the user (to fix bugs)
- This is not always a good thing

Hard Drive Backdoors



Fabio Assolini
@assolini

Follow

The only way to remove nls_933w.dll #TheSAS2015 #EquationAPT

2:48 PM - 16 Feb 2015

114 45

Hard Drive Backdoors

- By changing the software running on your hard drive, an attacker could:
  - Hide files from the main OS
  - Re-infect a computer even after wiping the HDD & reinstalling the OS
- There is evidence that the NSA has been using this technique

What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers VS Port IO
- Direct Memory Access
- Interrupts and interrupt handling
- Precise VS Imprecise interrupts
- IO software AKA Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

Accessing the Controller

The OS interacts with a controller

- By writing/reading registers (command/status)
- By writing/reading memory buffers (actual data)

Registers can be accessed through dedicate CPU instructions

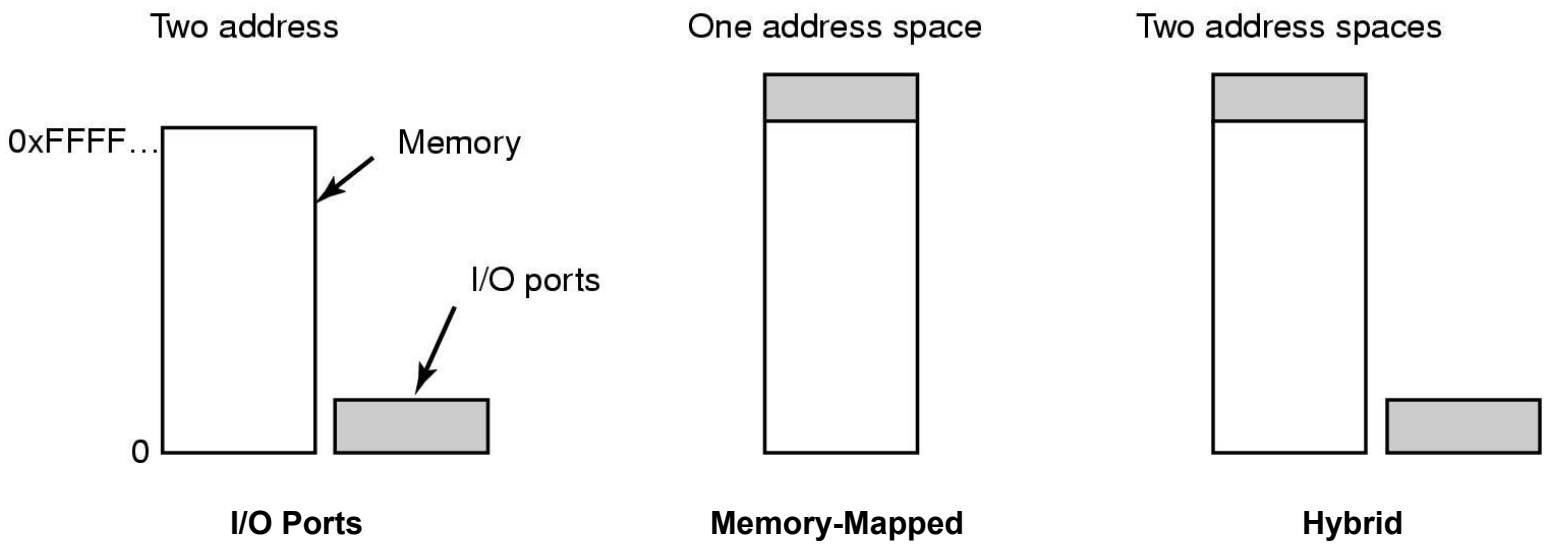
- Registers mapped to I/O ports
- `IN REG, PORT` and `OUT REG, PORT`
  transfer data from CPU's registers to a controller's registers

Registers can be mapped onto memory (Memory-Mapped)

Hybrid approach

- Registers are accessed as I/O ports
- Buffers are memory mapped
- Used by the Pentium (640K-1M mem-mapped buffer, 0-64K ports)

Accessing the Controller



Accessing the Controller

When a controller register has to be accessed

- CPU puts address on the bus
- CPU sets a line that tells if this address is a memory address or an I/O port
- In case the register/buffer is memory-mapped, the corresponding controller is responsible for checking the address and service the request if the address is in its range

What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers VS Port IO
- Direct Memory Access
- Interrupts and interrupt handling
- Precise VS Imprecise interrupts
- IO software AKA Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

Memory-Mapped I/O

Advantages

- Does not require special instructions to access the controllers
- Protection mechanisms can be achieved by not mapping processes' virtual memory space onto I/O memory

Disadvantages

- Caching would prevent correct interaction (hardware must provide a way to disable caching) How so?
- If the bus connecting the CPU to the main memory is not accessible to the device controllers, the hardware has find a way to let controllers know which addresses have been requested

x86 PDEs and PTEs

31–12	11–10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]	Ignored						PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame; Reserved (must be 0); Bits 39:32 of address[2]; PAT	Ignored		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table	Ignored			0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
											0	PDE: not present
Address of 4KB page frame	Ignored		G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored											0	PTE: not present

D … Caching disabled

Why Caching is Bad for MMIO

Reads can't come from the cache

- Register value can change unbeknownst to the cache

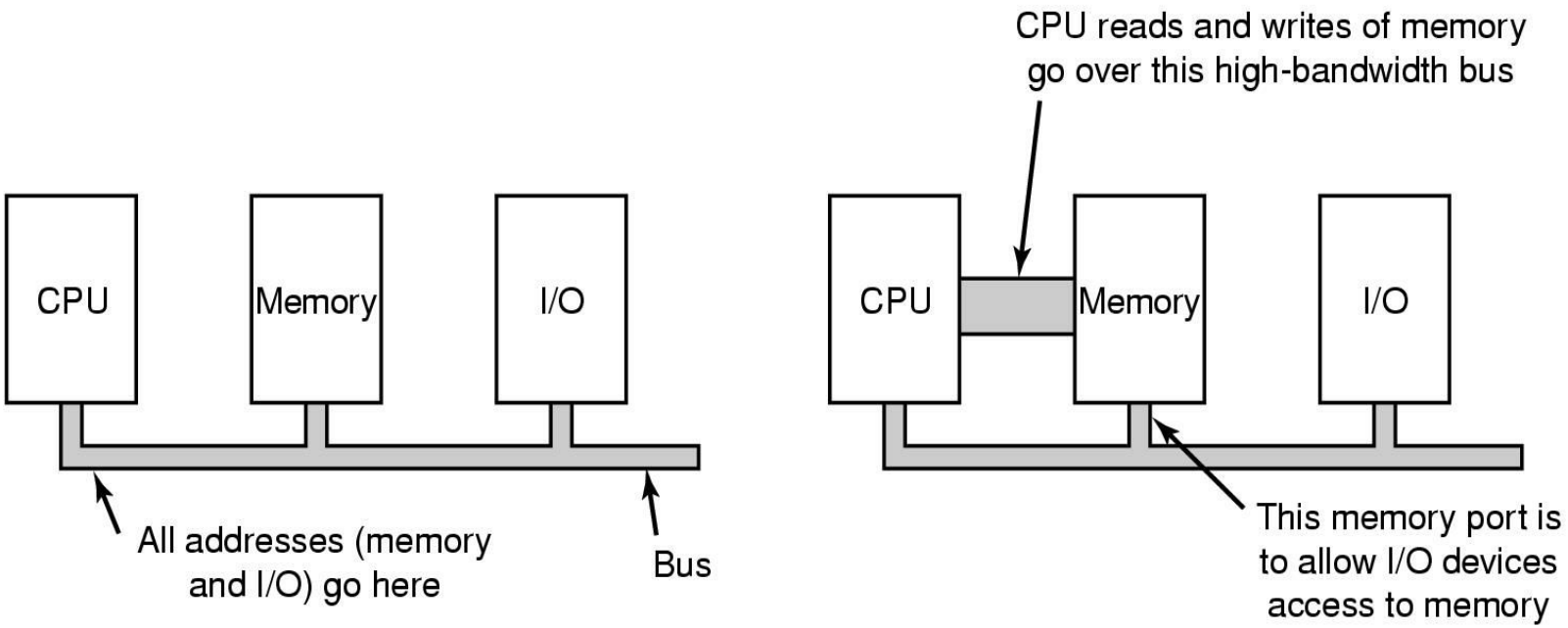
Write-back caches (and write buffers) cause problems

- You don't know when the line will be written

Reads and writes cannot be combined into cache lines

- Registers might require single word or byte writes only
- Line-size writes stomp on other registers
- Even spurious reads trigger device state changes

Memory-Mapped I/O



Port I/O vs. MMIO

Port I/O

- Need to have extra CPU instructions
- Not generally accessible from C/C++ code (must use assembly)
- with x86-64 restricted registers, and no 64-bit transfer; not used much servers

Memory mapped I/O

- Sacrifice some physical address space
- Can interact badly with memory caching
- Each address must be checked to see if it is I/O

What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers VS Port IO
- Direct Memory Access
- Interrupts and interrupt handling
- Precise VS Imprecise interrupts
- IO software AKA Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

Direct Memory Access (DMA)

Both port and mm I/O have a downside:

- Require the CPU to run for every piece of data transferred
- Reading/writing one word at a time may waste CPU time

Instead, we can ask the device to do a bulk transfer directly to memory

This transfer goes directly from the device to RAM, so the CPU can run (other processes) concurrently

DMA Controller

A DMA controller supports "automatic" transfer between device (controllers) and main memory

The DMA controller

- Has access to the device bus and to the memory
- Has a memory address register, a count register, and one or more control register (I/O port to use, direction of transfer, etc.)
- i.e., the CPU configures the DMA controller to indicate what device to transfer *from*, where to transfer *to*, *how much* data

A DMA controller can be associated with each device or can be one for all the devices

Reading with Direct Memory Access

The CPU

- Loads the correct values in the DMA controller
- Sends a read operation to the device controller

The DMA controller

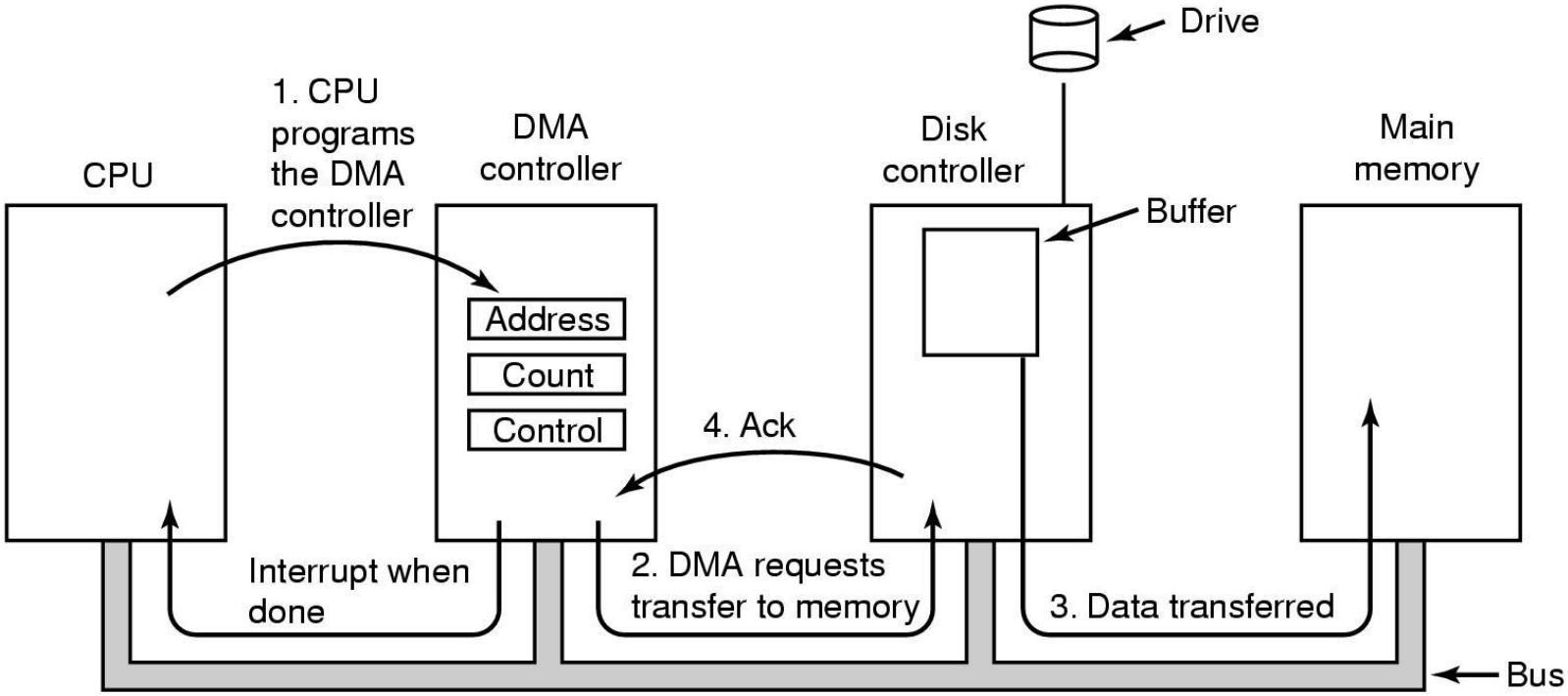
- Waits for the operation to complete
- Sets the destination memory address on the bus
- Sends a transfer request to the controller

The device controller

- Transfers the data to memory
- Sends an ACK signal when the operation is completed

When the DMA has finished it sends an interrupt to the CPU

Direct Memory Access (DMA)



DMA Schema Variations

Cycle stealing

- DMA acquires the bus competing with the CPU for each word transfer

Burst mode

- DMA tells the controller to acquire the bus and issue a number of transfers

The DMA may ask the controller to transfer data to a buffer on the DMA controller and then perform the actual transfer to memory

Real World DMA

- Typically DMA controllers can handle more than one transfer at a time (multi-channel)
  - If so, note that now the DMA controller must implement something like a scheduler!
- Sometimes there is not a single DMA controller, but rather individual peripherals each can do transfers directly to RAM (bus mastering)
  - Note that this can sometimes cause bus contention as devices can no longer coordinate transfers
- Some systems support peer-to-peer DMA – direct channel between two peripheral devices

What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers VS Port IO
- Direct Memory Access
- Interrupts and interrupt handling
- Precise VS Imprecise interrupts
- IO software AKA Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

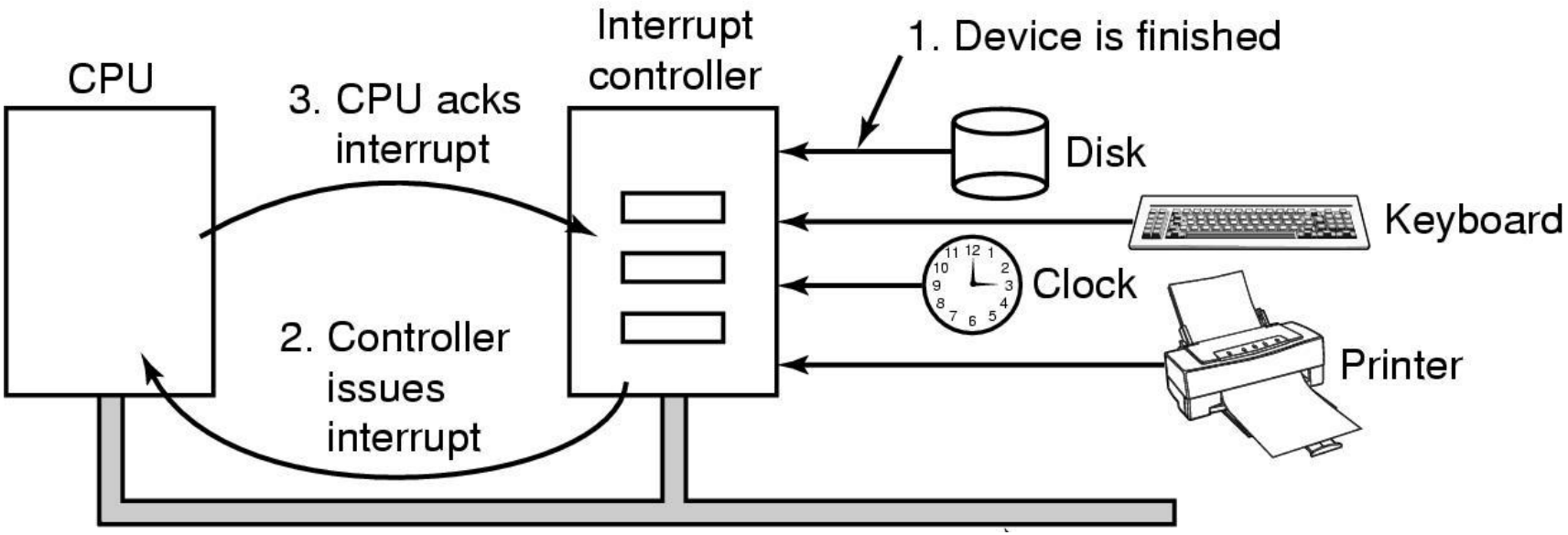
Interrupts

- When hardware devices need attention for any reason (e.g., done with some work), they raise an interrupt
- Interrupts are implemented by asserting a signal on a bus line, which is detected by the *interrupt controller*
- Interrupt controller queues interrupts and delivers them to the CPU
- CPU handles each one and then tells the interrupt controller it has acknowledged the interrupt

Interrupt Handling

- When the CPU is notified of an interrupt, it stops whatever it's doing; needs to save some state so that it can come back to it later, options: more later
- The interrupt will come with a number, which is used to index into a table (the interrupt vector table) to find the address of the interrupt handler
- The interrupt handler will acknowledge the interrupt, allowing the controller to deliver the next one

Interrupts


CPU
3. CPU acks interrupt
Interrupt controller
1. Device is finished
Disk
Keyboard
Clock
Printer
2. Controller issues interrupt

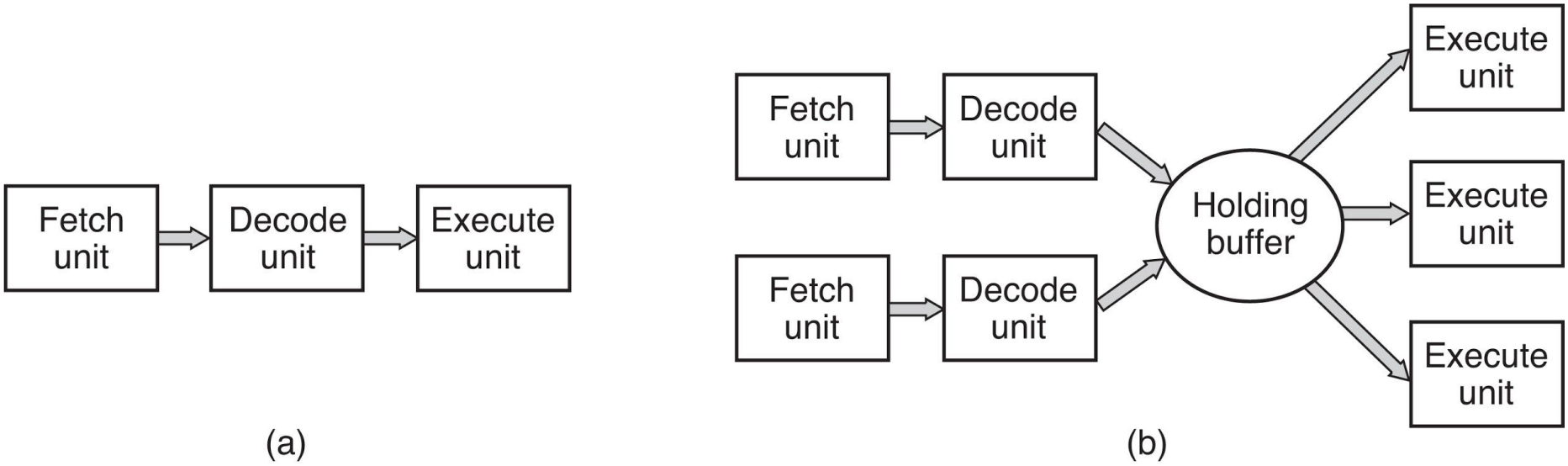
What we will cover

- I/O devices and hardware overview.
- I/O Controllers
- Memory Mapped IO registers VS Port IO
- Direct Memory Access
- Interrupts and interrupt handling
- Precise VS Imprecise interrupts
- IO software AKA Device Drivers
- Disk Drives and SSDs
- Clocks and Timers

Interrupt Precision

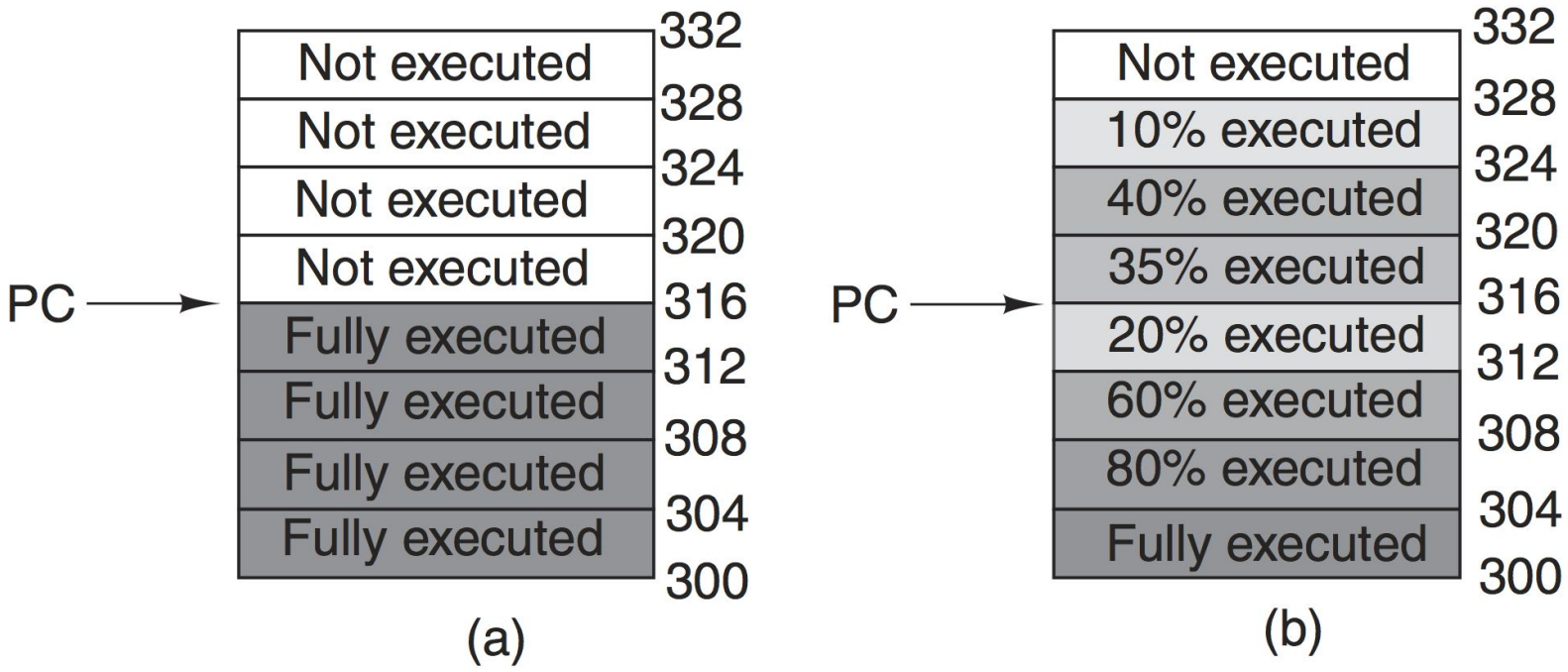
- Modern CPUs do a lot of things at once
  - Speculative execution, out of order processing, pipelining
- When an interrupt occurs, we may have multiple partially completed instructions pending
- All of that has to stop to handle the interrupt, and this may leave things in a weird, half-completed state

What's the problem



Processors have complex pipelines, superscalar….

Precise vs. Imprecise Interrupts



Precise Interrupts

- We say an interrupt is precise if:
  - The program counter is saved somewhere
  - All instructions before the program counter have finished
  - All instructions after the program counter have not yet started
  - Execution state of the instruction at program counter is known
- If these are satisfied, it's easy for the OS to resume after an interrupt

Imprecise Interrupts

- If these properties are not satisfied, the interrupt is imprecise
- Typically in this case, the architecture's interrupt handling will provide lots of information about the half-completed state at every interrupt
- Then it's the OS's job to roll back any party finished instructions before resuming
  - Very unpleasant for the OS writer
- x86 provides precise interrupts, at the cost of much greater hardware complexity

Saving the CPU State

- The interrupt handler should save the current CPU state
- If registers are used, nested interrupts would overwrite the data and, therefore, the acknowledgment to the interrupt controller must be delayed
- If a stack is used, the information should be stored in a portion of memory that will not generate page faults; i.e., HW has to switch to a kernel stack

Restoring the CPU State

- Restoring is easier said than done when instructions may end up... half-baked (in case of pipelining)

- A *precise* interrupt leaves the machine in a well-defined state
  - The PC is saved in a known place
  - All instructions before the one pointed by the PC have been fully executed
  - No instruction beyond the one pointed by the PC has been executed
  - The execution state of the instruction pointed by the PC is known

- Restoring in case of imprecise interrupts requires a lot of information to be saved