Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall 61C: Average Memory Access Time

- Used to compute access time probabilistically:
  \[ AMAT = \text{Hit Rate}_{L1} \times \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Time}_{L1} \]
  
  \[
  \text{Hit Rate}_{L1} + \text{Miss Rate}_{L1} = 1 \\
  \text{Hit Time}_{L1} = \text{Time to get value from L1 cache.} \\
  \text{Miss Time}_{L1} = \text{Hit Time}_{L1} + \text{Miss Penalty}_{L1} \\
  \text{Miss Penalty}_{L1} = \text{AVG Time to get value from lower level (DRAM)}
  
  \text{So, AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}
  
- What about more levels of hierarchy?
  \[ AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1} \]
  
  \[
  \text{Miss Penalty}_{L1} = \text{AVG time to get value from lower level (L2)} \\
  = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2} \\
  \text{Miss Penalty}_{L2} = \text{Average Time to fetch from below L2 (DRAM)}
  
  \[ AMAT = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}) \]
  
  • And so on … (can do this recursively for more levels!)
Recall: Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - \( \text{EAT} = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time} \)
  - \( \text{EAT} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty} \)

- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose \( p = \text{Probability of miss} \), \( 1-p = \text{Probability of hit} \)
  - Then, we can compute EAT as follows:
    \[
    \text{EAT} = 200\text{ns} + p \times 8\text{ ms} \\
    = 200\text{ns} + p \times 8,000,000\text{ns}
    \]

- If one access out of 1,000 causes a page fault, then \( \text{EAT} = 8.2\ \mu\text{s} \):
  - This is a slowdown by a factor of 40!

- What if want slowdown by less than 10%?
  - \( \text{EAT} < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6} \)
  - This is about 1 page fault in 400,000!
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 structure from pages ⇒ use, dirty, writable, present 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?
  - 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 ⇒ FIFO
  – If all ⇒ 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)

• 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
    » Same type of computation as previous scheme
  – Possible behavior: 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?

• Establish “acceptable” page-fault rate
  – If actual rate too low, process loses frame
  – If actual rate too high, process gains frame

• Question: What if we just don’t have enough memory?
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

Δ ≡ working-set window ≡ fixed number of page references
  – Example: 10,000 instructions

WSᵢ (working set of Process Pᵢ) = total set of pages referenced in the most recent Δ (varies in time)
  – if Δ too small will not encompass entire locality
  – if Δ too large will encompass several localities
  – if Δ = ∞ ⇒ will encompass entire program

D = Σ|WSᵢ| ≡ 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)
Linux Virtual memory map (Pre-Meltdown)

32-Bit Virtual Address Space

- Kernel Addresses
  - 0x00000000
  - 0xFFFFFFFF
  - 1GB
- User Addresses
  - 0x00000000
  - 0xC0000000
  - 896MB

64-Bit Virtual Address Space

- Kernel Addresses
  - 0x0000000000000000
  - 0xFFFFFFFF
  - 3GB Total
- Empty Space
  - 0xFFFF800000000000
  - 64 TiB
- User Addresses
  - 0x0000000000000000
  - 0xFFFFFFFF
  - 128TiB

Physical Memory

- 1GB
- 896MB
- 32-Bit
- 64-Bit
- 64 TiB
- 128TiB
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 0xFFFFF800000000000
Post Meltdown Memory Map

- Meltdown flaw (2018, Intel x86, IBM Power, ARM)

- **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
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?
Next Objective
OS Basics: I/O
Recall: I/O is at very different timescale

And Range of Timescales

Jeff Dean: "Numbers Everyone Should Know"

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000 ns</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000 ns</td>
</tr>
<tr>
<td>Send packet CA--&gt;Netherlands--&gt;CA</td>
<td>150,000,000 ns</td>
</tr>
</tbody>
</table>
In a Picture

- I/O devices you recognize are supported by I/O Controllers
- Processors accesses them by reading and writing I/O registers as if they were memory
  – Write commands and arguments, read status and results
What’s a bus?

• Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
  – Operations: e.g., Read, Write
  – Control lines, Address lines, Data lines
  – Typically multiple devices
• Protocol: initiator requests access, arbitration to grant, identification of recipient, handshake to convey address, length, data
• Very high BW close to processor (wide, fast, and inflexible), low BW with high flexibility out in I/O subsystem
Example: PCI Architecture

- RAM
- CPU
- Memory Bus
- Host Bridge
- ISA Bridge
- PCI Bridge
- ISA Controller
- Legacy Devices
- PCI Slots
- USB Controller
- SATA Controller
- Root Hub
- Hub
- Webcam
- Mouse
- Keyboard
- Scanner
- Hard Disk
- DVD ROM
Chip-scale Features of 2015 x86 (Sky Lake)

- Significant pieces:
  - Four OOO cores with deeper buffers
    » Intel MPX (Memory Protection Extensions)
    » Intel SGX (Software Guard Extensions)
    » Issue up to 6 \( \mu \)-ops/cycle
  - 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)

Sky Lake System Configuration
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
  – 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
• 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):**
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):**

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

Diagram:
- CPU
- DMA/bus/interrupt controller
- cache
- CPU memory bus
- memory
- buffer
- IDE disk controller
- disk
- PCI bus
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

User Program:
- User process
  - request I/O
    - system call
      - can already satisfy request?
        - yes
          - kernel I/O subsystem
            - transfer data (if appropriate) to process, return completion or error code
        - no
          - kernel I/O subsystem
            - send request to device driver, block process if appropriate
            - device driver
              - process request, issue commands to controller, configure controller to block until interrupted
              - device-controller commands
                - interrupt
                  - device controller
                    - monitor device, interrupt when I/O completed
        - return from system call
          - I/O completed, input data available, or output completed

Kernel I/O Subsystem:
- transfer data (if appropriate) to process, return completion or error code

Device Driver Top Half:
- determine which I/O completed, indicate state change to I/O subsystem

Device Driver Bottom Half:
- receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

Device Hardware:
- I/O completed, generate interrupt

3/19/20
Kubiatowicz CS162 ©UCB Spring 2020
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 + $b$/TransferCapacity
Example (Fast Network)

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

- Latency: $b/(S + b/B) = B\cdot b/(B\cdot S + b) = B/(B\cdot S/b + 1)$

- Bandwidth: $b/(S + b/B) = B\cdot b/(B\cdot S + b) = B/(B\cdot S/b + 1)$
Example (Fast Network)

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

- Half-power point occurs at $b = S \times B = 125,000$ bytes
Example: at 10 ms startup (like Disk)

Performance of Gbps link with 10 ms startup

Half-power $b = 1,250,000$ bytes!
What Determines Peak BW for I/O?

• Bus Speed
  – PCI-X: 1064 MB/s = 133 MHz × 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…
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