Recall: Clock Algorithm (Not Recently Used)

- Which bits of a PTE entry are useful to us?
  - **Use**: Set when page is referenced; cleared by clock algorithm
  - **Modified**: set when page is modified, cleared when page written to disk
  - **Valid**: ok for program to reference this page
  - **Read-only**: ok for program to read page, but not modify
    - For example for catching modifications to code pages!

- **Clock Algorithm**: pages arranged in a ring
  - On page fault:
    - Advance clock hand (not real time)
    - Check use bit: 1 → used recently; clear and leave alone
    - 0 → selected candidate for replacement
  - Crude partitioning of pages into two groups: young and old

Recall: Clock Algorithms Details (continued)

- Do we really need hardware-supported “use” or “dirty” bits?
  - No. Can emulate them in software!
    - Keep software data structure mapping pages → use, dirty, valid, and read-only bits
    - Start by marking all pages invalid (even if in memory)
  - On **read** to invalid page, trap to OS:
    - If page actually in memory, OS sets use bit, and marks page read-only
    - Otherwise handle page fault
  - On **write** to invalid/read-only page, trap to OS:
    - If page actually in memory and supposed to be writable, OS sets use and dirty bits, and marks page read-write
    - Otherwise handle page fault
  - When clock hand advances:
    - Check software use and dirty bits to decide what to do
    - If not reclaiming, mark page invalid and reset software use/dirty bits

- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)

- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list
Second-Chance List Algorithm (continued)

- How many pages for second chance list?
  - If 0 \( \Rightarrow \) FIFO
  - If all \( \Rightarrow \) LRU, but page fault on every page reference
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn’t VAX include “use” bit?
  - Strecker (architect) asked OS people, they said they didn’t need it, so didn’t implement it
  - He later got blamed, but VAX did OK anyway

Free List

- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique (“Pageout demon”)
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
  - Advantage: faster for page fault
  - Can always use page (or pages) immediately on fault

Reverse Page Mapping
(Sometimes called “Coremap”)

- Physical page frames often shared by many different address spaces/page tables
  - All children forked from given process
  - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages “active”
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    - Management nightmare – expensive
  - Linux: Object-based reverse mapping
    - Link together memory region descriptors instead (much coarser granularity)

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?*
  - Should we completely swap some processes out of memory?
- Each process needs minimum number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle from
    - 2 pages to handle to
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames
Fixed/Priority Allocation

• Equal allocation (Fixed Scheme):
  – Every process gets same amount of memory
  – Example: 100 frames, 5 processes → process gets 20 frames

• Proportional allocation (Fixed Scheme)
  – Allocate according to the size of process
  – Computation proceeds as follows:
    \[ s_i = \text{size of process } p_i \text{ and } S = \sum s_i \]
    \[ m = \text{total number of physical frames in the system} \]
    \[ a_i = \text{(allocation for } p_i) = \frac{s_i}{S} \times m \]

• Priority Allocation:
  – Proportional scheme using priorities rather than size
  – If process \( p_i \) generates a page fault, select for replacement a frame from a process with lower priority number
  – Perhaps we should use an adaptive scheme instead???
  – What if some application just needs more memory?

Page-Fault Frequency Allocation

• Can we reduce Capacity misses by dynamically changing the number of pages/application?

Thrashing

• If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  – low CPU utilization
  – operating system spends most of its time swapping to disk
• Thrashing = a process is busy swapping pages in and out with little or no actual progress
• Questions:
  – How do we detect Thrashing?
  – What is best response to Thrashing?

Locality In A Memory-Reference Pattern

• Program Memory Access Patterns have temporal and spatial locality
  – Group of Pages accessed along a given time slice called the “Working Set”
  – Working Set defines minimum number of pages needed for process to behave well
• Not enough memory for Working Set ⇒ Thrashing
  – Better to swap out process?
Working-Set Model

- $\Delta$ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- $W_{Si}$ (working set of Process $P_i$) = total set of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty$ ⇒ will encompass entire program
- $D = \Sigma |W_{Si}|$ ≡ total demand frames
- if $D > m$ ⇒ Thrashing
  - Policy: if $D > m$, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Linux Memory Details?

- Memory management in Linux considerably more complex than the examples we have been discussing
- Memory Zones: physical memory categories
  - ZONE_DMA: < 16MB memory, DMAable on ISA bus
  - ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
  - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
  - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
  - Anonymous memory (not backed by a file, heap/stack)
  - Mapped memory (backed by a file)
- Allocation priorities
  - Is blocking allowed/etc

Linux Virtual memory map (Pre-Meltdown)

- 32GB Total
  - 128TiB Physical
  - 896MB Physical
  - 0x0000000000000000
  - 0xFFFFFFFFFFFFFFFF
  - 0x00007FFFFFFF
  - 0xFFFF800000000000
  - 0xFFFFFFFFFFFFFFFF
- 36GB Total
  - 64 TiB Physical
  - 0xFFFF800000000000
  - 0x0000000000000000
  - “Canonical Hole”
- 32-Bit Virtual Address Space
  - User Addresses
  - Kernel Addresses
- 64-Bit Virtual Address Space
  - User Addresses
  - Kernel Addresses
  - Empty Space
Pre-Meltdown Virtual Map (Details)

• Kernel memory not generally visible to user
  – Exception: special VDSO (virtual dynamically linked shared objects) facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as gettimeofday())

• Every physical page described by a “page” structure
  – Collected together in lower physical memory
  – Can be accessed in kernel virtual space
  – Linked together in various “LRU” lists

• For 32-bit virtual memory architectures:
  – When physical memory < 896MB
    » All physical memory mapped at 0xC0000000
  – When physical memory >= 896MB
    » Not all physical memory mapped in kernel space all the time
    » Can be temporarily mapped with addresses > 0xCC000000

• For 64-bit virtual memory architectures:
  – All physical memory mapped above 0xFFFF800000000000

Post Meltdown Memory Map

• Meltdown flaw (2018, Intel x86, IBM Power, ARM)
  – Exploit speculative execution to observe contents of kernel memory
  1: // Set up side channel (array flushed from cache)
  2: uchar array[256 * 4096];
  3: flush(array); // Make sure array out of cache
  4: try {
  5:   uchar result = *(uchar *)kernel_address; // Try access!
  6:   uchar dummy = array[result * 4096]; // leak info!
  7: } catch(){;} // Could use signal() and setjmp/longjmp….
  8: // scan through 256 array slots to determine which loaded

  – Some details:
    » Reason we skip 4096 for each value: avoid hardware cache prefetch
    » Note that value detected by fact that one cache line is loaded
    » Catch and ignore page fault: set signal handler for SIGSEGV, can use setjump/longjmp….

• Patch: Need different page tables for user and kernel
  – Without PCID tag in TLB, flush TLB twice on syscall (800% overhead!)
  – Need at least Linux v 4.14 which utilizes PCID tag in new hardware to avoid flushing when change address space

• Fix: better hardware without timing side-channels
  – Will be coming, but still in works

Administrivia (1/2)

• Supplemental Project 1 Peer evaluations will be out soon (Forgot to include the collaboration question):
  – It is very important that you fill this out as well.
  – You get 20 points for each partner that you distribute to partners (with 3 other partners, get 60 total points):
    » For instance: happy with all partners, give them each 20 points
    » Less happy with one of them, give them 18, 21 to other two, etc
    » Everything validated by TA, in the end, of course
  – The project grades are a zero-sum game; if you do not contribute to the project, your points might be distributed to those who do

• This is not about giving yourself more points by giving partners less….
  – It is about you evaluating your partners (and they evaluate you!)

Administrivia (2/2)

• Midterm Survey is up
  – Please fill this out so that we can know how we are doing

• Midterm 2: Thursday 4/4
  – Will definitely include Scheduling material (lecture 10)
  – Will include I/O material from today’s lecture (lecture 16)
  – Will have a Midterm review in early part of that week…. Stay tuned.
The Requirements of I/O

• So far in this course:
  – We have learned how to manage CPU and memory

• What about I/O?
  – Without I/O, computers are useless (disembodied brains?)
  – But… thousands of devices, each slightly different
  » How can we standardize the interfaces to these devices?
  – Devices unreliable: media failures and transmission errors
  » How can we make them reliable???
  – Devices unpredictable and/or slow
  » How can we manage them if we don’t know what they will do or how they will perform?

OS Basics: I/O

In a Picture

• I/O devices you recognize are supported by I/O Controllers
• Processors accesses them by reading and writing IO registers as if they were memory
  – Write commands and arguments, read status and results

Operational Parameters for I/O

• Data granularity: Byte vs. Block
  – Some devices provide single byte at a time (e.g., keyboard)
  – Others provide whole blocks (e.g., disks, networks, etc.)

• Access pattern: Sequential vs. Random
  – Some devices must be accessed sequentially (e.g., tape)
  – Others can be accessed “randomly” (e.g., disk, cd, etc.)
  » Fixed overhead to start transfers
  – Some devices require continual monitoring
  – Others generate interrupts when they need service

• Transfer Mechanism: Programmed IO and DMA
The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices
  - This code works on many different devices:
    ```c
    FILE fd = fopen("/dev/something", "rw");
    for (int i = 0; i < 10; i++) {
        fprintf(fd, "Count %d\n", i);
    }
    close(fd);
    ```
- Why? Because code that controls devices ("device driver") implements standard interface
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

Want Standard Interfaces to Devices

- **Block Devices:** e.g. disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices:** e.g. keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices:** e.g. Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include `socket` interface
    » Separates network protocol from network operation
    » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

- **Blocking Interface:** "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** "Don’t Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** "Tell Me Later"
  - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Chip-scale Features of 2015 x86 (Sky Lake)

- Significant pieces:
  - Four OOO cores with deeper buffers
    » New Intel MPX (Memory Protection Extensions)
    » New Intel SGX (Software Guard Extensions)
    » Issue up to 6 μ-ops/cycle
  - Integrated GPU, System Agent (Mem, Fast I/O)
  - Large shared L3 cache with on-chip ring bus
    » 2 MB/core instead of 1.5 MB/core
    » High-BW access to L3 Cache
- Integrated I/O
  - Integrated memory controller (IMC)
    » Two independent channels of DRAM
  - High-speed PCI-Express (for Graphics cards)
  - Direct Media Interface (DMI) Connection to PCH (Platform Control Hub)

Sky Lake I/O: PCH

- Platform Controller Hub
  - Connected to processor with proprietary bus
    » Direct Media Interface
- Types of I/O on PCH:
  - USB, Ethernet
  - Thunderbolt 3
  - Audio, BIOS support
  - More PCI Express (lower speed than on Processor)
  - SATA (for Disks)

Modern I/O Systems

Example: PCI Architecture
Example Device-Transfer Rates in Mb/s (Sun Enterprise 6000)

- Device Rates vary over 12 orders of magnitude!!!
  - System better able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

How does the processor talk to the device?

- CPU interacts with a Controller
  - Contains a set of registers that can be read and written
  - May contain memory for request queues or bit-mapped images

Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
- I/O instructions: in/out instructions
  - Example from the Intel architecture: out 0x21, AL
- Memory mapped I/O: load/store instructions
  - Registers/memory appear in physical address space
  - I/O accomplished with load and store instructions

Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by HW jumpers or at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - Addr: 0x8000F000 — 0x8000FFFF
  - Writing graphics description to cmd queue
    - Say enter a set of triangles describing some scene
    - Addr: 0x80010000 — 0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
    - Addr: 0x0007F004

- Can protect with address translation

Transferring Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size

- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data blocks to/from memory directly

- Sample interaction with DMA controller (from OSC book):
Programmed I/O:
- Each byte transferred via processor in/out or load/store
- Pro: Simple hardware, easy to program
- Con: Consumes processor cycles proportional to data size

Direct Memory Access:
- Give controller access to memory bus
- Ask it to transfer data blocks to/from memory directly

Sample interaction with DMA controller (from OSC book):

I/O Device Notifying the OS
- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - Interrupt for first incoming packet
    - Poll for following packets until hardware queues are empty

Device Drivers
- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the ioctl() system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - implements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    - This is the kernel’s interface to the device driver
    - Top half will start I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request
Basic Performance Concepts

- **Response Time or Latency**: Time to perform an operation(s)

- **Bandwidth or Throughput**: Rate at which operations are performed (op/s)
  - Files: MB/s, Networks: Mb/s, Arithmetic: GFLOP/s

- **Start up or “Overhead”**: time to initiate an operation

- Most I/O operations are roughly linear in \( b \) bytes
  - Latency(\( b \)) = Overhead + \( b/\text{TransferCapacity} \)

---

Example (Fast Network)

- Consider a 1 Gb/s link (B = 125 MB/s)
  - With a startup cost \( S = 1 \) ms

\[
\text{Latency}(b) = S + \frac{b}{B} \\
\text{Bandwidth} = \frac{b}{(S + \frac{b}{B})} = \frac{B*b}{(B*S + b)} = \frac{B}{(B*S/b + 1)}
\]

---

Example: at 10 ms startup (like Disk)

- **Half-power Bandwidth** \( \Rightarrow \frac{B}{(B*S/b + 1)} = B/2 \)

- Half-power point occurs at \( b = S*B = 125,000 \) bytes
What Determines Peak BW for I/O?

- **Bus Speed**
  - PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
  - ULTRA WIDE SCSI: 40 MB/s
  - Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire): 1.6 Gb/s full duplex (200 MB/s)
  - USB 3.0 – 5 Gb/s
  - Thunderbolt 3 – 40 Gb/s

- **Device Transfer Bandwidth**
  - Rotational speed of disk
  - Write / Read rate of NAND flash
  - Signaling rate of network link

- Whatever is the bottleneck in the path…

---

Storage Devices

- **Magnetic disks**
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access (except for SMR – later!)
  - Slow performance for random access
  - Better performance for sequential access

- **Flash memory**
  - Storage that rarely becomes corrupted
  - Capacity at intermediate cost (5-20x disk)
  - Block level random access
  - Good performance for reads; worse for random writes
  - Erasure requirement in large blocks
  - Wear patterns issue

---

Hard Disk Drives (HDDs)

- **IBM/Hitachi Microdrive**
  - Western Digital Drive
  - 30 MB hard disk - $500
  - 30-40ms seek time
  - 0.7-1 MB/s (est.)

---

The Amazing Magnetic Disk

- **Unit of Transfer: Sector**
  - Ring of sectors form a track
  - Stack of tracks form a cylinder
  - Heads position on cylinders

- **Disk Tracks ~ 1µm (micron) wide**
  - Wavelength of light is ~ 0.5µm
  - Resolution of human eye: 50µm
  - 100K tracks on a typical 2.5” disk

- **Separated by unused guard regions**
  - Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)
The Amazing Magnetic Disk

- Track length varies across disk
  - Outside: More sectors per track, higher bandwidth
  - Disk is organized into regions of tracks with same # of sectors/track
  - Only outer half of radius is used
    » Most of the disk area in the outer regions of the disk
- Disks so big that some companies (like Google) reportedly only use part of disk for active data
  - Rest is archival data

Shingled Magnetic Recording (SMR)

- Overlapping tracks yields greater density, capacity
- Restrictions on writing, complex DSP for reading
- Examples: Seagate (8TB), Hitachi (10TB)

Review: Magnetic Disks

- Cylinders: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track
  - Rotational latency: wait for desired sector to rotate under r/w head
  - Transfer time: transfer a block of bits (sector) under r/w head

Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time
Typical Numbers for Magnetic Disk

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Info / Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space/Density</td>
<td>Space: 14TB (Seagate), 8 platters, in 3½ inch form factor! Areal Density: ≥ 1 Terabit/square inch! (PMR, Helium, …)</td>
</tr>
<tr>
<td>Average seek time</td>
<td>Typically 4-6 milliseconds. Depending on reference locality, actual cost may be 25-33% of this number.</td>
</tr>
<tr>
<td>Average rotational latency</td>
<td>Most laptop/desktop disks rotate at 3600-7200 RPM (16-8 ms/rotation). Server disks up to 15,000 RPM. Average latency is halfway around disk so 8-4 milliseconds</td>
</tr>
<tr>
<td>Controller time</td>
<td>Depends on controller hardware</td>
</tr>
</tbody>
</table>
| Transfer time                 | Typically 50 to 250 MB/s. Depends on:  
• Transfer size (usually a sector): 512B – 1KB per sector  
• Rotation speed: 3600 RPM to 15000 RPM  
• Recording density: bits per inch on a track  
• Diameter: ranges from 1 in to 5.25 in |
| Cost                          | Used to drop by a factor of two every 1.5 years (or even faster); now slowing down |

Disk Performance Example

• Assumptions:  
  – Ignoring queuing and controller times for now  
  – Avg seek time of 5ms,  
  – 7200RPM ≡ Time for rotation: 60000 (ms/min) / 7200(rev/min) ≈ 8ms  
  – Transfer rate of 50MByte/s, block size of 4Kbyte ⇒ 4096 bytes/50×10^6 (bytes/s) = 8.12 × 10^-4 sec ≈ 0.082 ms for 1 sector

• Read block from random place on disk:  
  – Seek (5ms) + Rot. Delay (4ms) + Transfer (0.082ms) = 9.082ms  
  – Approx 9ms to fetch/put data: 4096 bytes/9.082×10^-3 s = 451KB/s

• Read block from random place in same cylinder:  
  – Rot. Delay (4ms) + Transfer (0.082ms) = 4.082ms  
  – Approx 4ms to fetch/put data: 4096 bytes/4.082×10^-3 s = 1.03MB/s

• Read next block on same track:  
  – Transfer (0.082ms): 4096 bytes/0.082×10^-3 s = 50MB/sec

• Key to using disk effectively (especially for file systems) is to minimize seek and rotational delays

(Lots of) Intelligence in the Controller

• Sectors contain sophisticated error correcting codes  
  – Disk head magnet has a field wider than track  
  – Hide corruptions due to neighboring track writes

• Sector sparing  
  – Remap bad sectors transparently to spare sectors on the same surface

• Slip sparing  
  – Remap all sectors (when there is a bad sector) to preserve sequential behavior

• Track skewing  
  – Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops

…

Hard Drive Prices over Time

![Hard Drive Prices Chart](chart.png)
Example of Current HDDs

- Seagate Exos X14 (2018)
  - 14 TB hard disk
  - 8 platters, 16 heads
  - Helium filled: reduce friction and power
  - 4.16ms average seek time
  - 4096 byte physical sectors
  - 7200 RPMs
  - 6 Gbps SATA /12Gbps SAS interface
    » 261MB/s MAX transfer rate
    » Cache size: 256MB
  - Price: $615 (< $0.05/GB)

- IBM Personal Computer/AT (1986)
  - 30 MB hard disk
  - 30-40ms seek time
  - 0.7-1 MB/s (est.)
  - Price: $500 ($17K/GB, 340,000x more expensive !!)

Some “Current” 3.5in SSDs

- Seagate Nytro SSD: 15TB (2017)
  - Dual 12Gb/s interface
  - Seq reads 860MB/s
  - Seq writes 920MB/s
  - Random Reads (IOPS): 102K
  - Random Writes (IOPS): 15K
  - Price (Amazon): $6325 ($0.41/GB)

- Nimbus SSD: 100TB (2019)
  - Dual port: 12Gb/s interface
  - Seq reads/writes: 500MB/s
  - Random Read Ops (IOPS): 100K
  - Unlimited writes for 5 years!
  - Price: ~ $50K? ($0.50/GB)

HDD vs SSD Comparison

- SSD prices drop much faster than HDD

Summary

- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    » Blocking, Non-blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
- Notification mechanisms
  - Interrupts
  - Polling: Report results through status register that processor looks at periodically
- Device drivers interface to I/O devices
  - Provide clean Read/Write interface to OS above
  - Manipulate devices through PIO, DMA & interrupt handling
  - Three types: block, character, and network