EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical & Computer Engineering

Boston University

Slides from Christopher Kruegel, UCSB

Input and Output

Input/Output Devices

- The OS is responsible for managing I/O devices
  - Issue requests
  - Manage 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 OSes 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=1722041,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 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

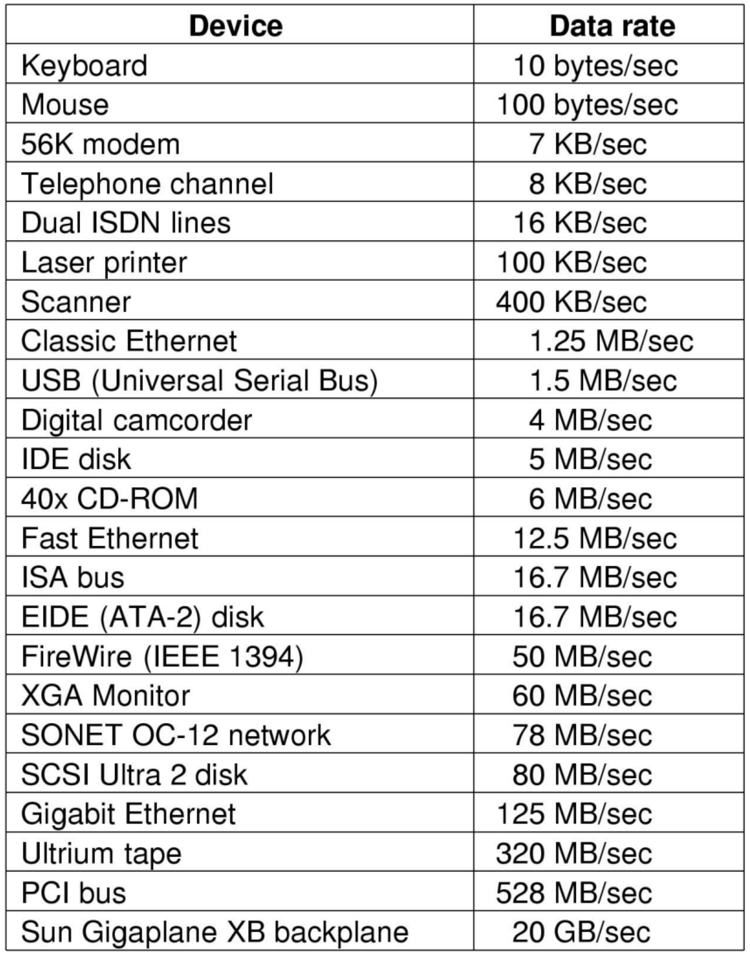
Classification not perfect

- Clocks/timers
- Graphics cards
- 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

Device Data Rates



Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

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)

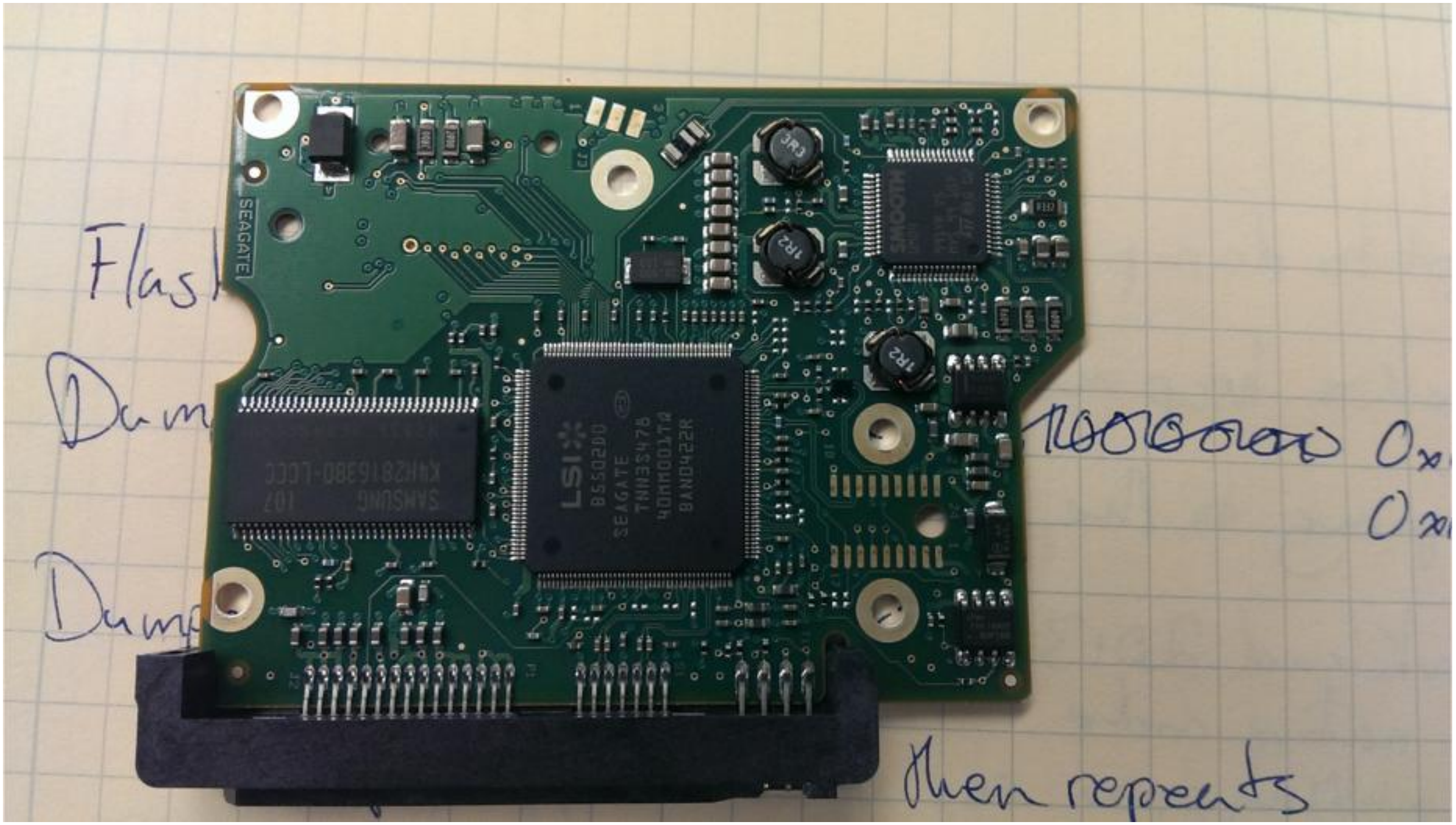
Controller's tasks

- 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
  - Some permanent storage (usually flash)

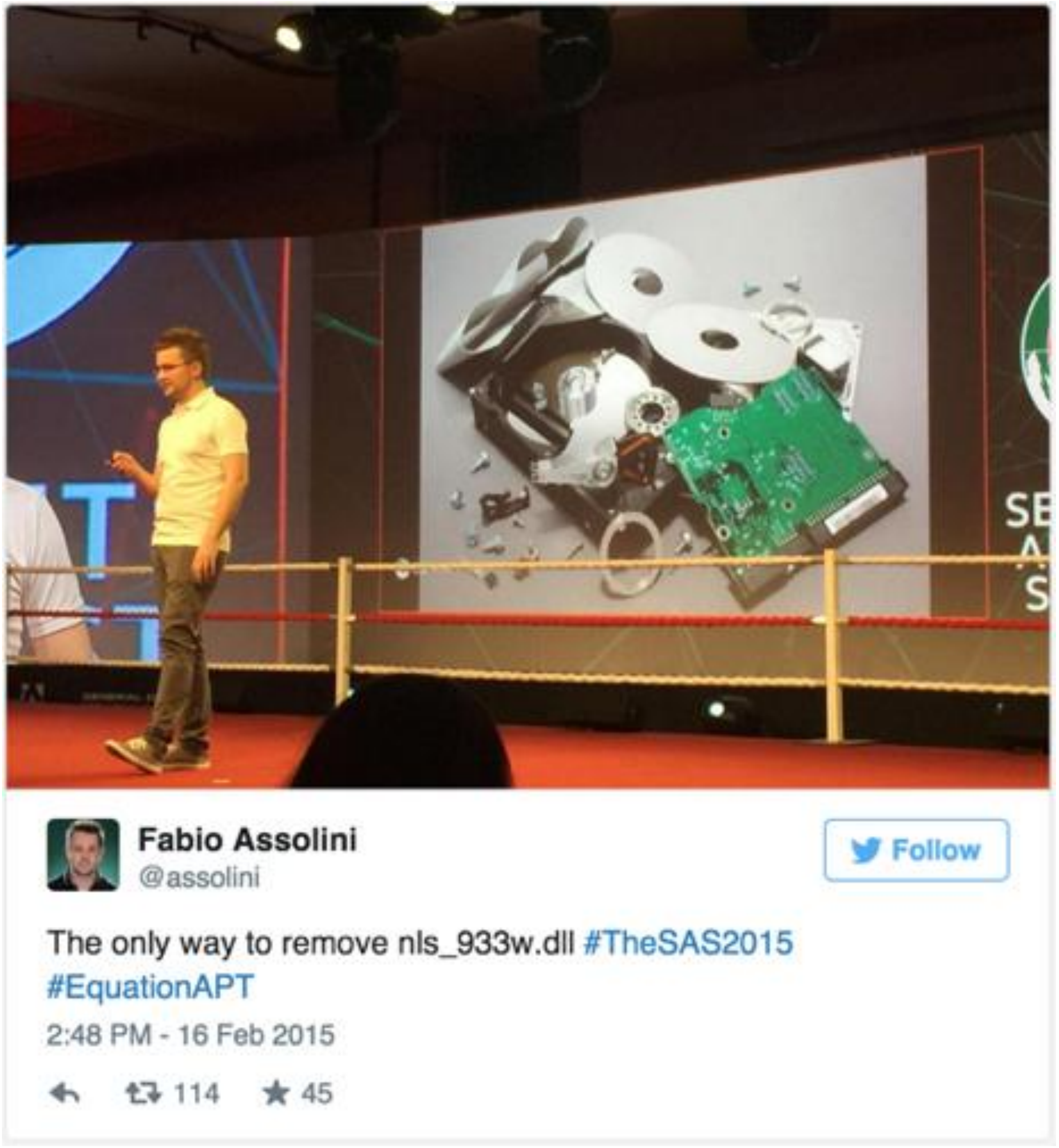
Hard Drive Controller



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

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

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

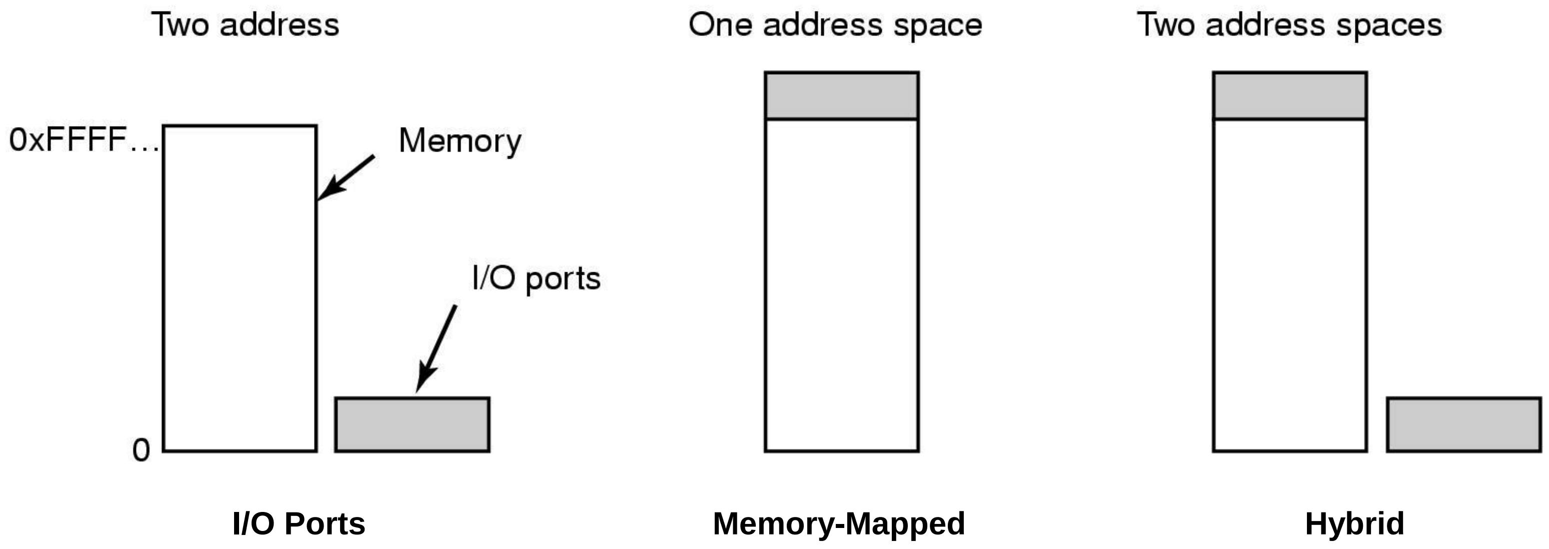
- Registers mapped to I/O ports
- `IN REG, PORT` and `OUT REG, PORT`
  transfer data from CPU's registers to a controller's registers

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

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 to find a way to let controllers know which addresses have been requested

x86 PDEs and PTEs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	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

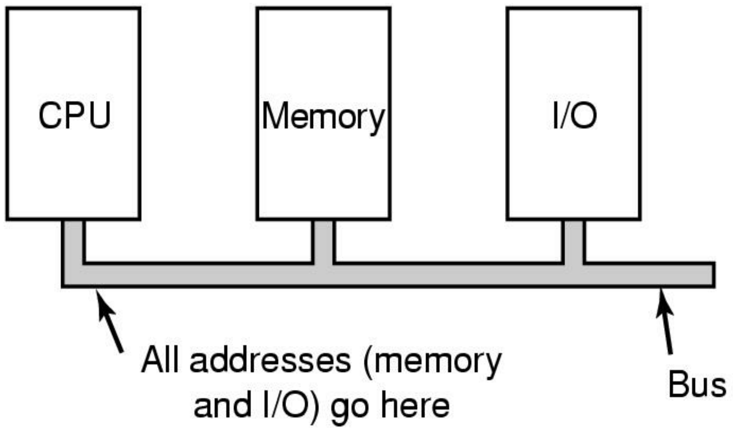
Write-back caches (and write buffers) cause problems

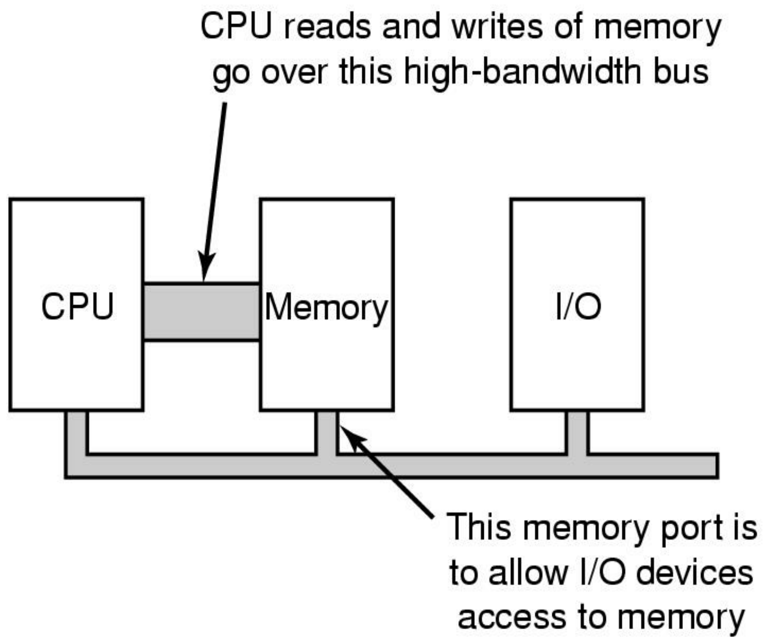
- You don’t know when the line will be written

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 can trigger device state changes

Memory-Mapped I/O




CPU reads and writes of memory go over this high-bandwidth bus
CPU
Memory
I/O
This memory port is to allow I/O devices access to memory

Port I/O vs. MMIO

Port I/O

- Need to have extra CPU instructions
- Not generally accessible from C/C++ code (must use assembly)

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

Direct Memory Access (DMA)

Direct Memory Access (DMA)

Both port and MMIO 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

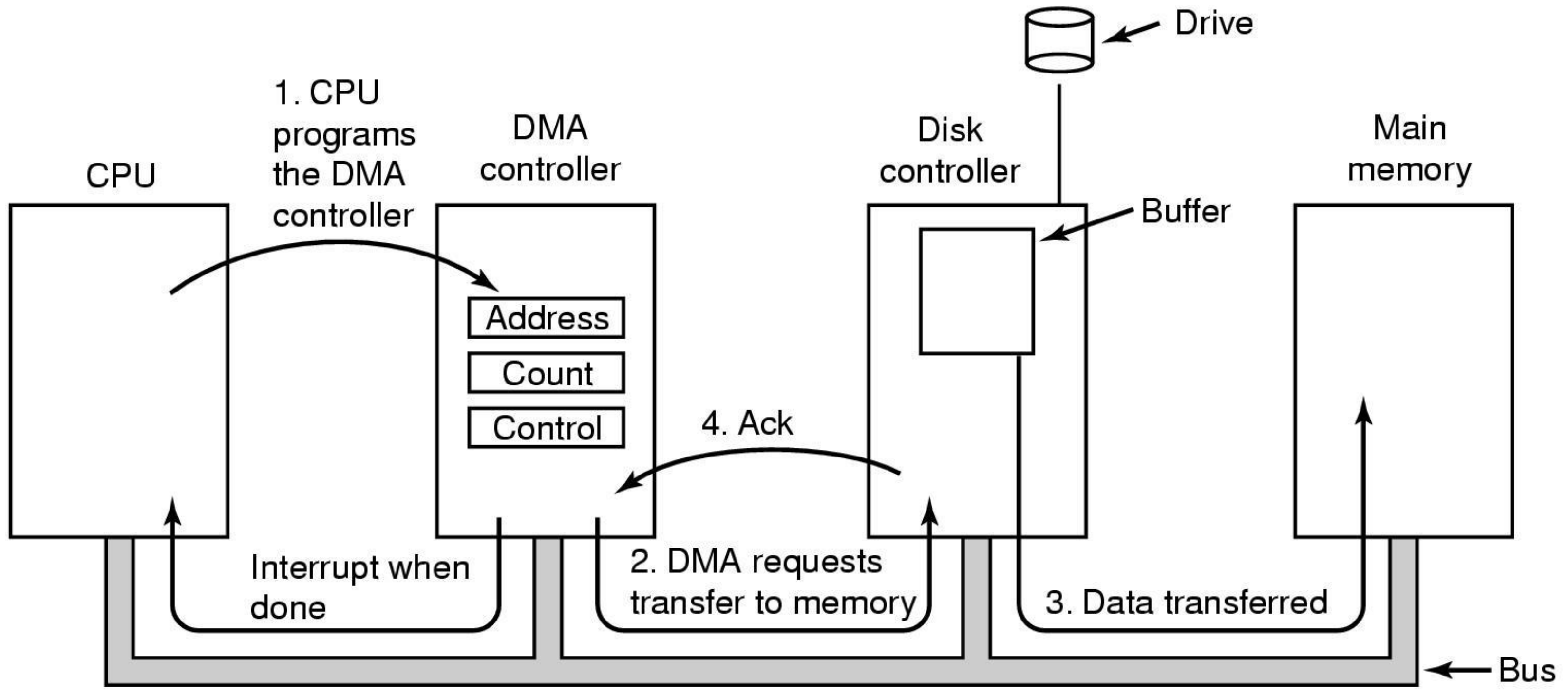
- Waits for the operation to complete
- Sets the destination memory address on the bus
- Sends a transfer request to the controller

The device controller

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

When the DMA has finished the DMA controller sends 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

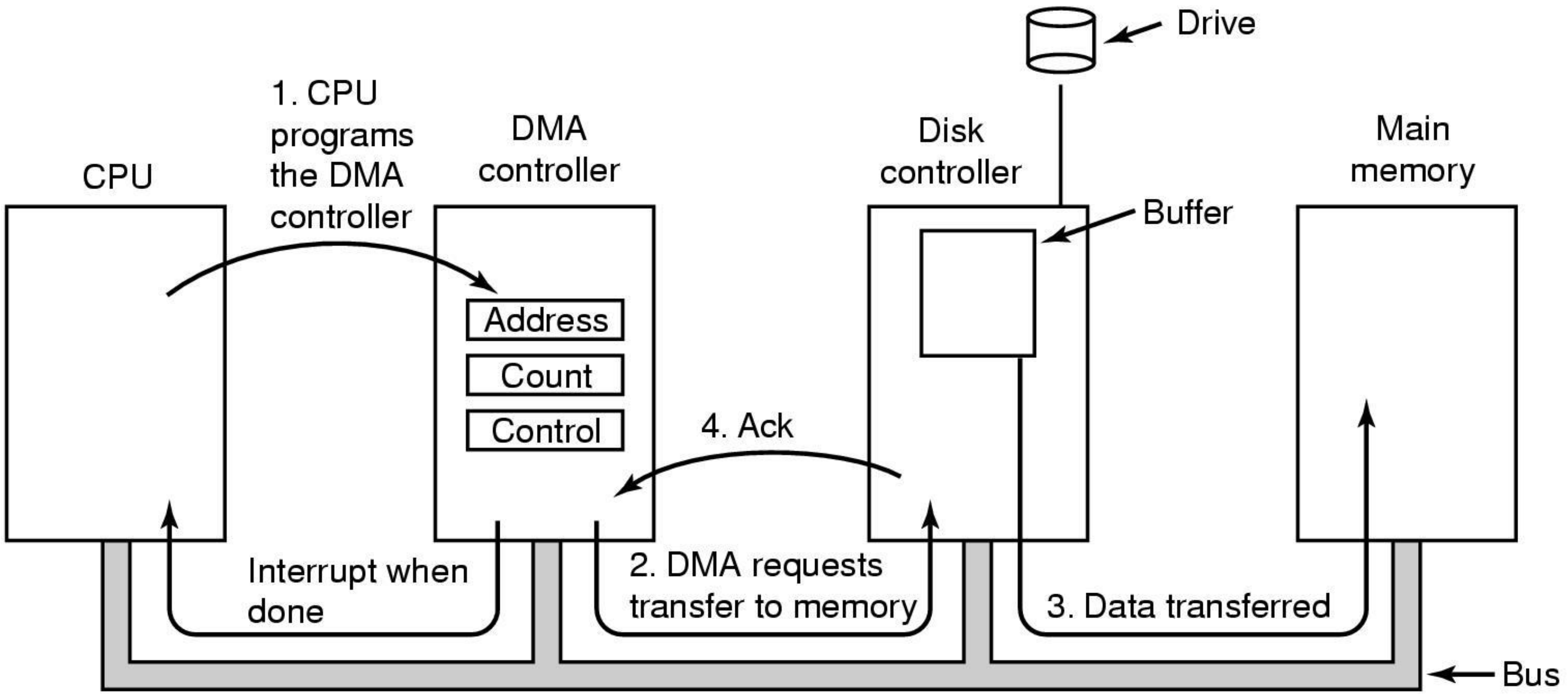
The DMA may ask the controller to transfer data to a buffer on the DMA controller and then perform the actual transfer to memory

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

Last Time ...

Direct Memory Access (DMA)



DMA Security Problems

- Firewire & Thunderbolt …
  - DMA devices read/write to physical memory *directly* w/o involvement of the CPU
  - Firewire & Thunderbolt devices are DMA capable
  - Maybe we should not trust external devices to behave normal/benign
- Solution:
  - IOMMU (Input/Output Memory Management Unit)
  - Instead of allowing devices to write to any physical address, constrain where in phys mem devices can write
  - Translates IO-virtual to physical addresses

Interrupts

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

Interrupts vs. Exceptions

Exceptions also signal the CPU that special attention is needed, so what’s the difference?

- Exceptions:

  Synchronous (e.g., div-0, page fault, etc.)

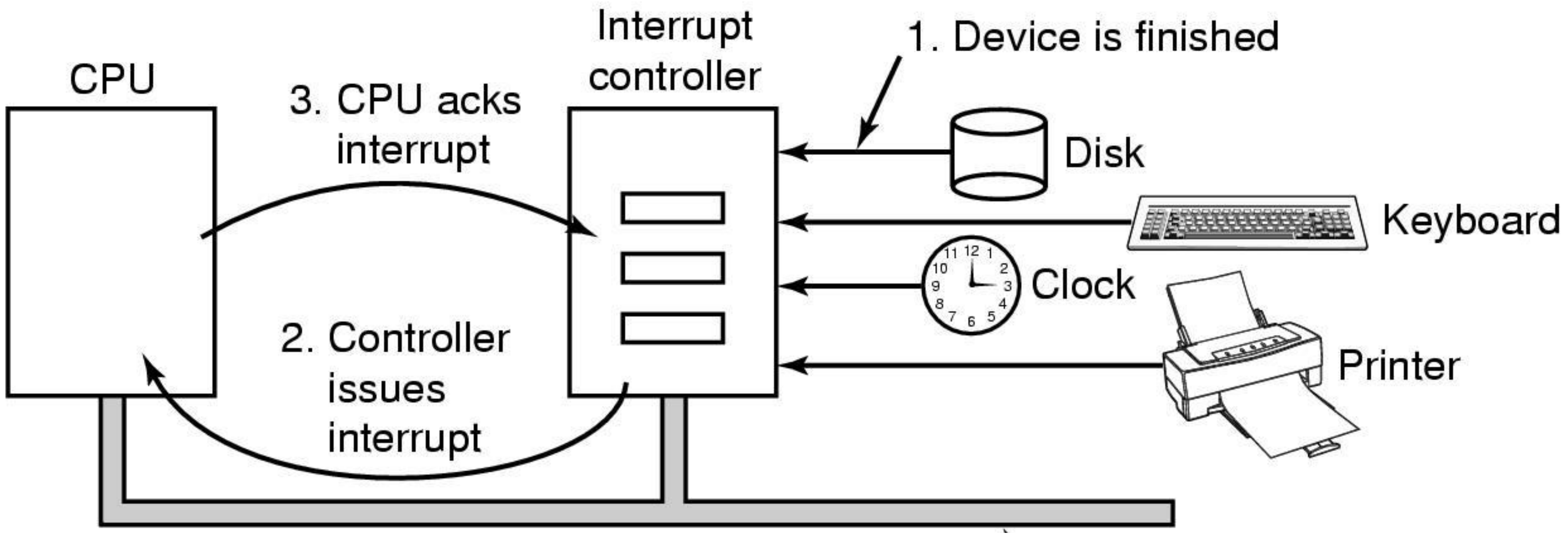
- Interrupts:

  Asynchronous (e.g., user hits keyboard, DMA transfer completed, etc.)

Interrupt Handling

- When the CPU is notified of an interrupt, it stops whatever it’s doing
  - At this point it needs to save some state so that it can come back to it 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

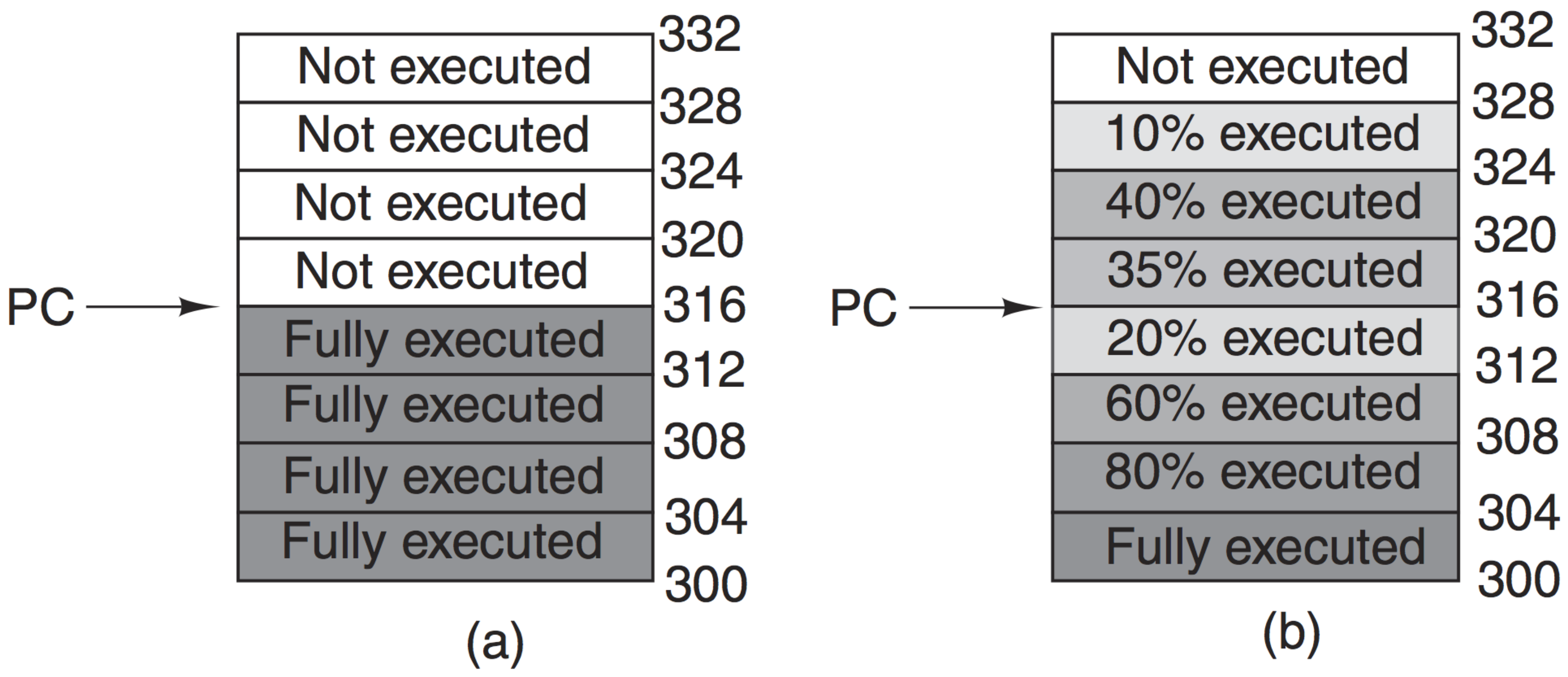
Interrupts


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

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

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 partly 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 why?

Restoring the CPU State

- Restoring is easier said then 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

I/O Software

Goals of I/O Software

Device independence

- Programs can access any I/O device without specifying device in advance (reading from floppy, hard drive, or CD-ROM should not be different)

Uniform naming

- Name of a file or device should not depending on the device

Error handling

- Errors should be handled as close to the hardware as possible

Synchronous vs. asynchronous transfers

- User program should see blocking operations even though the actual transfer is implemented asynchronously

Buffering

I/O Software

System call in user-space

Data is copied from user space to kernel space

I/O software can operate in several modes

- Programmed I/O
  - Polling/Busy waiting for the device
- Interrupt-Driven I/O
  - Operation is completed by interrupt routine
- DMA-based I/O
  - Set up controller and let it deal with the transfer

Programmed I/O

Syscall handler/Driver:

```
copy_from_user(buffer, p, count);                 /* p is the kernel bufer */
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

Syscall handler/Driver:

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

Interrupt handler:

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

Interrupt-Driven I/O

- Instead of polling while waiting for hardware to be ready, we could ask the hardware to tell us via an interrupt
- Now we can go do other things while we wait for the hardware to finish
- This can make a system much more responsive if the device is slow

I/O Using DMA

Syscall handler/Driver:

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

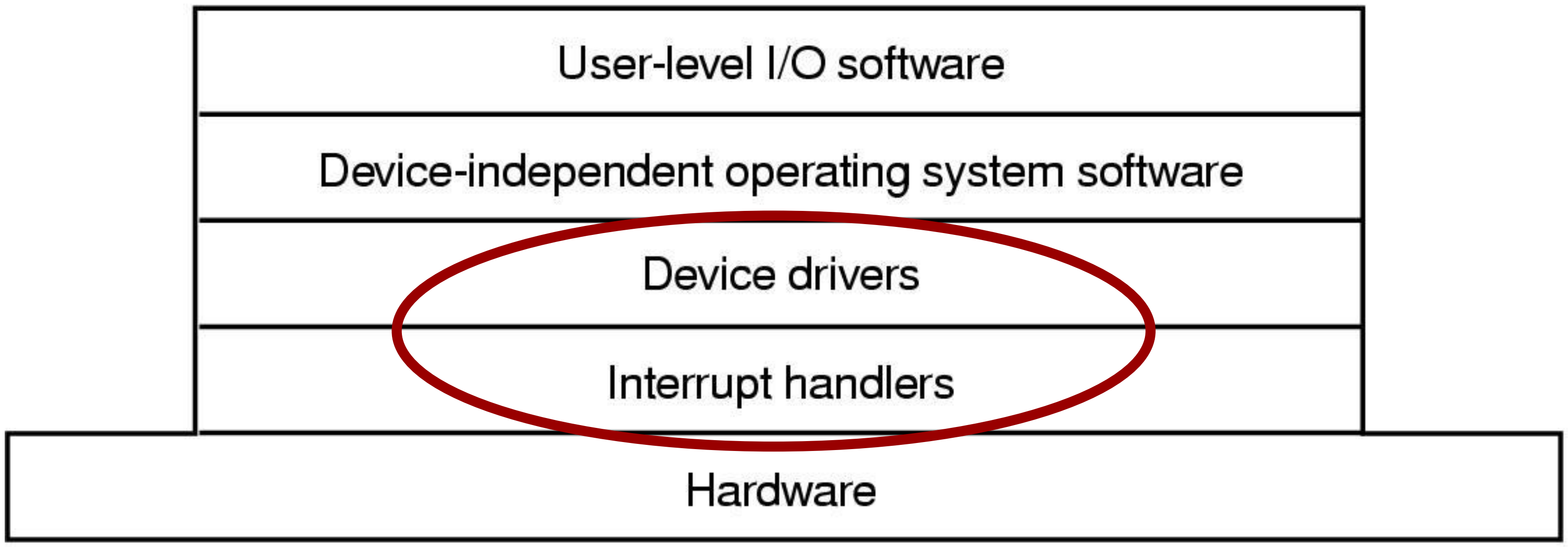
Interrupt handler:

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

I/O Using DMA

- This is essentially an extension of interrupt-driven I/O
- Instead of interrupting every time a piece of data is ready, program DMA controller for a bulk transfer
- Advantage over plain interrupt-driven access is that if your data is large, you get just one interrupt rather than many

I/O Handling – Architecture



Device Driver & Interrupt Handler

- Device driver starts I/O and then blocks (e.g., p->down)
- Interrupt handler does the actual work and then then unblocks driver that started it (e.g., p->up)
- Mechanism works best if device drivers are threads in the kernel

Interrupt Handlers

- Conceptually simple – just do what's necessary to handle the interrupt and then resume execution
- Reality is more complicated …

Interrupt Handlers

- Save registers not already saved by interrupt hardware
- Set up context for interrupt service procedure (TLB, MMU)
- Set up stack for interrupt service procedure
- Acknowledge interrupt controller, re-enable interrupts
- Copy registers from where saved to process table
- Run service procedure
- Decide which process to run next
- Set up MMU context for process to run next
- Load new process' registers
- Start running the new process

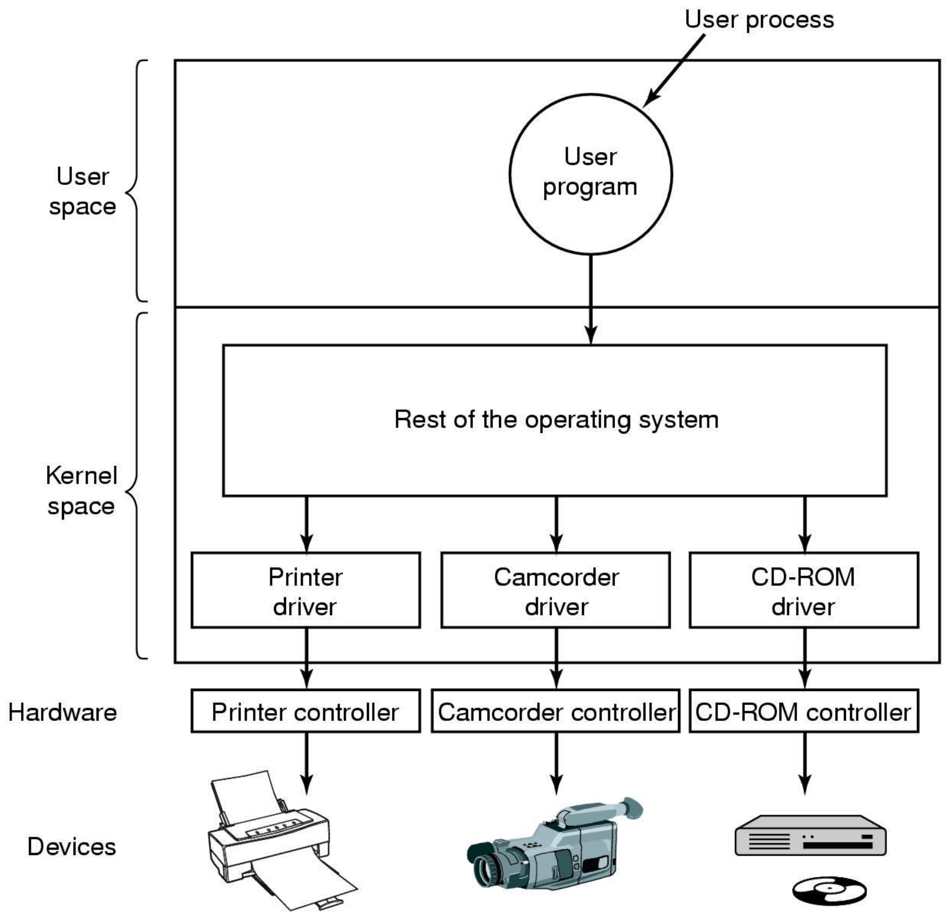
Interrupt Handler Organization

- If the rate of interrupts is high and we take a while to service each one, we may fall behind
- To avoid this, interrupt handlers are often written to do the minimum possible work needed to acknowledge the interrupt
- They then queue the remaining work to do later (with interrupts enabled)
- In Linux, interrupt acknowledgement is called the *top half*, and the remaining work happens in the *bottom half*

Device Drivers

- A device driver is a specific module that manages the interaction between the device controller and the OS
- Device drivers are usually provided by the device manufacturer (or by frustrated Linux users!)
- Device drivers are usually part of the kernel
  - compiled and linked in
  - loadable modules
- Usually provide a standard API depending on the type of device
  - Character
  - Block
- Device drivers are frequently the source of kernel problems

Device Drivers



Device Driver’s Tasks

- Device initialization
- Accept read-write request from the OS

  i.e., take commands from higher levels in the OS and translate them into hardware requests
- Start the device if necessary (e.g., start spinning the CD-ROM)
- Check if device is available: if not, wait
- Wait for results
  - Busy wait (awakened by interrupt)
  - Block
- Check for possible errors
- Return results
- Power management – put the device to sleep when it’s not being used

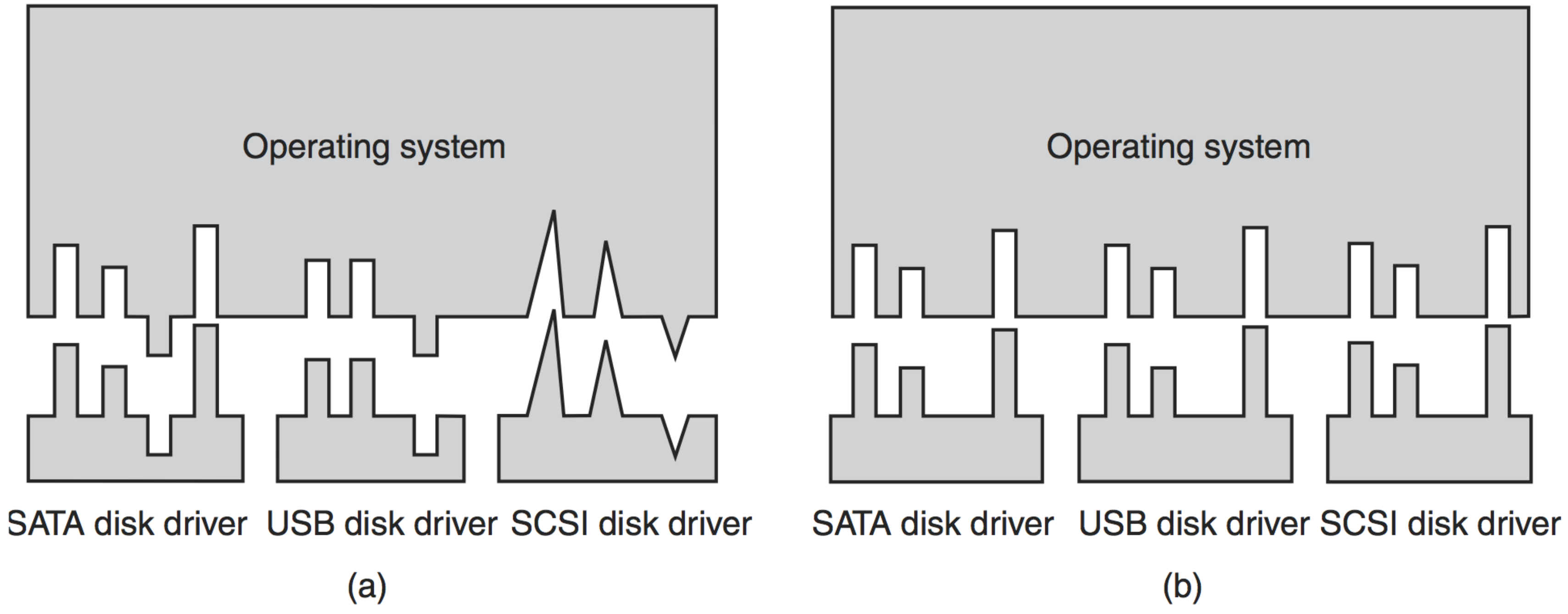
Device Driver Considerations

- Drivers may be interrupted while working, and the interrupt may call into the same driver
  - So drivers must be written to be *reentrant* – expect that it can be called again before finishing its first task
- Because hardware may be hot-pluggable (e.g., USB devices), drivers may get loaded and unloaded throughout the lifetime of a system

Driver APIs

- Because it’s convenient not to write drivers for every piece of hardware yourself, OS creators typically specify a driver API
- This API specifies a well-defined model for the OS to use when interacting with the driver, and says what kernel functions are available for driver use
- So, if your OS is popular enough, hardware manufacturers will write drivers for you!

Driver APIs


Operating system
SATA disk driver
USB disk driver
SCSI disk driver
(a)
Operating system
SATA disk driver
USB disk driver
SCSI disk driver
(b)

Generic I/O Layer

- Above the driver level, there may be a common layer for handling I/O issues common to multiple drivers
  - Buffering I/O requests
  - Generalized error reporting
  - Provide uniform block size

Buffering

Where do we store the buffer?

- In user space?

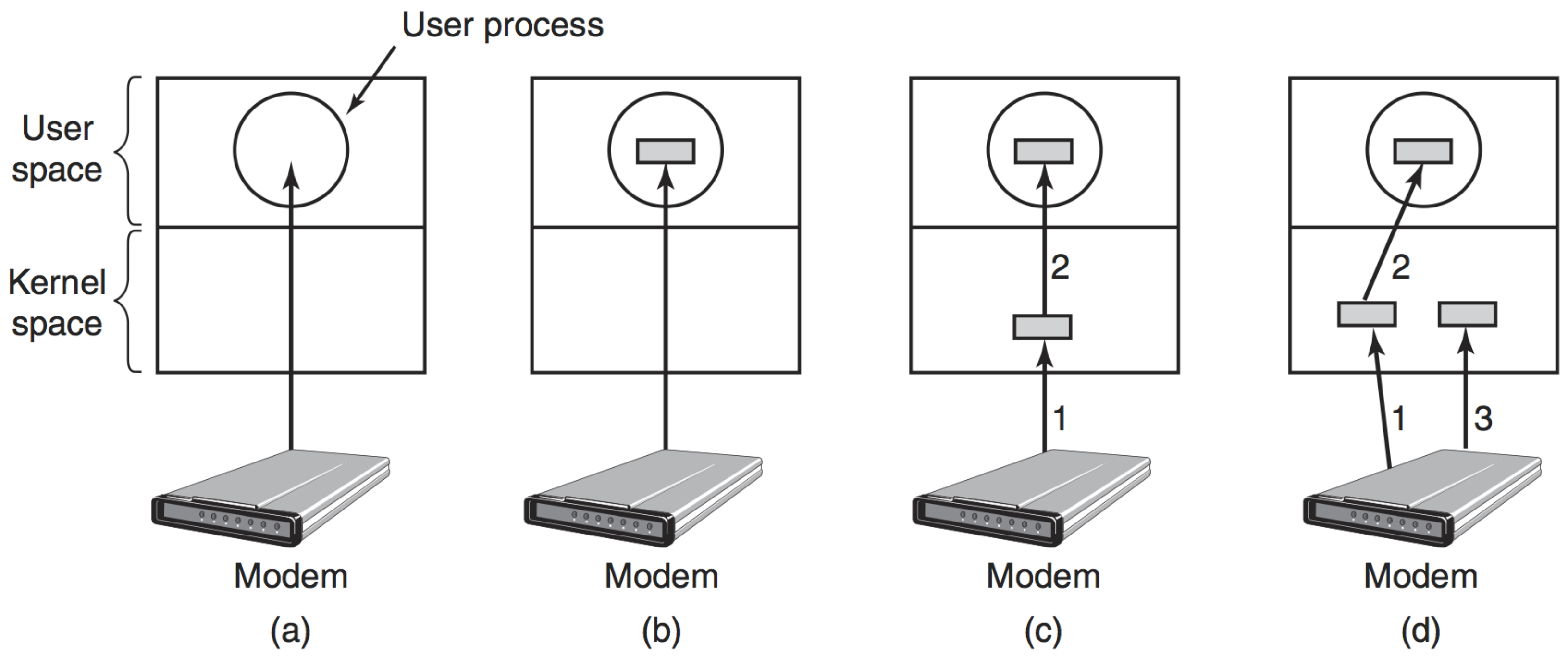
  No – might have to swap out user page, but I/O has to go somewhere

- In kernel space?

  Better – but now what happens when the kernel needs to copy things to the user, but data is still coming in?

- In kernel space with *two* buffers – double buffering

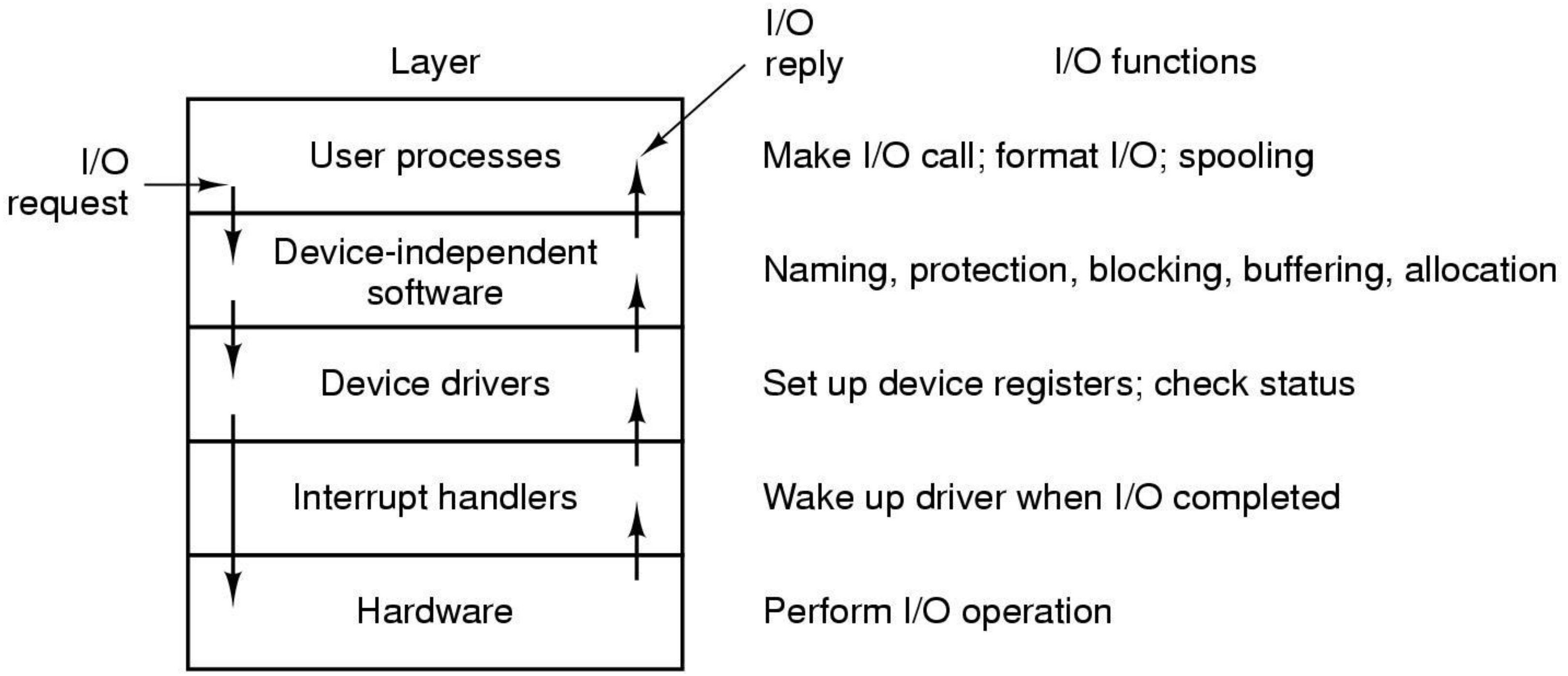
Buffering Example



Buffering Performance

- If there are too many layers of buffering between the hardware and the user program, performance suffers
- Some operating systems try to minimize the number of copy operations ("Zero-copy I/O"), or number of user/kernel transitions
- How do we copy data from one file descriptor to another?
  - Series of read/write system calls
  - Alternative: sendfile API, which copies data between two file descriptors
    - Because descriptors are used, both src and dest are in the kernel and we can avoid copying to/from user land
    - Windows has similar TransmitFile

User-Space I/O Software

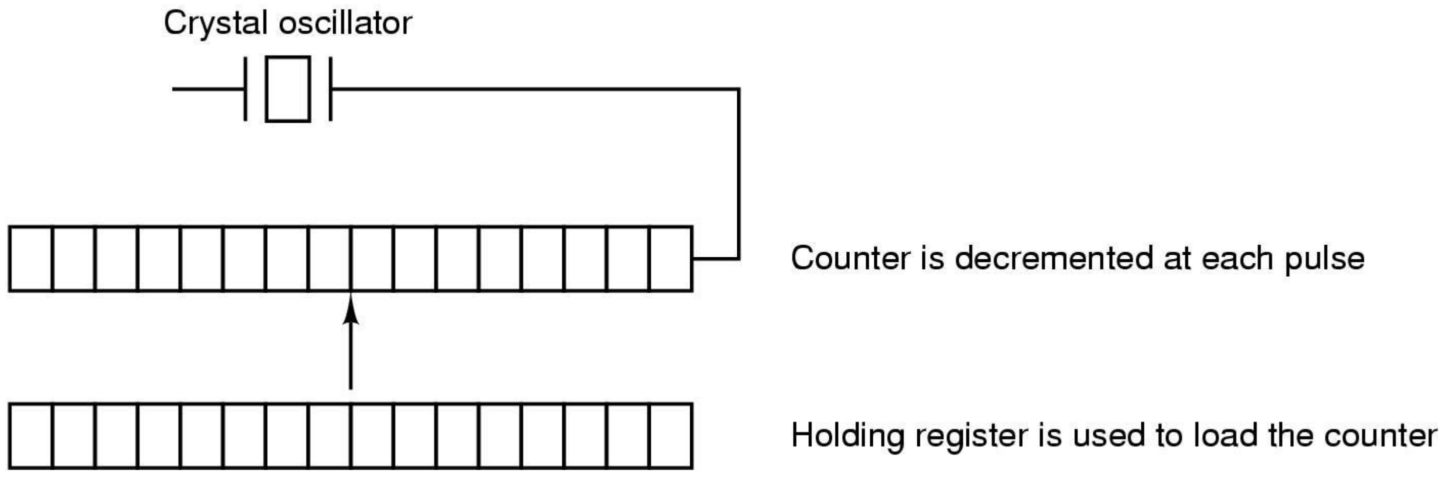

Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Specific Devices

- Let’s look at some interesting hardware devices
  - Keyboard
  - Timer

Clocks

- The clock is a fundamental device
- The counter is initialized with a OS-defined value
- The hardware decrements the counter with a certain frequency (e.g., 500 MHz)
- When the counter reaches 0 an interrupt is sent and the start value is restored


Crystal oscillator
Counter is decremented at each pulse
Holding register is used to load the counter

Clock Driver

- Maintains the time of day
- Checks processes’ CPU quantum usage
  - Calls the scheduler if quantum expired
- Does accounting of CPU usage and profiling of the system
- Handles alarms
  - Alarms are maintained in a list and fired whenever they expire

Timers

- Timers keep the OS running – we have seen how they allow preemptive scheduling to work
- Generally, implemented in hardware using a quartz crystal oscillator
- Can be programmed for periodic interrupts or one-shot (fire once and then disable)

Watchdog Timer

- Act as a “dead man’s switch”
- If the OS doesn’t write to the timer every interval, reset the system
- This is more common in embedded systems, where you don’t want a hang to disrupt things, but a reset every so often might be okay

Keyboards

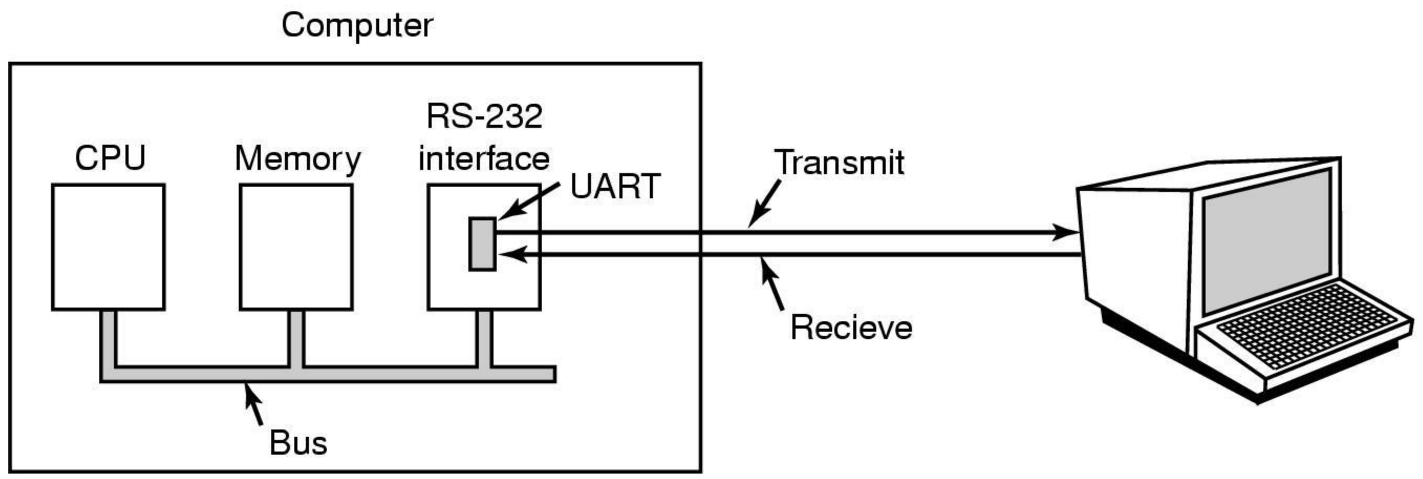
- Simple device!
- Each keypress generates an interrupt
- Information about which key it was can be read out using port I/O
- Why is raising an interrupt for every key good enough?

Character Oriented Terminals

- Simplest form of user-interaction
- A terminal is composed of a keyboard and a screen
- Characters typed from the keyboard are sent to the driver
- Characters sent by the driver are displayed on the screen
- Different modes of operation
  - Raw (non canonical): characters are passed by the driver to the user process as they are typed
  - Cooked, line-oriented (canonical): the drivers performs line-by-line processing before passing the line to the user process
- Drivers maintain buffered input/output and process special characters

RS-232 Terminal Hardware

- An RS-232 terminal communicates with computer 1 bit at a time (serial line)
- Bits are reassembled into characters by the UART (Universal Asynchronous Receiver/Transmitter)
- Windows uses COM1 and COM2 ports, UNIX uses /dev/ttyN
- Computer and terminal are completely independent

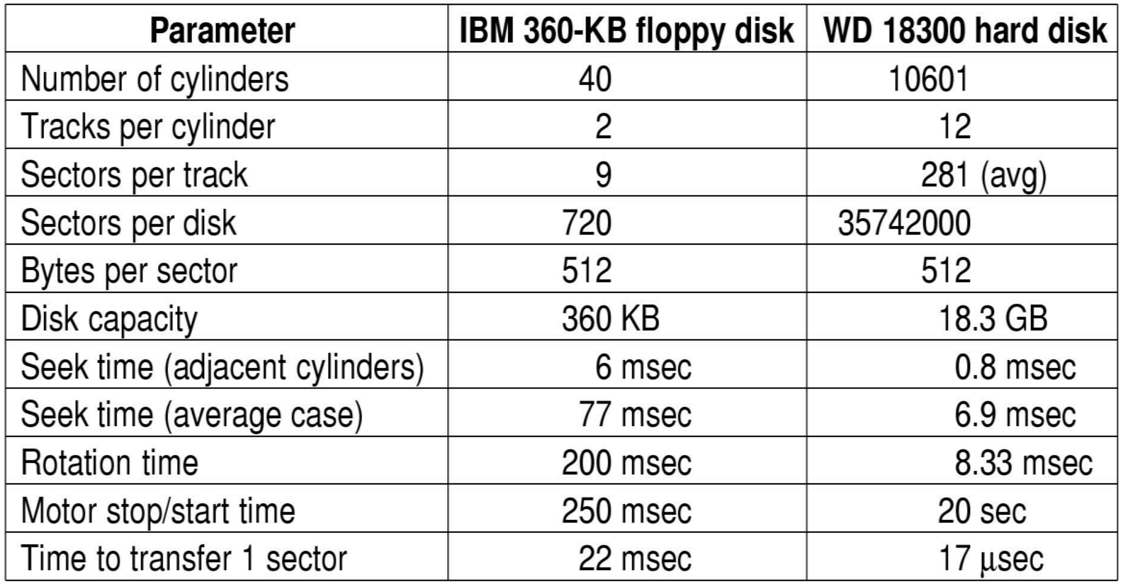


Disk

- Most important and commonly used device
- Used for secondary memory (swap space, file system)
- Different types:
  - Magnetic (floppy, hard disk)
  - Optical (CD-ROM, DVD)

Magnetic Disks

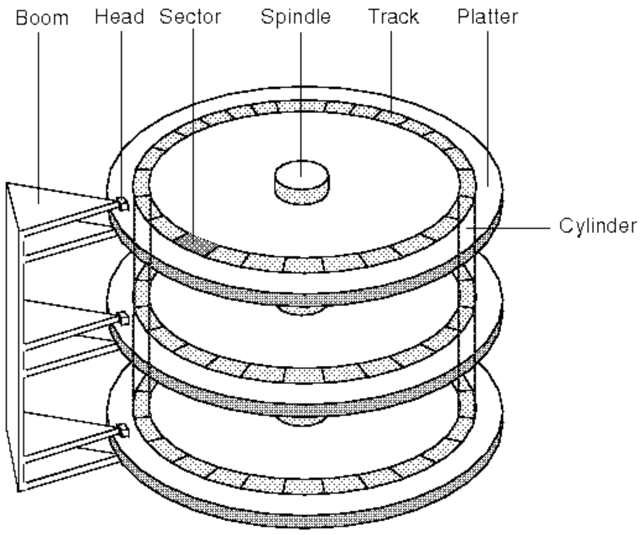
- Disk "geometry" specified in terms of
  - Cylinders composed of tracks (one per head)
  - Tracks composed of sectors
  - Sectors composed of bytes



Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

Disk Architecture

- Hard disk
  - several platters – disks (heads)
  - each platter has multiple tracks (start with 0)
  - each track has multiple sectors (start with 1)



Disk Architecture

Addressing sectors (blocks)

- CHS (cylinder, head, sector) triple
  - old disks use 10 bits for cylinder, 8 bits for head, 6 for sector
  - limits maximum disk size to ~ 8.4 GB

- Logical block address (LBA)
  - decouples logical and physical location
  - specifies 48 bit logical block numbers
  - allows controller to mask corrupt blocks

Disk Architecture

Disk Interfaces

between controller (motherboard) and disk

- ATA (AT Attachment)
  - 28 bit addresses (~128 GB maximum size)
  - 40 pin cables, 16 bit parallel transfer (single-ended signaling)
  - 2 devices (master and slave) can be attached to connection cable
  - ATA-3 introduced security features (passwords)
- Serial ATA (SATA)
  - 8 pin cables
  - higher data transfer (differential signaling)

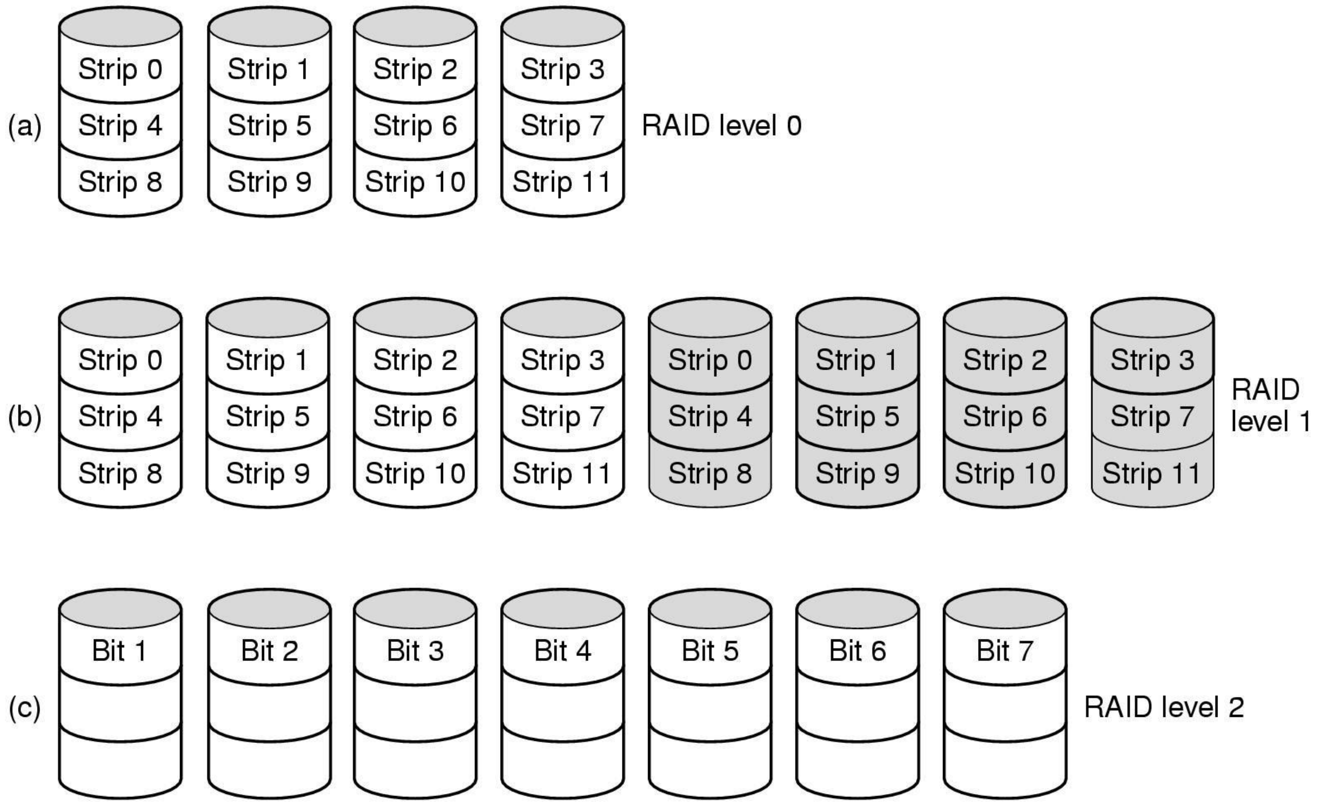
RAID

- Redundant Array of Inexpensive Disks vs.
  Single Large Expensive Disk (SLED)

- A set of disks is managed by a RAID controller
- Different RAID modes (called “levels”)

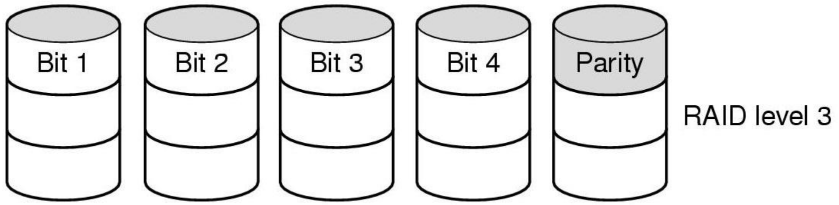
- RAID 0
  - Disks are divided into strips of k sector each
  - Strips are allocated to disks in a round-robin fashion
  - Request for consecutive strips can be carried out in parallel
- RAID 1
  - Striping + redundancy
- RAID 2
  - Striping at the word/byte level + ECC

RAID

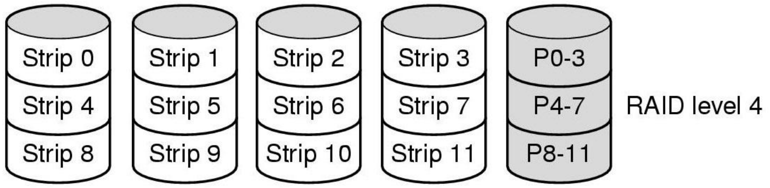

(a)
Strip 0
Strip 1
Strip 2
Strip 3
Strip 4
Strip 5
Strip 6
Strip 7
Strip 8
Strip 9
Strip 10
Strip 11
RAID level 0
(b)
Strip 0
Strip 1
Strip 2
Strip 3
Strip 0
Strip 1
Strip 2
Strip 3
Strip 4
Strip 5
Strip 6
Strip 7
Strip 4
Strip 5
Strip 6
Strip 7
Strip 8
Strip 9
Strip 10
Strip 11
Strip 8
Strip 9
Strip 10
Strip 11
RAID level 1
(c)
Bit 1
Bit 2
Bit 3
Bit 4
Bit 5
Bit 6
Bit 7
RAID level 2

RAID

- RAID 3
  - Parity word kept on a separate drive

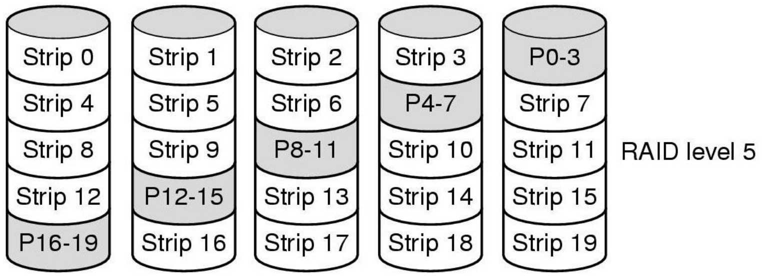


- RAID 4
  - Strip parity on extra drive (XOR of strip contents)



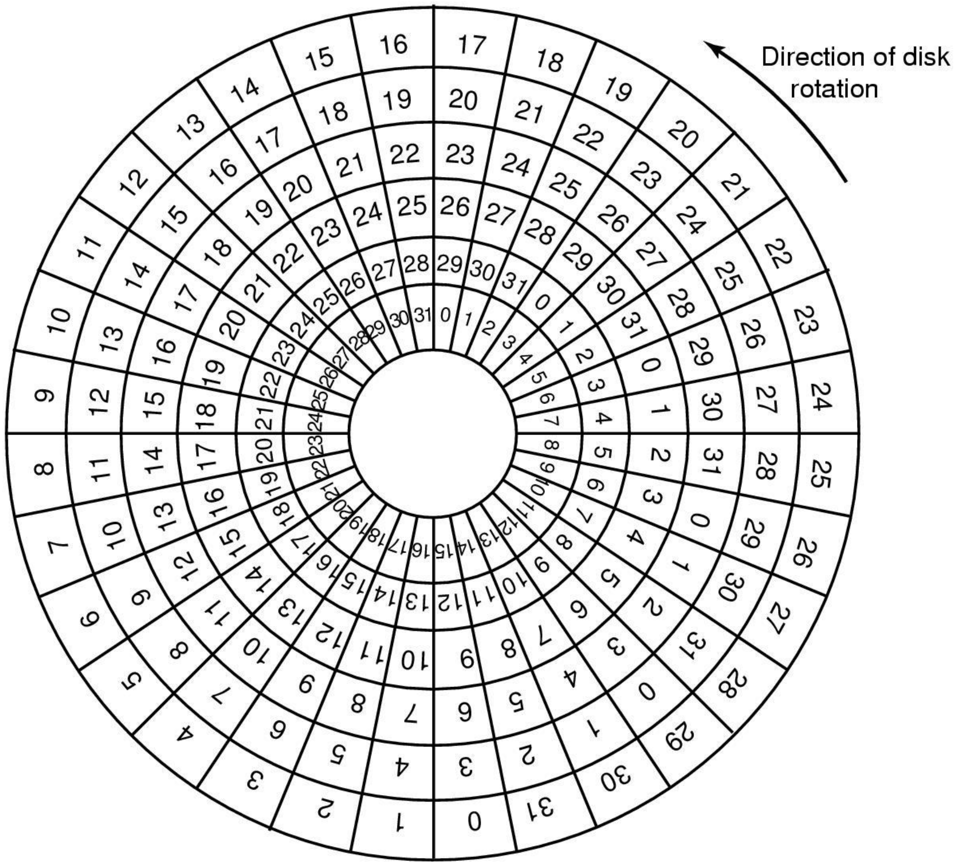
RAID

- RAID 5
  - Parity strips are distributed over the disks



Cylinder Skew

- The initial sector for each track is skewed with respect to the previous one
- This facilitates continuous reads across contiguous tracks by taking into account the rotation of the disk when the arm is moved
- 7,200 rpm with 360 sectors
- Cycle in 60/7,200 = 8,3msec
- Sector rate 8.3msec/360 = 23usec
- Moving from track to track = 900usec
- Skew ~ 40 sectors



Disk Arm Scheduling Algorithms

- Time required to read or write a disk block determined by 3 factors
  - Seek time
  - Rotational delay
  - Actual transfer time

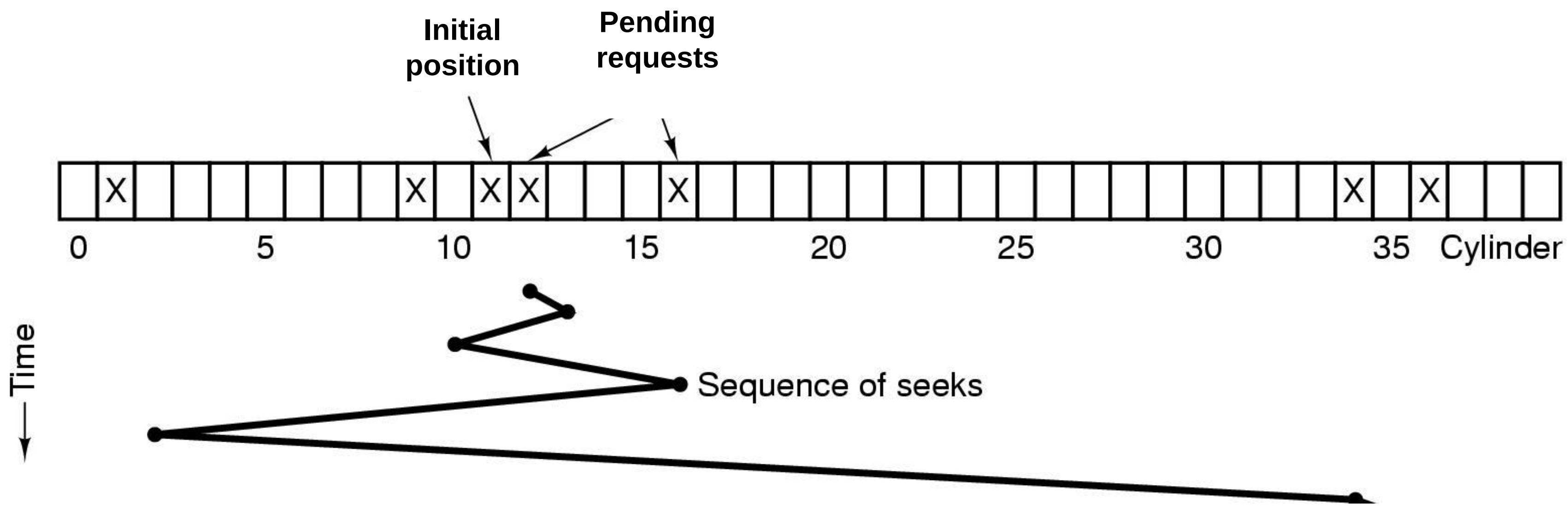
- Seek time is the most relevant and must be minimized

- Possible scheduling algorithms
  - First-Come First-Served: bad
  - Algorithms with request buffering in the driver
    - Shortest Seek First (SSF)
    - Elevator Algorithm

- Note that these algorithms imply that logical/physical geometry match or at least mapping is known

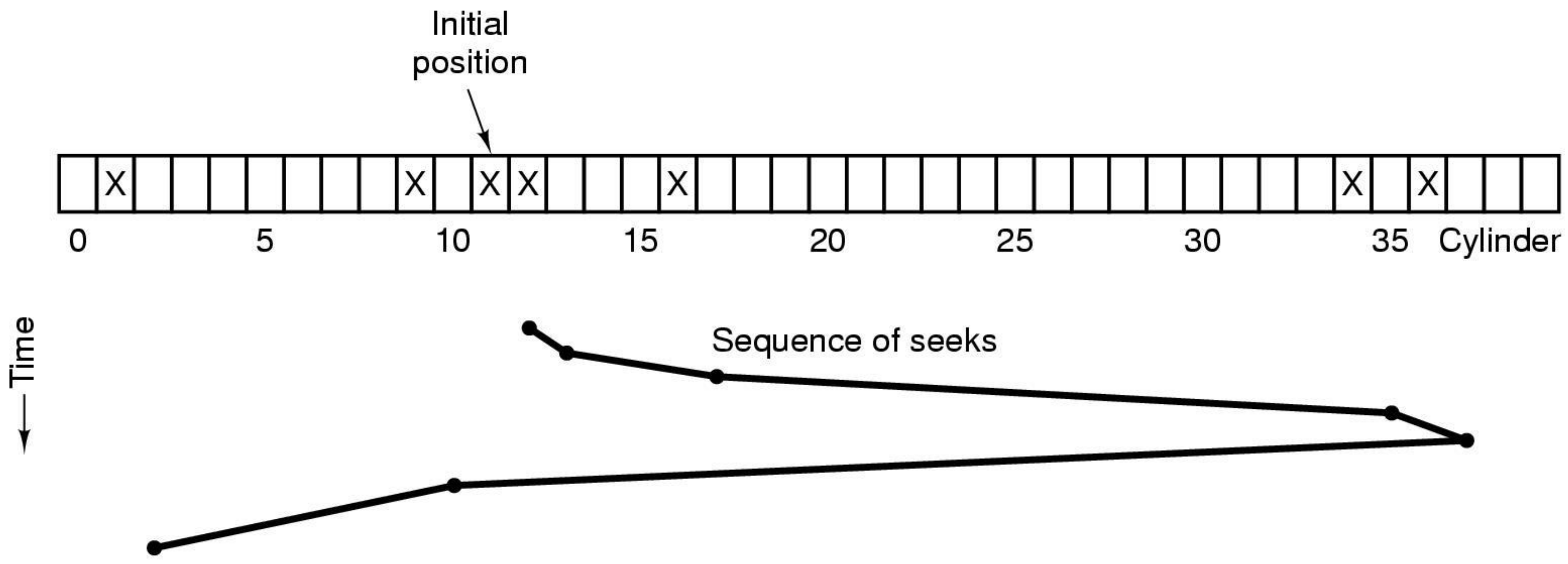
Shortest Seek First Algorithm

- SSF moves the arm towards the closest request
- If request are many the algorithm may be unfair towards request for sectors far from the arm’s current position



Elevator Algorithm

- The arms moves in one direction until there is no request left, then it changes direction



Flash Storage

- SSD, USB thumb drive, …
- Pros:
  - No moving parts
  - Much faster
- Cons:
  - More expensive (per MB)
  - Flash cells can only be written a finite number of times
  - Controller must be more complicated
    - (e.g., wear leveling, bad cell detection, etc.)

Questions?