EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Review

- General Hardware:
  - 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

What we will cover

- General Hardware:
  - 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

I/O software: Device Drivers

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

- User program should have a memory buffer where data is read/written to/from.

I/O Software

System call in user-space

open()/close()/read()/write() for typical operations

ioctl() for device specific operations

Data is copied between user space and kernel space

buffers don't need to be locked in virtual memory

I/O software can operate in several modes

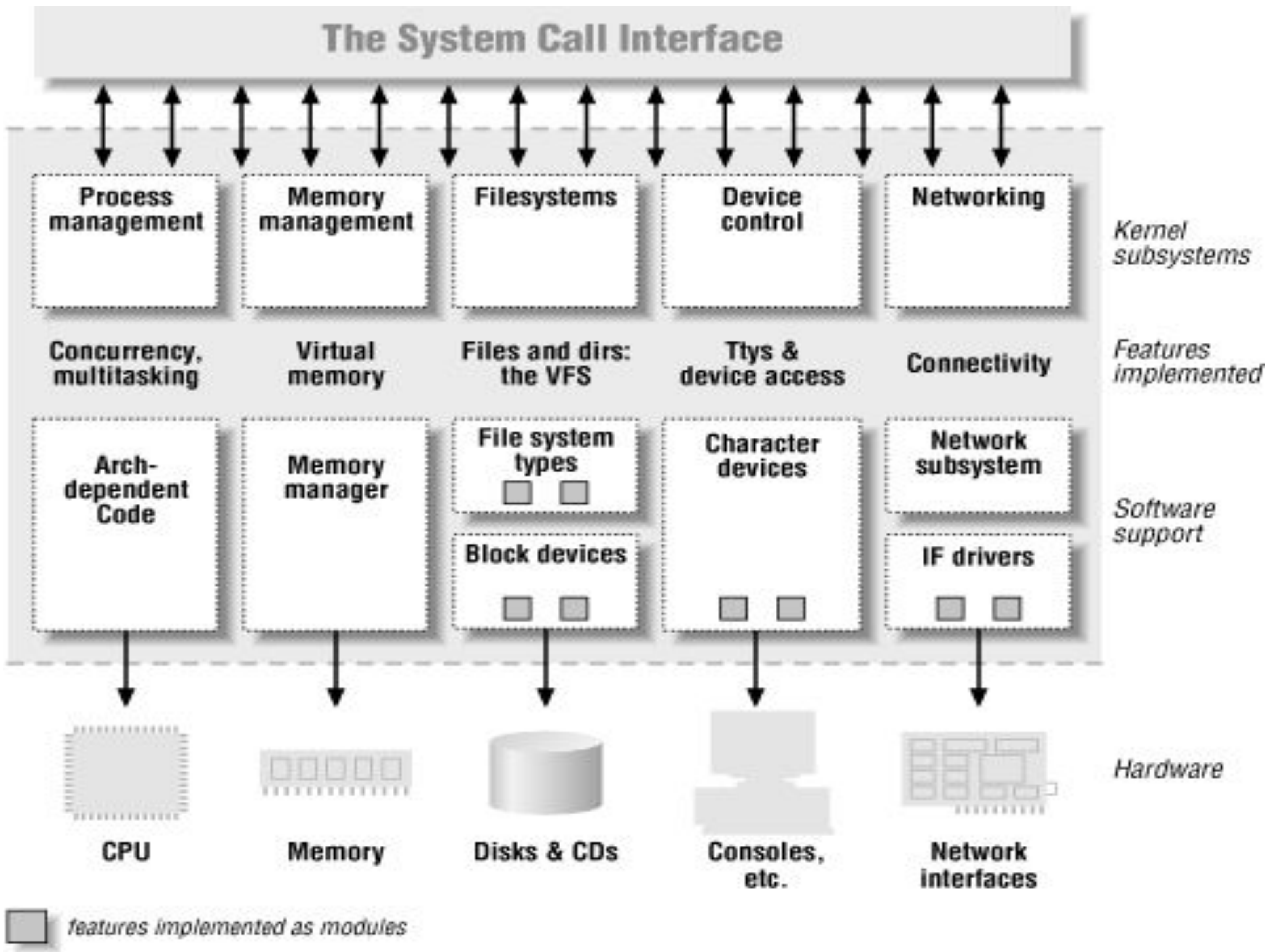
- Programmed I/O
  - Polling/Busy waiting for the device
- Interrupt-Driven I/O
  - Operation is completed by interrupt routine
- DMA-based I/O
  - Set up controller and let it deal with the transfer

Overview Software

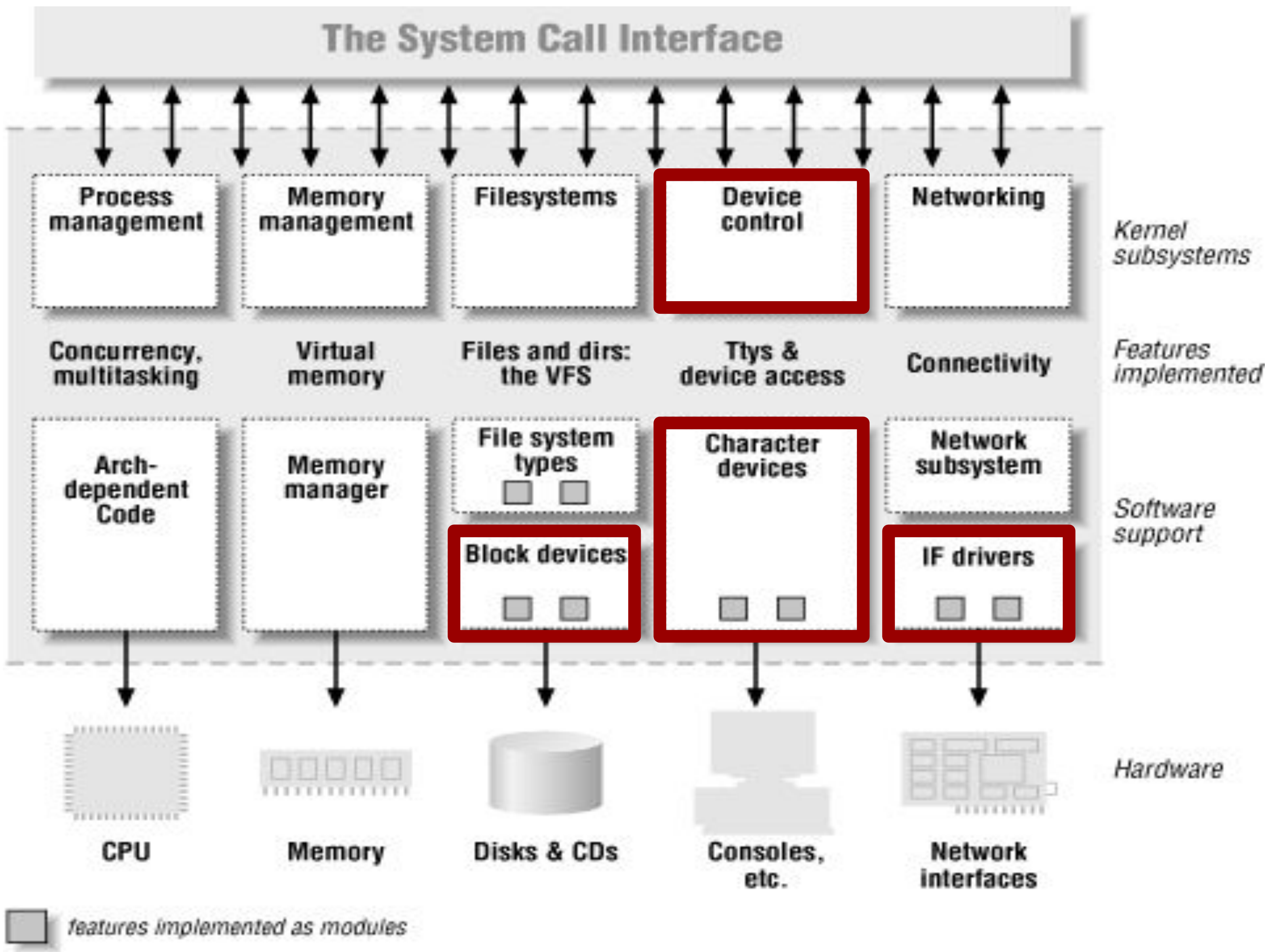
- Where drivers sit in the OS
- Polling/Interrupts/DMA
- Buffering
- Putting it together

Device Drivers

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

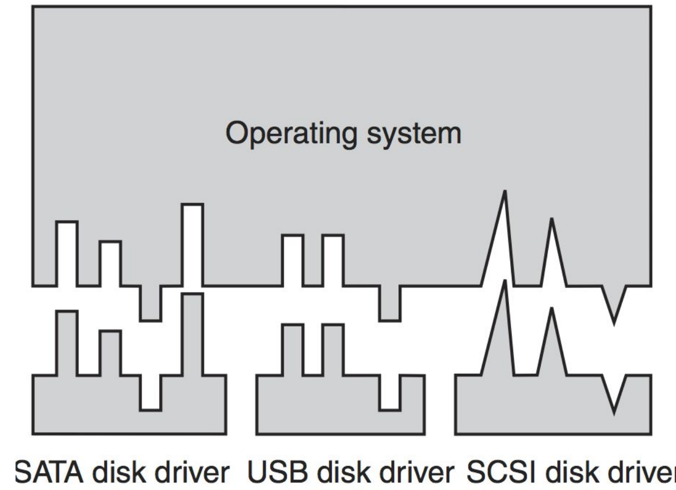


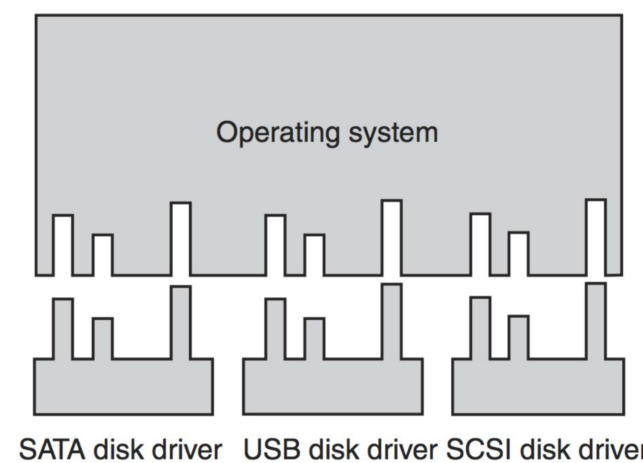
The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent Code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
Hardware
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
features implemented as modules


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent Code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Driver APIs: standard interfaces

- OS creators define a standard driver API
- Specifies:
  - model for OS to interact with the driver
  - what kernel functions are available for driver use
- For popular OS hardware manufacturers will write drivers

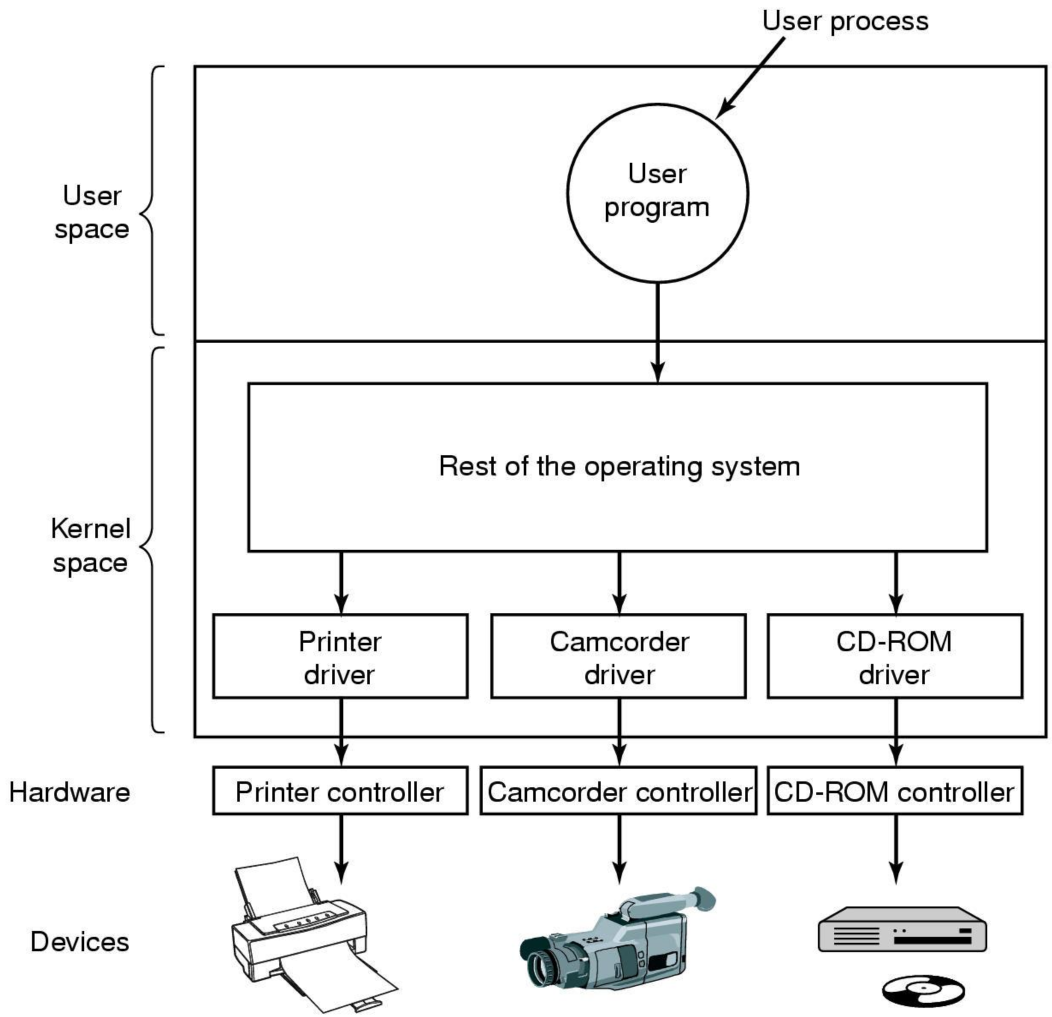




Generic I/O Layer

- 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
- Much like the vfs layer (next lecture);
  - set of functions each device driver must support
  - set of optional functions can support

Device Drivers



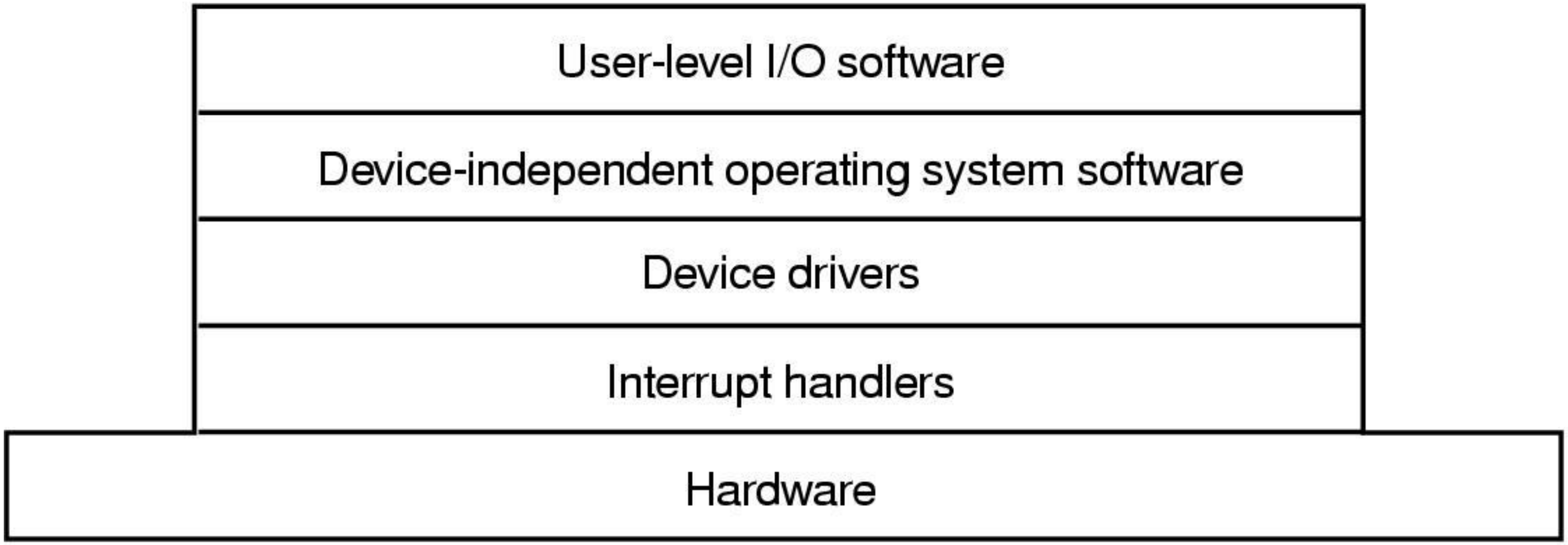
Device Driver's Tasks

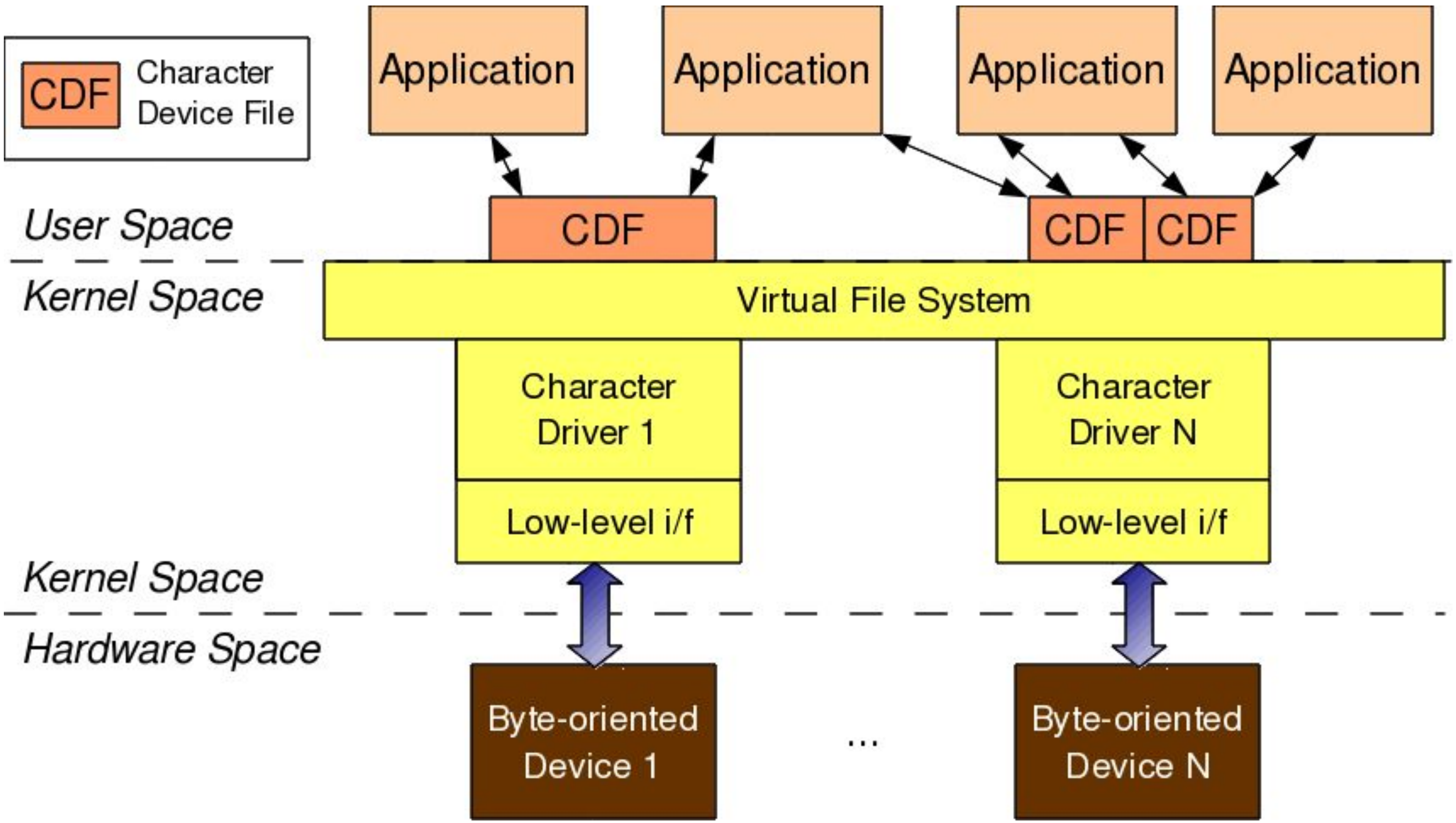
- Device initialization
- Accept read-write request from the OS
  i.e., take commands from higher levels in the OS and translate them into hardware requests
- Start the device if necessary (e.g., start spinning the CD-ROM)
- Check if device is available: if not, wait
- Wait for results
  - Busy wait (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

- Many devices can traverse buffers in memory:
  - e.g., NIC circular buffer read/write buffers
  - SCSI controller list of blocks to read/write
- 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

I/O Handling – Architecture




CDF
Character Device File
Application
Application
Application
Application
User Space
CDF
CDF
CDF
Kernel Space
Virtual File System
Character Driver 1
Character Driver N
Low-level i/f
Low-level i/f
Kernel Space
Hardware Space
Byte-oriented Device 1
...
Byte-oriented Device N

Overview Software

- Where drivers sit in the OS
- Polling/Interrupts/DMA
- Buffering
- Putting it together

Programmed I/O

```
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

- 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

Interrupt-Driven I/O

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

a) Code executed on system call

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

b) Code executed on interrupt service

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 Using DMA

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

a) Code executed on system call

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

b) Code executed on interrupt service

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
- In real OS, don't limit concurrency this way

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; often higher priority thread runnable
- 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*
  - *top half* is the hardware interrupt handler runs with interrupts disabled
  - *bottom half* is a software interrupt handler running with interrupts enabled

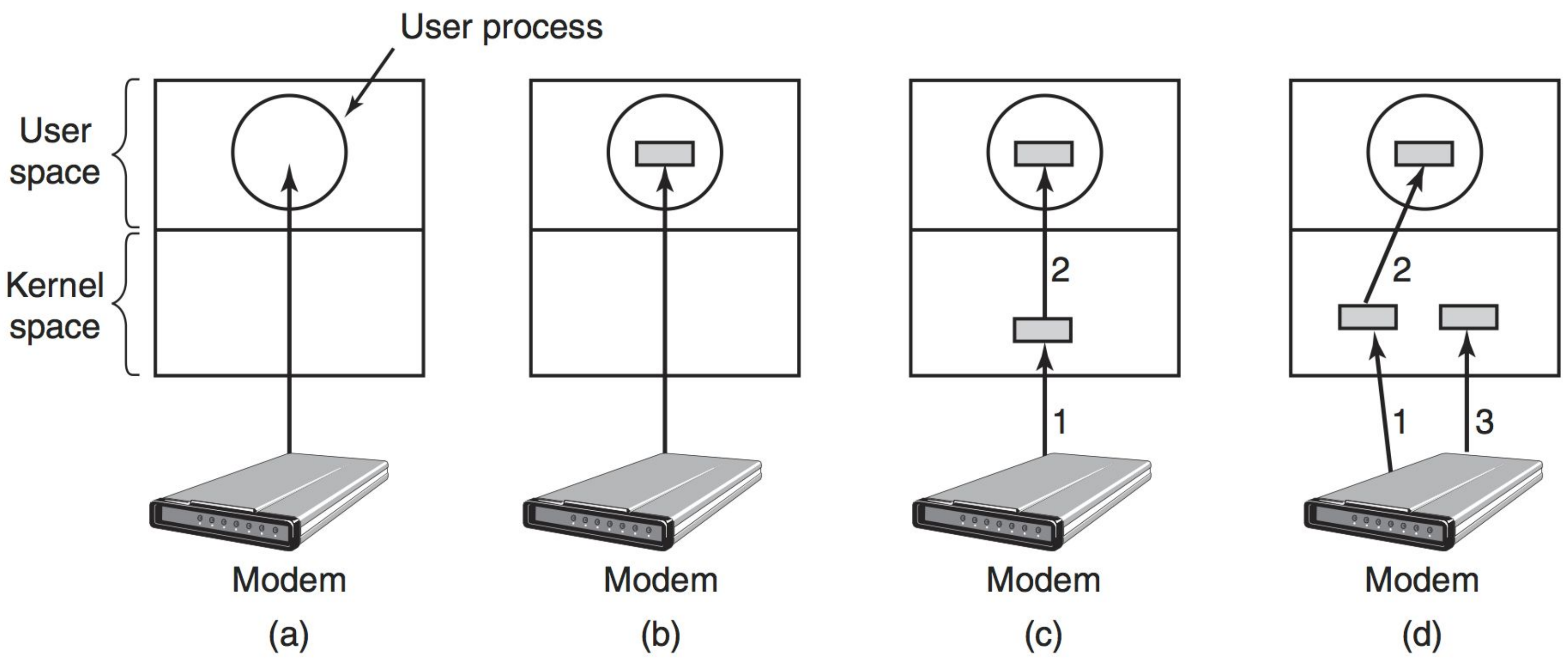
Overview Software

- Where drivers sit in the OS
- Polling/Interrupts/DMA
- Buffering
- Putting it together

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; when one buffer filled, switch to other while copying first

Buffering Example



Buffering Performance

- If there are too many layers of buffering between the hardware and the user program, performance suffers
- Consumes memory and require coping between buffers
- Some operating system/drivers 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

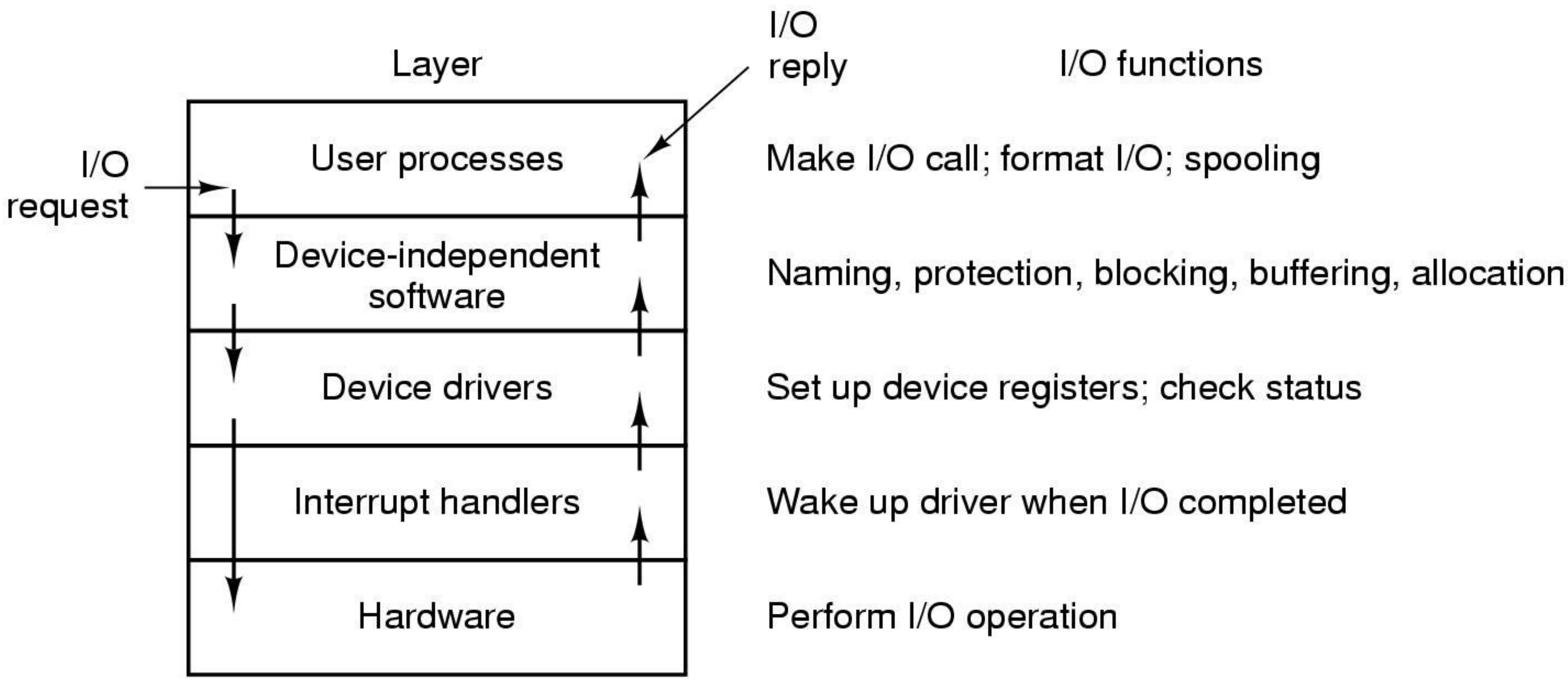
Overview Software

- Where drivers sit in the OS
- Polling/Interrupts/DMA
- Buffering
- Putting it together

User level

- Set of libraries, e.g., printf, scanf process strings and convert to a read/write call
- What about devices that can be accessed by just one user: e.g. printers
- Could let one user access at a time, but what if it fails?
- Typical solution is a user-level spooling daemon; maintains a directory with files to be written...

Putting it all together



What we will cover

- General Hardware:
  - 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

Disk devices

- Usually the 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

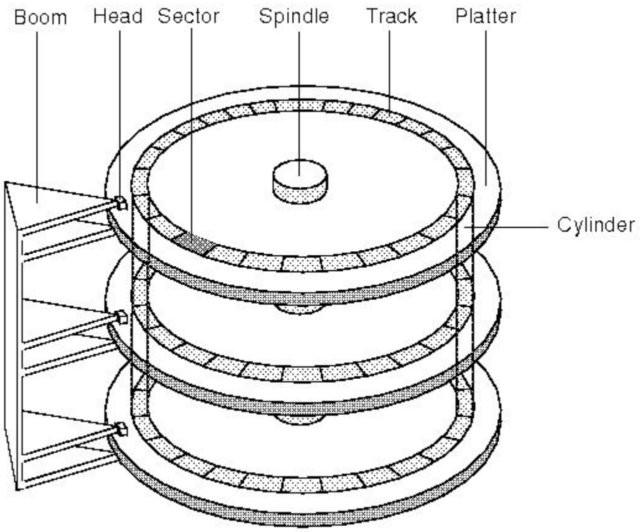
30 Years later improvement

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	800,000x
Seek time (adjacent cylinders)	6 msec	0.8 msec	
Seek time (average case)	77 msec	6.9 msec	9x
Rotation time	200 msec	8.33 msec	24x
Motor stop/start time	250 msec	20 sec	
Time to transfer 1 sector	22 msec	17 μsec	16,000x

Moving parts improve gradually, bit density on recording media increases much faster

Disk Architecture

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



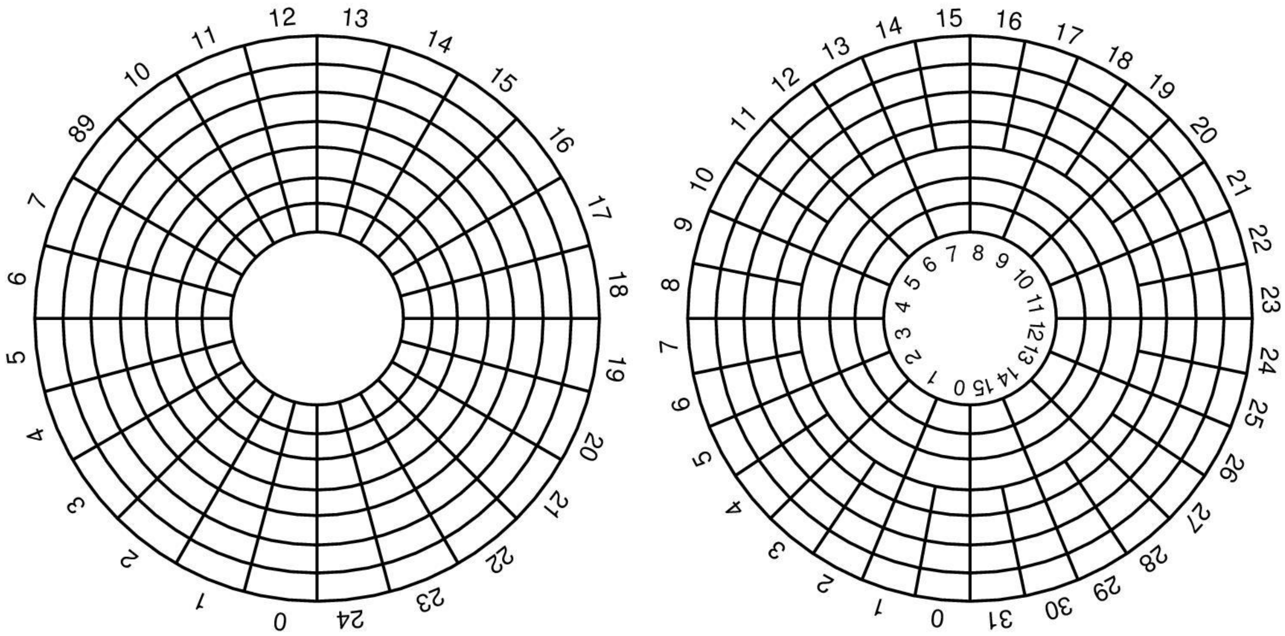
Disk Architecture

Addressing sectors (blocks)

- Logical block address (LBA)
  - decouples logical and physical location
  - specifies 48 bit logical block numbers
  - allows controller to mask corrupt blocks
  - bad block elimination
- Geometry can be complicated, i.e., outside tracks much bigger inner tracks
  - modern disks have 16 or more zones, where each zone differs in tracks
  - virtual geometry used by OS with fixed cylinders/heads/sectors per track; good enough for optimizations described before

Disk Geometry

Simple example with just 2 zones



Disk Architecture

<u>Disk Interfaces</u>

between controller (motherboard) and disk

- ATA (Advanced Technology 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 (Serial ATA)
  - 8 pin cables
  - higher data transfer (differential signaling)

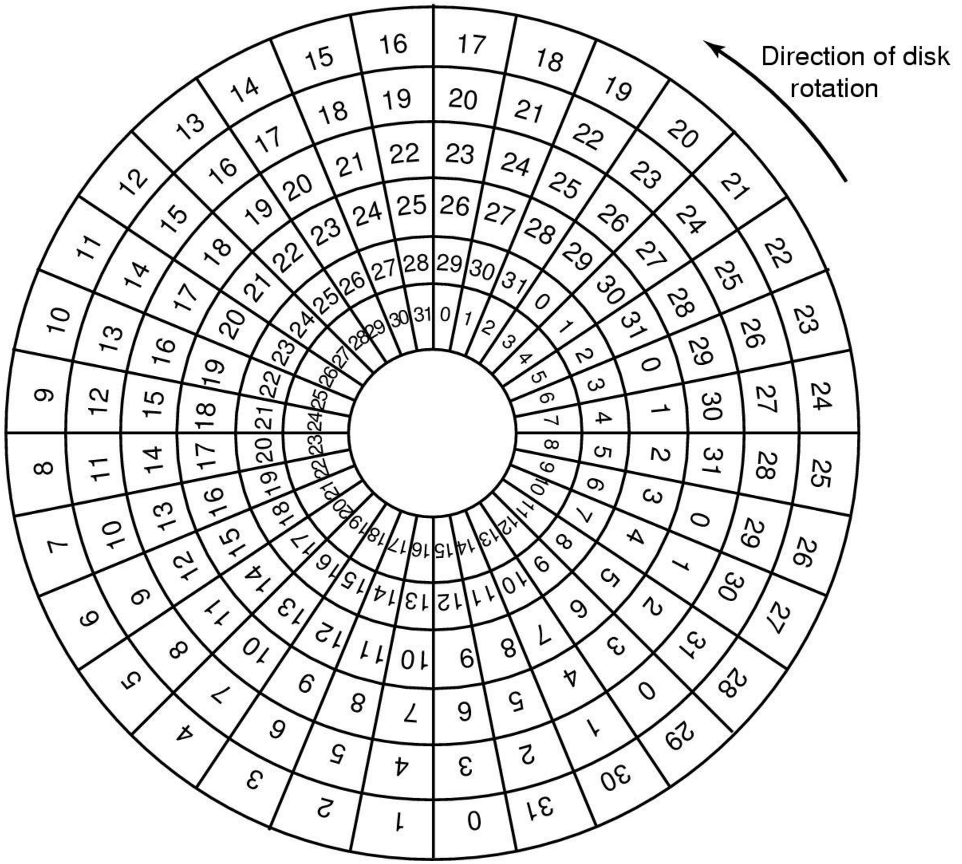
Disk Architecture

- Hidden protected area (HPA)
  - introduced with ATA-4
  - disk can be set to report to OS less blocks than actually available
  - remaining blocks can be used for data that is not formatted utilities and diagnostic tools, but also malicious code or illegal material

- Device configuration overlay (DCO)
  - introduced with ATA-6
  - additional space (blocks) after HPA
  - used by manufacturers to shrink different disks to appear with exactly the same size

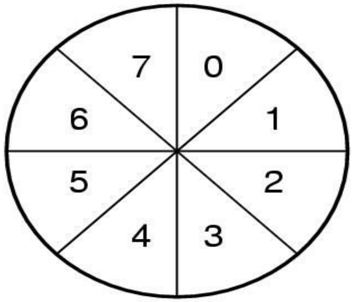
Cylinder Skew

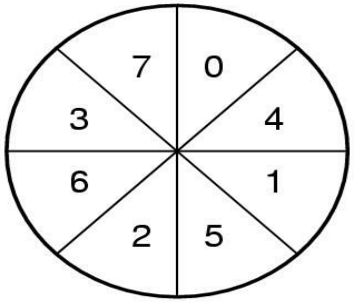
- The initial sector for each track is skewed with respect to the previous one
- This facilitates continuous reads across contiguous tracks by taking into account the rotation of the disk when the arm is moved
- 7,200 rpm with 360 sectors
- Cycle in 60/7,200 = 8,3msec
- Sector rate 8.3msec/360 = 23usec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

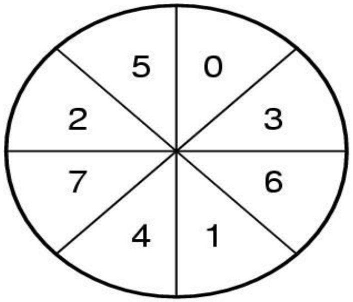


Interleaving

- A disk reads a sector and puts it in the controller's buffer
- While the sector is being transferred to memory the next sector will pass under the disk head
- Solution: Interleaving (single, double, etc)
- Solution: Buffer a whole track at a time







Disk Arm Scheduling Algorithms

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

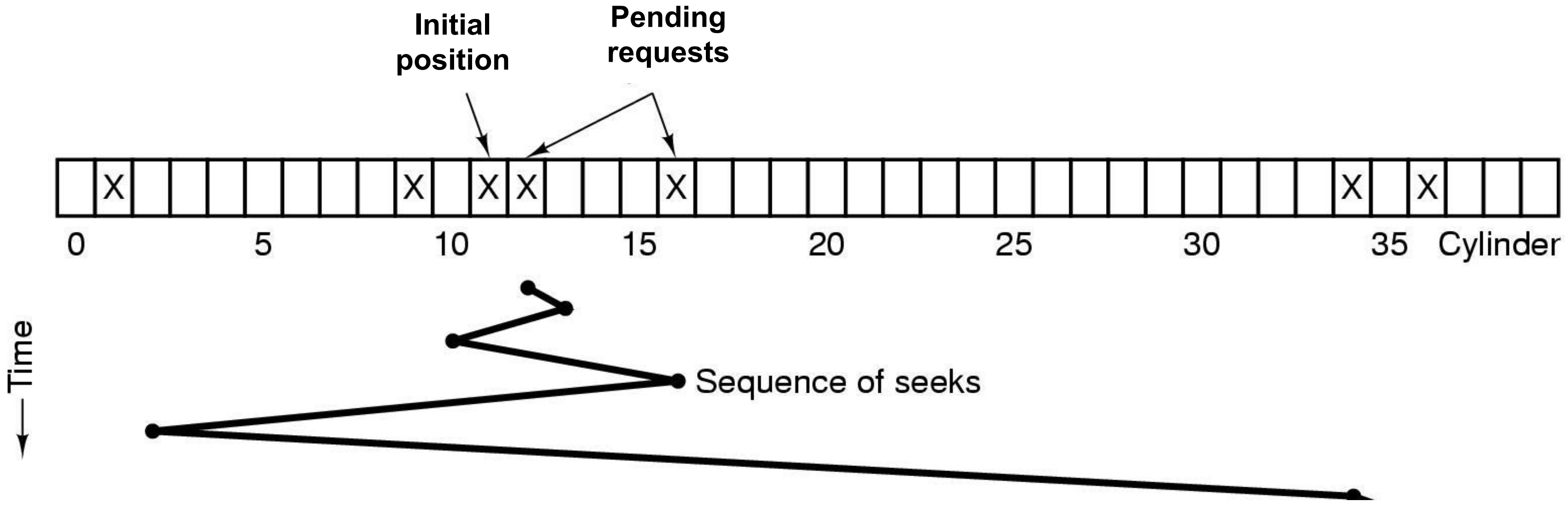
- Seek time is the most relevant and must be minimized

- Possible scheduling algorithms when you have lots of requests:
  - 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

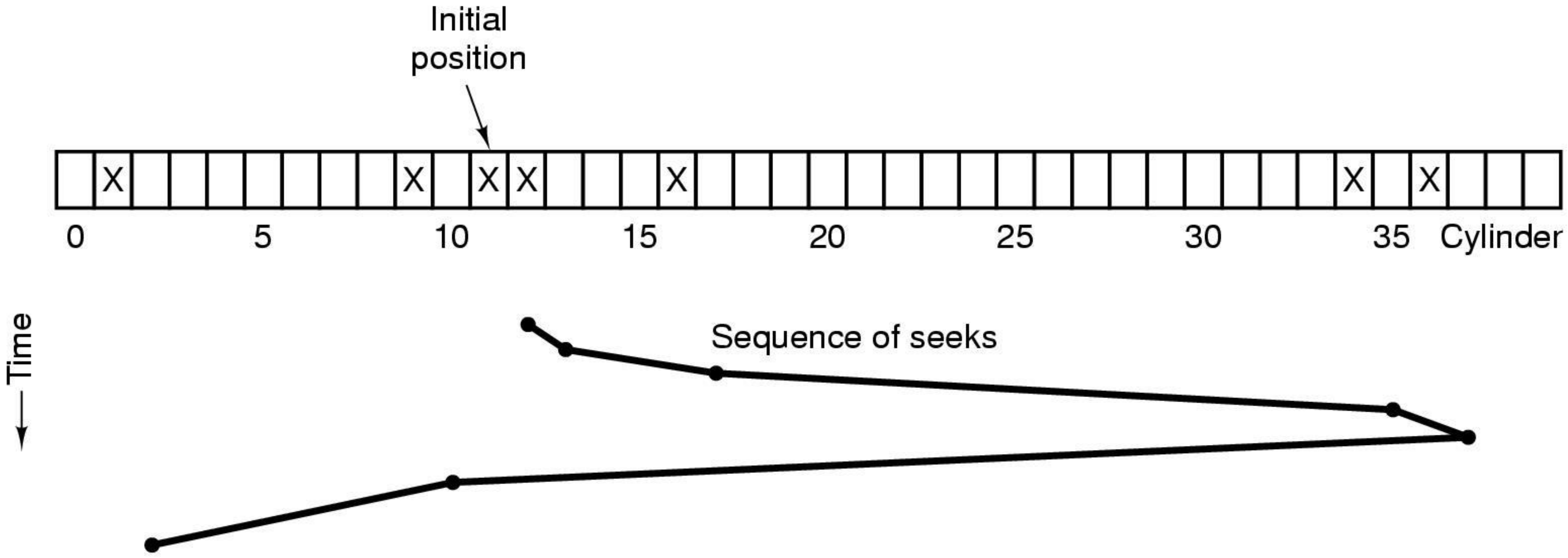
Shortest Seek First Algorithm

- SSF moves the arm towards the closest request
- If request are many the algorithm will keep head near middle, requests at edge/start of disk will starve



Elevator Algorithm

- Same problem occurs elevators in high-rises
- The arms moves in one direction until there is no request left, then it changes direction

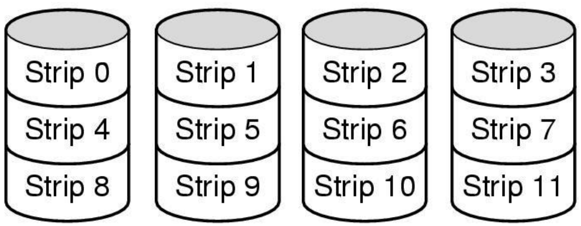


RAID

- Disk performance improved much slower than memory; solution exploit lots of disks
- 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”)
  - distribute data across lots of disks
  - (potentially) maintain redundancy in different ways

RAID

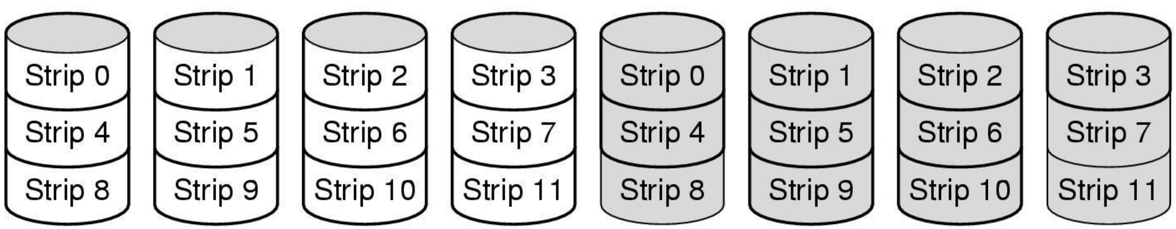
(a)



RAID Level 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
- problem: increases failure rate

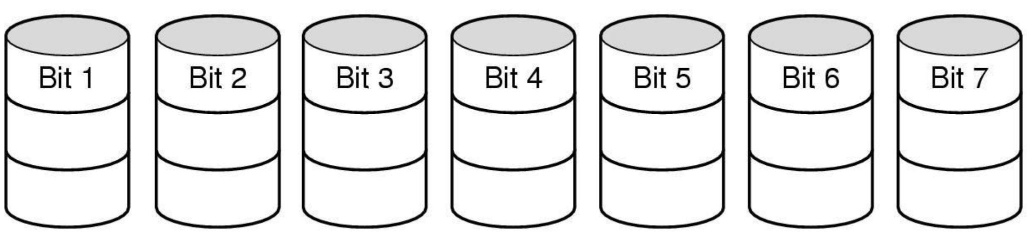
(b)



RAID Level 1

- Striping + Replication
- Doubles amount of storage

(c)

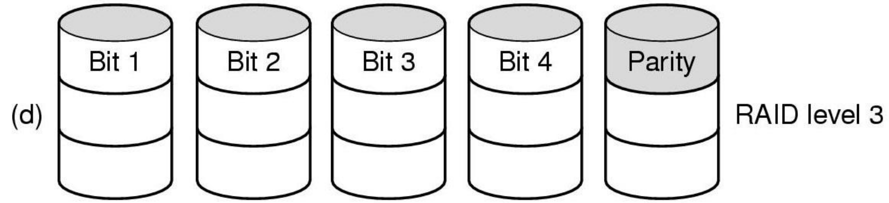


RAID Level 2

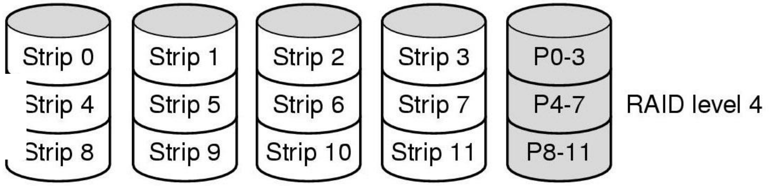
- Striping at word/byte
- ECC to different disks
- disks may be synchronized...

RAID

- RAID 3
  - Parity bit kept on a separate drive
  - In case of disk failure, provides error correcting; note disks have ECC per sector so failure mode is typically disk failes

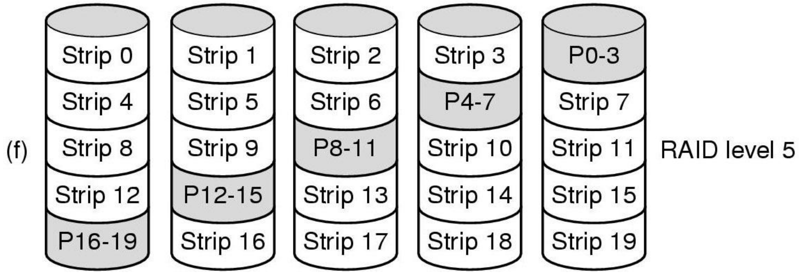


- RAID 4
  - Strip parity on extra drive (XOR of strip contents)
  - Poor performance small writes, need to read rest of stripes
  - Parity disk may become bottleneck



RAID

- RAID level 5
  - Parity stripes distributed over disks to avoid bottleneck



- RAID level 6
  - Multiple parity stripes to handle additional failure

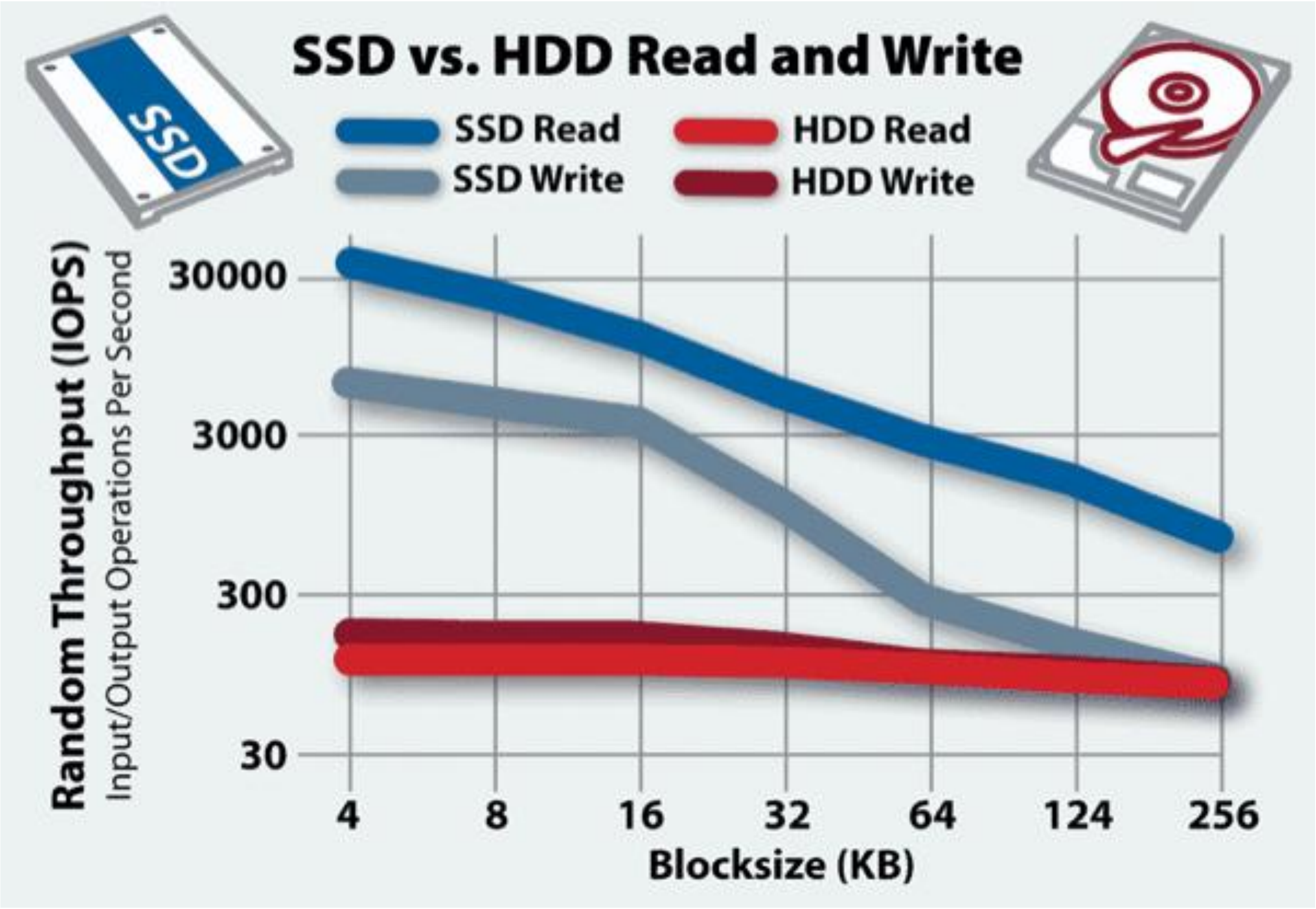
Where things are going...

- Network performance means that disks can be remote
- Increasingly moving to stateless servers, disks accessed over network
- RAID like techniques implemented in software with distributed file systems e.g., Ceph
- SSD increasingly important, and persistent memory coming

SSDs

- SSD - Solid State Disk
  - smaller than HDD
  - faster than HDD
  - more expensive than HDD, limited number of writes
  - shorter lifespan than HDD
  - no seek of rotational components to performance.
  - SSD:
    - reads - 550 megabytes per second (MBps)
    - write - 520 MBps
    - 120 GB to 4 TB capacities
  - HDD:
    - reads and writes at just 125MBps.
    - HDD - 250 GB to 14 TB

SSDs

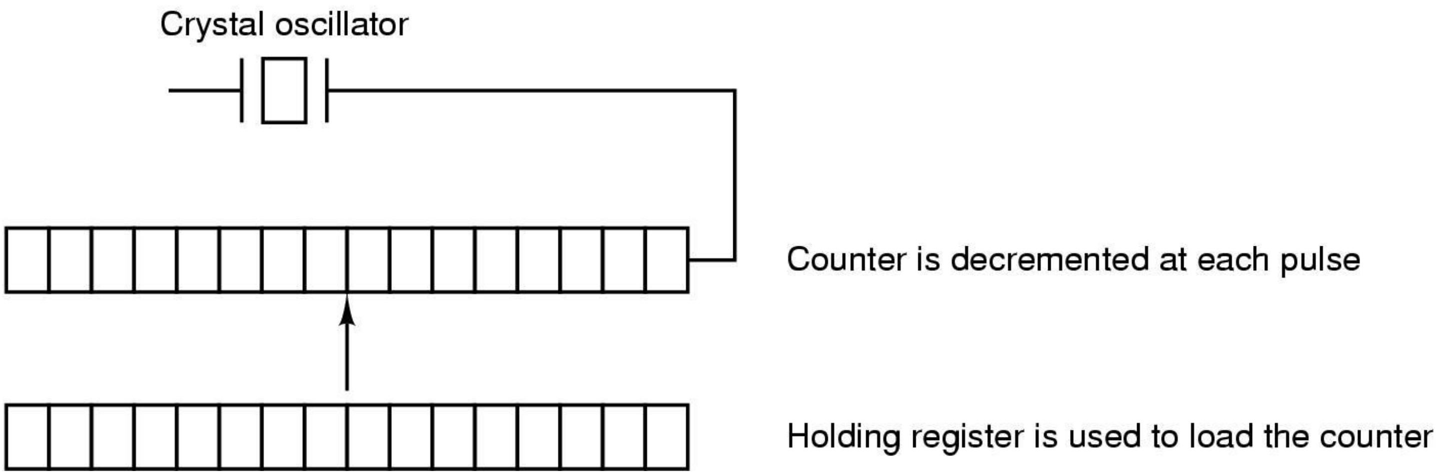

SSD vs. HDD Read and Write
SSD
SSD Read
SSD Write
HDD Read
HDD Write
Random Throughput (IOPS)
Input/Output Operations Per Second
30000
3000
300
30
4
8
16
32
64
124
256
Blocksize (KB)

What we will cover

- General Hardware:
  - 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, Timers & keyboards

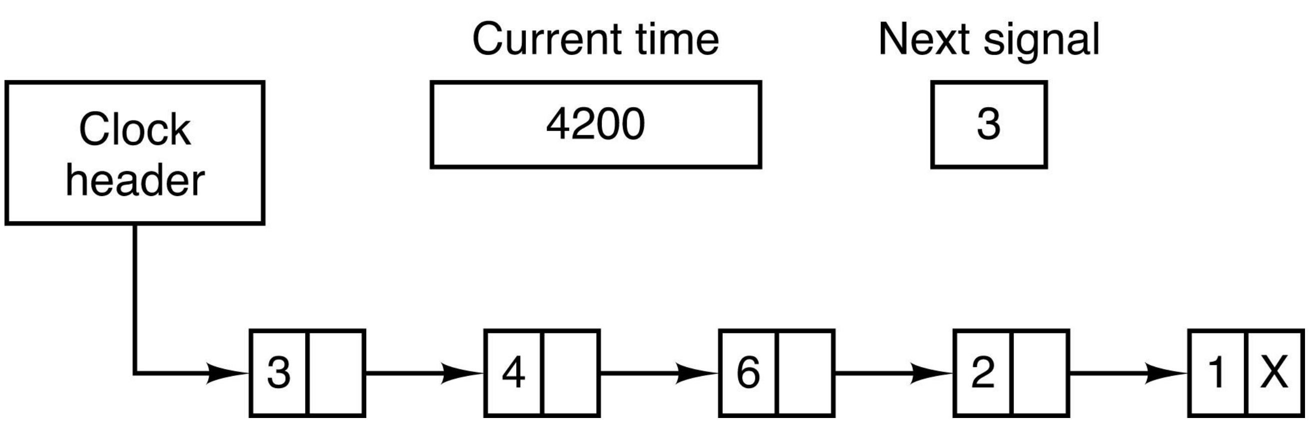
Clock HW

- 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



Clocks/timer

- All the HW does is generate interrupt at know intervals
- SW/kernel:
  - 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/signals/watchdog timers
    - Maintained in a list and fired whenever they expire



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 character

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