

CSC 112: Computer Operating Systems

Lecture 17

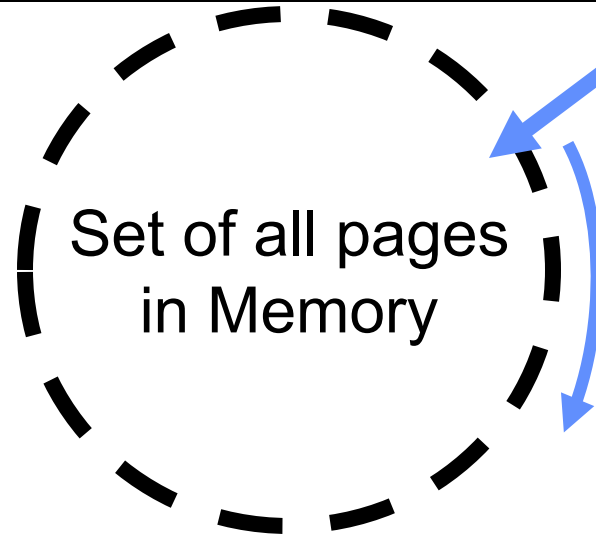
Demand Paging (Finished),
General I/O, Storage Devices

Department of Computer Science,
Hofstra University

Recall: Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! (“Effective Access Time”)
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
 - $EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:
$$\begin{aligned} EAT &= 200\text{ns} + p \times 8 \text{ ms} \\ &= 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2 \mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $EAT < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!

Recall: Clock Algorithm (Not Recently Used)



Single Clock Hand:

Advances only on page fault!

Check for pages not used recently

Mark pages as not used recently

- **Clock Algorithm:** Arrange physical pages in circle with single clock hand
 - Approximate LRU (*approximation to approximation to MIN*)
 - Replace **an** old page, not **the oldest** page
- Details:
 - Hardware “**use**” bit per physical page (called “**accessed**” in Intel architecture):
 - » Hardware sets **use** bit on each reference
 - » If **use** bit isn’t set, means not referenced in a long time
 - » Some hardware sets **use** bit in the TLB; must be copied back to page TLB entry gets replaced
 - On page fault:
 - » Advance clock hand (not real time)
 - » Check **use** bit: 1→ used recently; clear and leave alone
0→ selected candidate for replacement

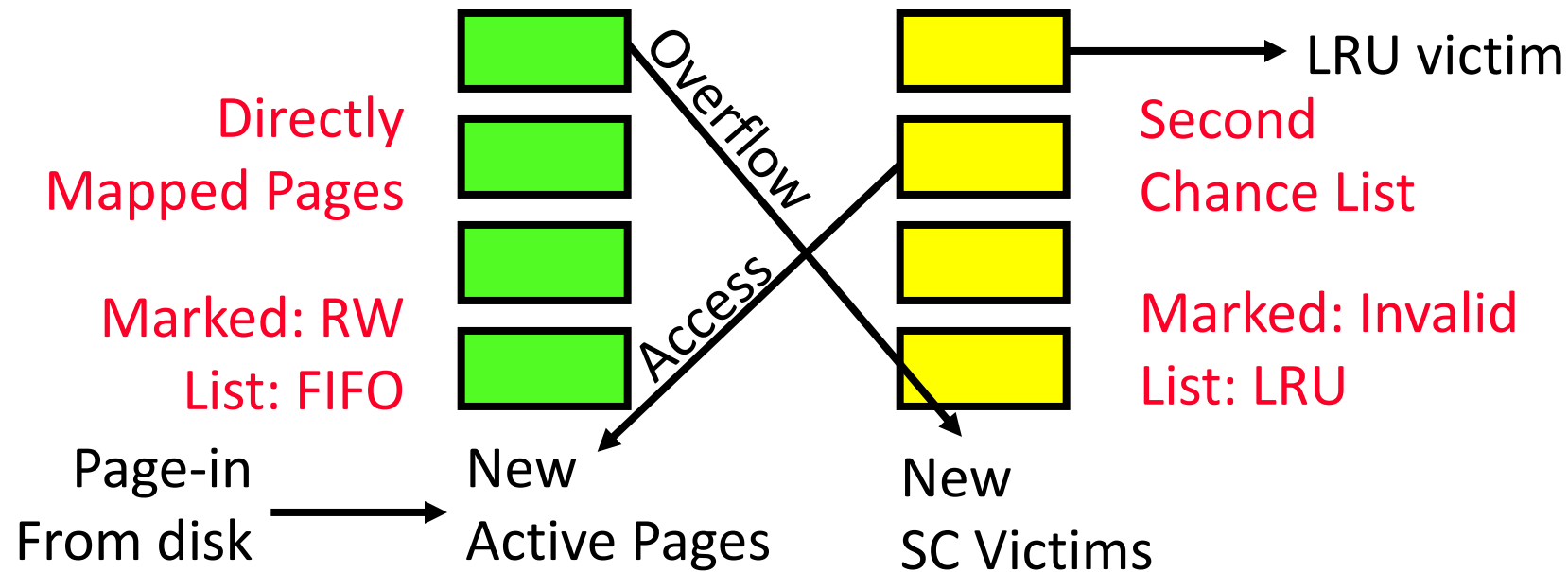
Recall: Meaning of PTE bits

- Which bits of a PTE entry are useful to us for the Clock Algorithm? Remember Intel PTE:



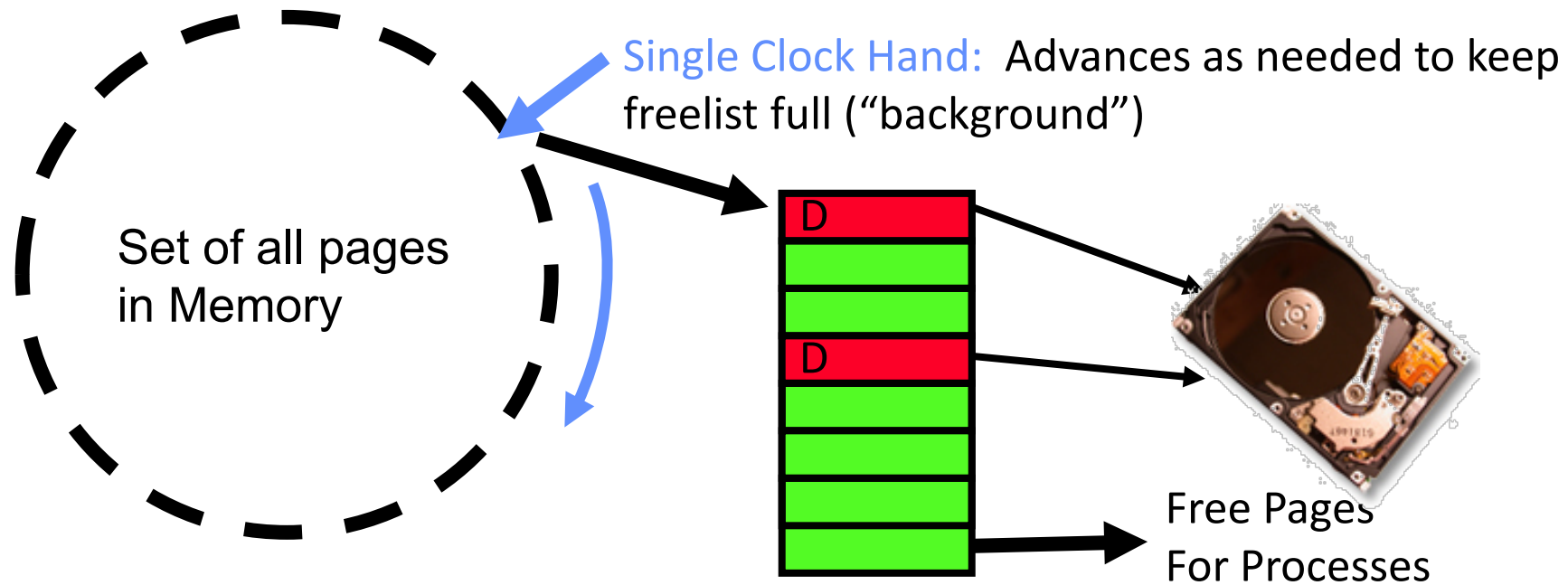
- The “**P**resent” bit (called “**V**alid” elsewhere):
 - » $P=0$: Page is invalid and a reference will cause page fault
 - » $P=1$: Page frame number is valid and MMU is allowed to proceed with translation
- The “**W**ritable” bit (could have opposite sense and be called “**R**ead-only”):
 - » $W=0$: Page is read-only and cannot be written.
 - » $W=1$: Page can be written
- The “**A**ccessed” bit (called “**U**se” elsewhere):
 - » $A=0$: Page has not been accessed (or used) since last time software set $A \rightarrow 0$
 - » $A=1$: Page has been accessed (or used) since last time software set $A \rightarrow 0$
- The “**D**irty” bit (called “**M**odified” elsewhere):
 - » $D=0$: Page has not been modified (written) since PTE was loaded
 - » $D=1$: Page has changed since PTE was loaded

Recall: 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

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”)

- When evicting a page frame, how to know which PTEs to invalidate?
 - Hard in the presence of shared pages (forked processes, shared memory, ...)
- Reverse mapping mechanism 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)

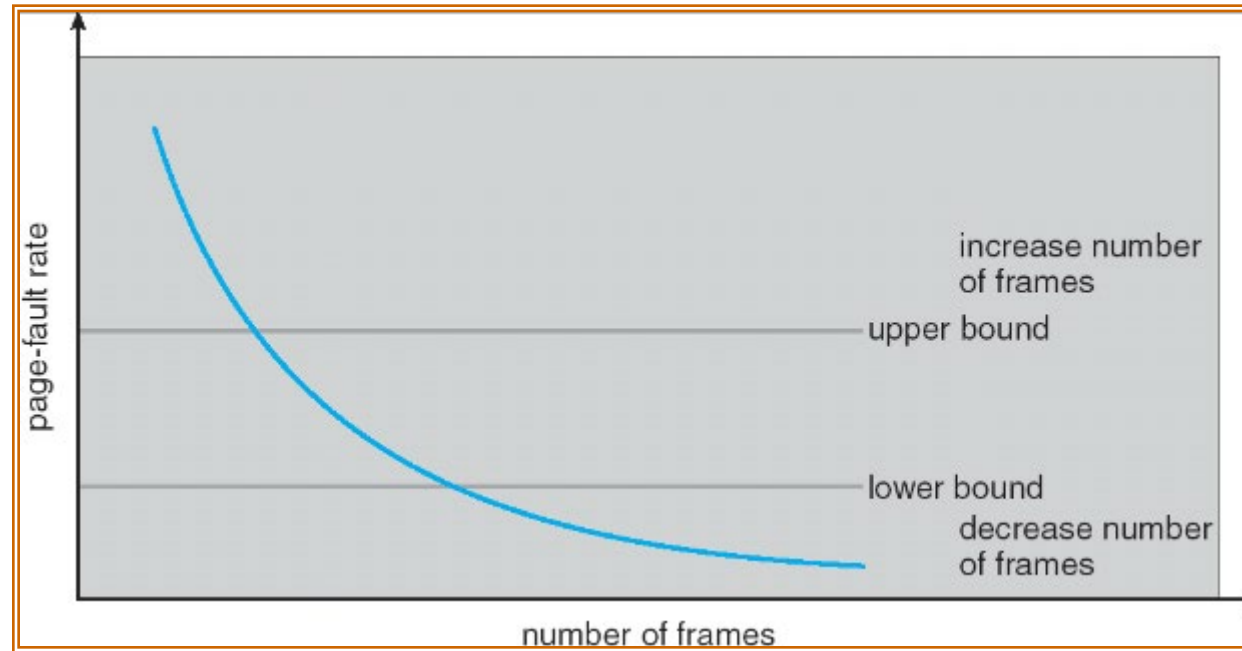
- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
 - Want to make sure that all processes **that are loaded into memory** 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 = size of process p_i and $S = \sum s_i$
 - m = total number of physical frames in the system
 - a_i = (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

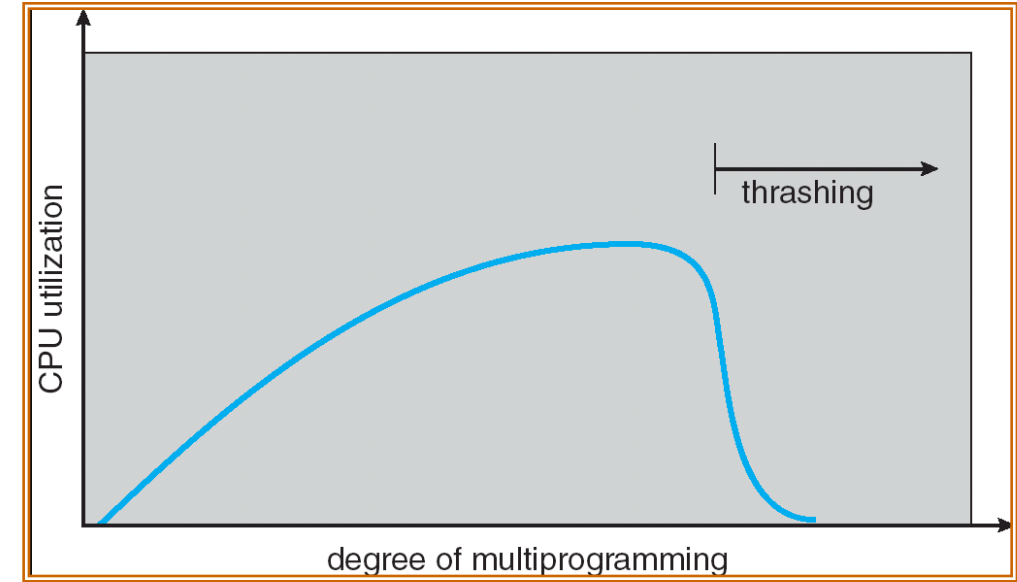
- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- 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?

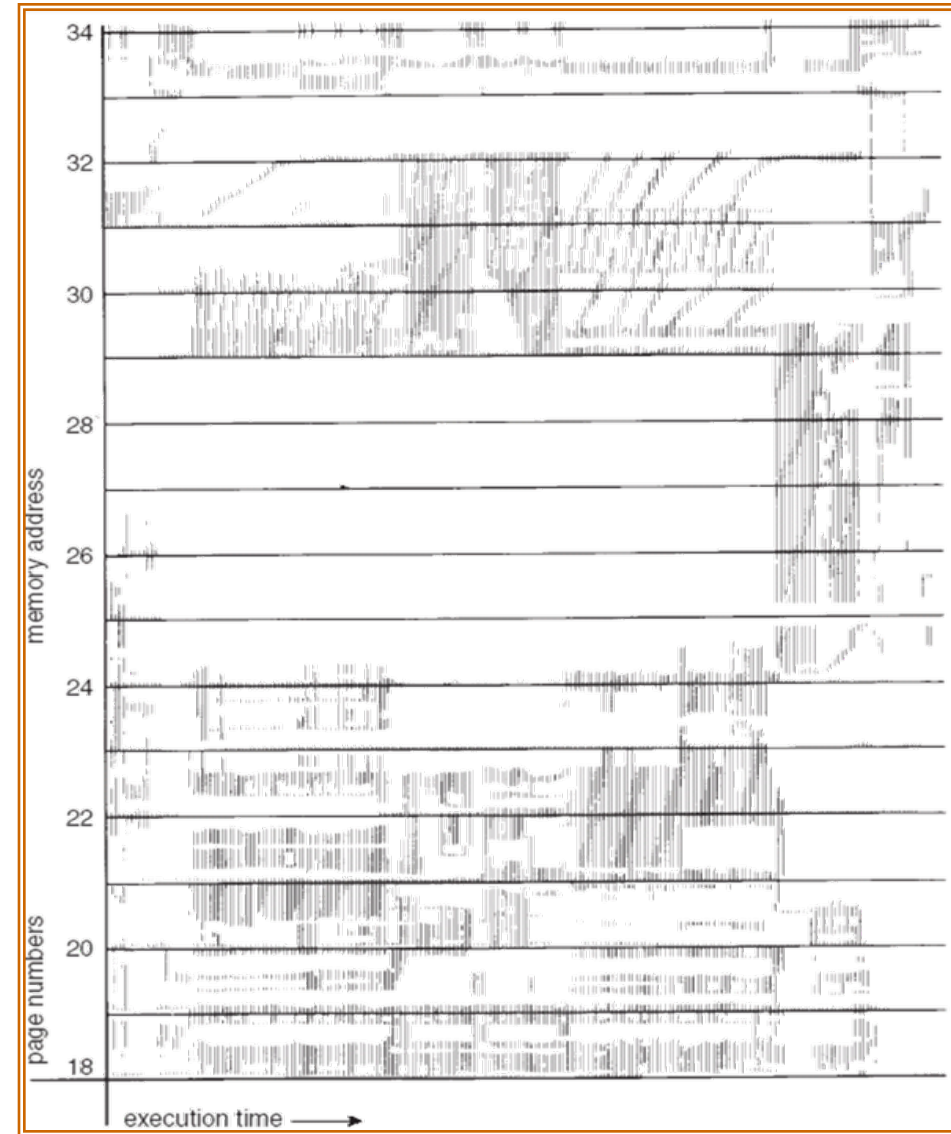
Thrashing

- 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** \equiv 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?

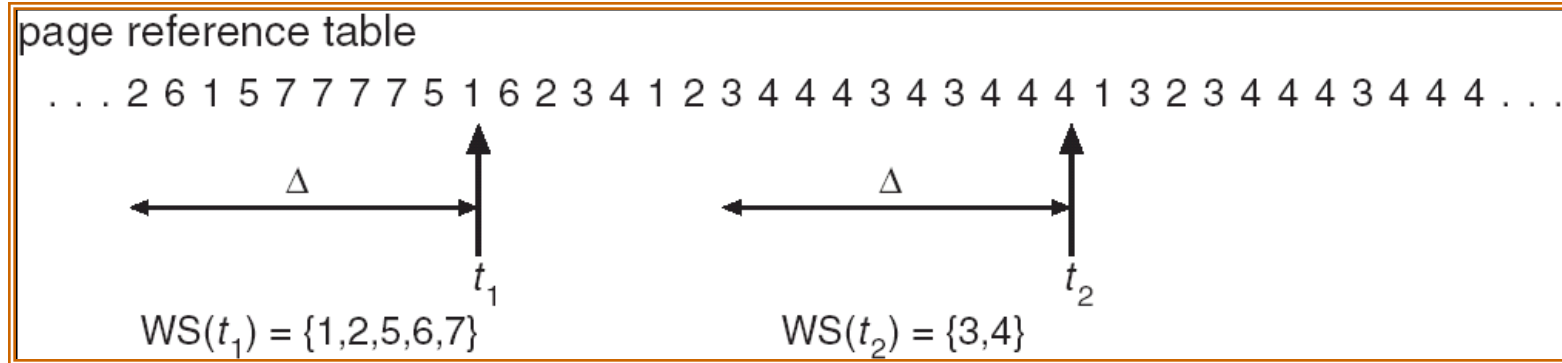


Locality In A Memory-Reference Pattern

- 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 for process to behave well
- Not enough memory for Working Set \Rightarrow Thrashing
 - Better to swap out process?



Working-Set Model

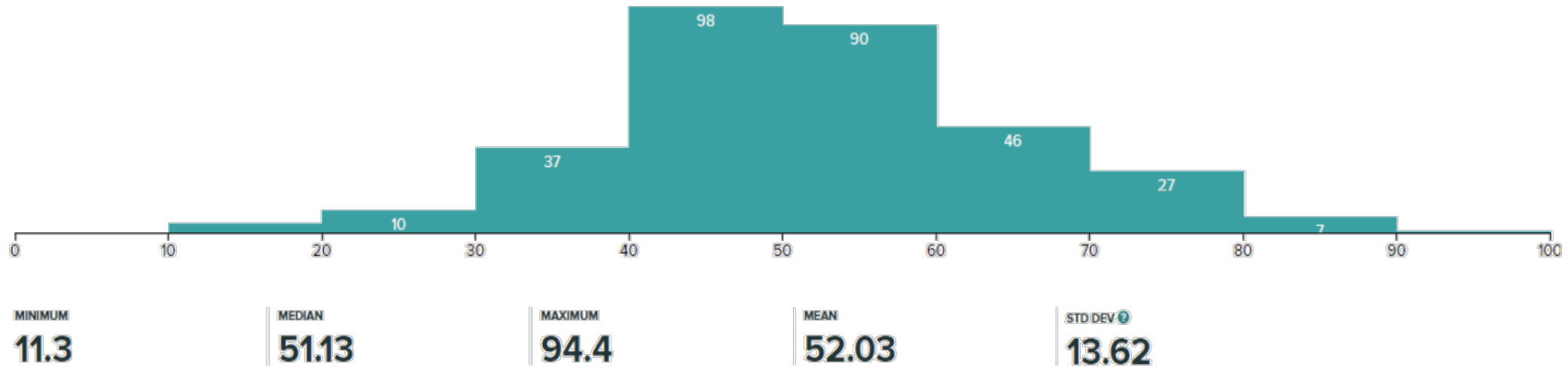


- $\Delta \equiv$ working-set window \equiv fixed number of page references
 - Example: 10,000 instructions
- W_{Si} (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum |W_{Si}| \equiv$ total demand frames
- if $D > m \Rightarrow$ 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

Administrivia



- Midterm 2 Graded!
 - Mean: 52.03, Std Dev: 13.62, Max: 94.4
 - Regrade requests due by tomorrow (March 30th)
- Project 2 is due Saturday (April 2nd)
 - Including group evaluations!
- Project 3 released Sunday (April 3rd)

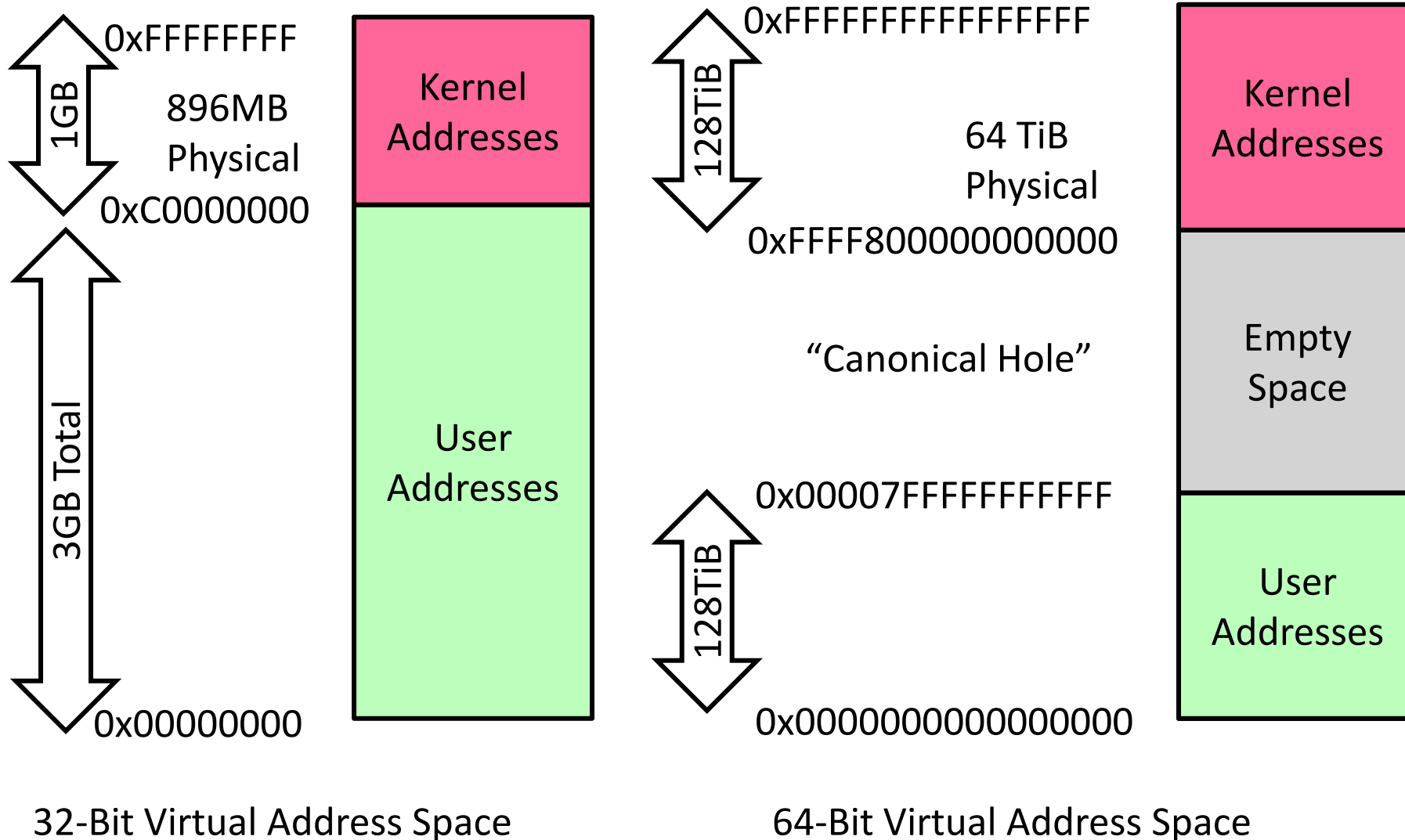
Administrivia (Con't)

- You need to know your units as CS/Engineering students!
- Units of Time: “s”: Second, “min”: 60s, “h”: 3600s, (of course)
 - Millisecond: $1\text{ms} \Rightarrow 10^{-3}\text{s}$
 - Microsecond: $1\mu\text{s} \Rightarrow 10^{-6}\text{s}$
 - Nanosecond: $1\text{ns} \Rightarrow 10^{-9}\text{s}$
 - Picosecond: $1\text{ps} \Rightarrow 10^{-12}\text{s}$
- Integer Sizes: “b” \Rightarrow “bit”, “B” \Rightarrow “byte” == 8 bits, “W” \Rightarrow “word” ==? (depends. Could be 16b, 32b, 64b)
- Units of Space (memory), sometimes called the “binary system”
 - Kilo: $1\text{KB} \equiv 1\text{KiB} \Rightarrow 1024\text{ bytes} == 2^{10}\text{ bytes} == 1024 \approx 1.0 \times 10^3$
 - Mega: $1\text{MB} \equiv 1\text{MiB} \Rightarrow (1024)^2\text{ bytes} == 2^{20}\text{ bytes} == 1,048,576 \approx 1.0 \times 10^6$
 - Giga: $1\text{GB} \equiv 1\text{GiB} \Rightarrow (1024)^3\text{ bytes} == 2^{30}\text{ bytes} == 1,073,741,824 \approx 1.1 \times 10^9$
 - Tera: $1\text{TB} \equiv 1\text{TiB} \Rightarrow (1024)^4\text{ bytes} == 2^{40}\text{ bytes} == 1,099,511,627,776 \approx 1.1 \times 10^{12}$
 - Peta: $1\text{PB} \equiv 1\text{PiB} \Rightarrow (1024)^5\text{ bytes} == 2^{50}\text{ bytes} == 1,125,899,906,842,624 \approx 1.1 \times 10^{15}$
 - Exa: $1\text{EB} \equiv 1\text{EiB} \Rightarrow (1024)^6\text{ bytes} == 2^{60}\text{ bytes} == 1,152,921,504,606,846,976 \approx 1.2 \times 10^{18}$
- Units of Bandwidth, Space on disk/etc, Everything else..., sometimes called the “decimal system”
 - Kilo: $1\text{KB/s} \Rightarrow 10^3\text{ bytes/s}, 1\text{KB} \Rightarrow 10^3\text{ bytes}$
 - Mega: $1\text{MB/s} \Rightarrow 10^6\text{ bytes/s}, 1\text{MB} \Rightarrow 10^6\text{ bytes}$
 - Giga: $1\text{GB/s} \Rightarrow 10^9\text{ bytes/s}, 1\text{GB} \Rightarrow 10^9\text{ bytes}$
 - Tera: $1\text{TB/s} \Rightarrow 10^{12}\text{ bytes/s}, 1\text{TB} \Rightarrow 10^{12}\text{ bytes}$
 - Peta: $1\text{PB/s} \Rightarrow 10^{15}\text{ bytes/s}, 1\text{PB} \Rightarrow 10^{15}\text{ bytes}$
 - Exa: $1\text{EB/s} \Rightarrow 10^{18}\text{ bytes/s}, 1\text{EB} \Rightarrow 10^{18}\text{ bytes}$

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)



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 {                  // ... catch and ignore SIGSEGV (illegal access)
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

Recall: Five Components of a Computer

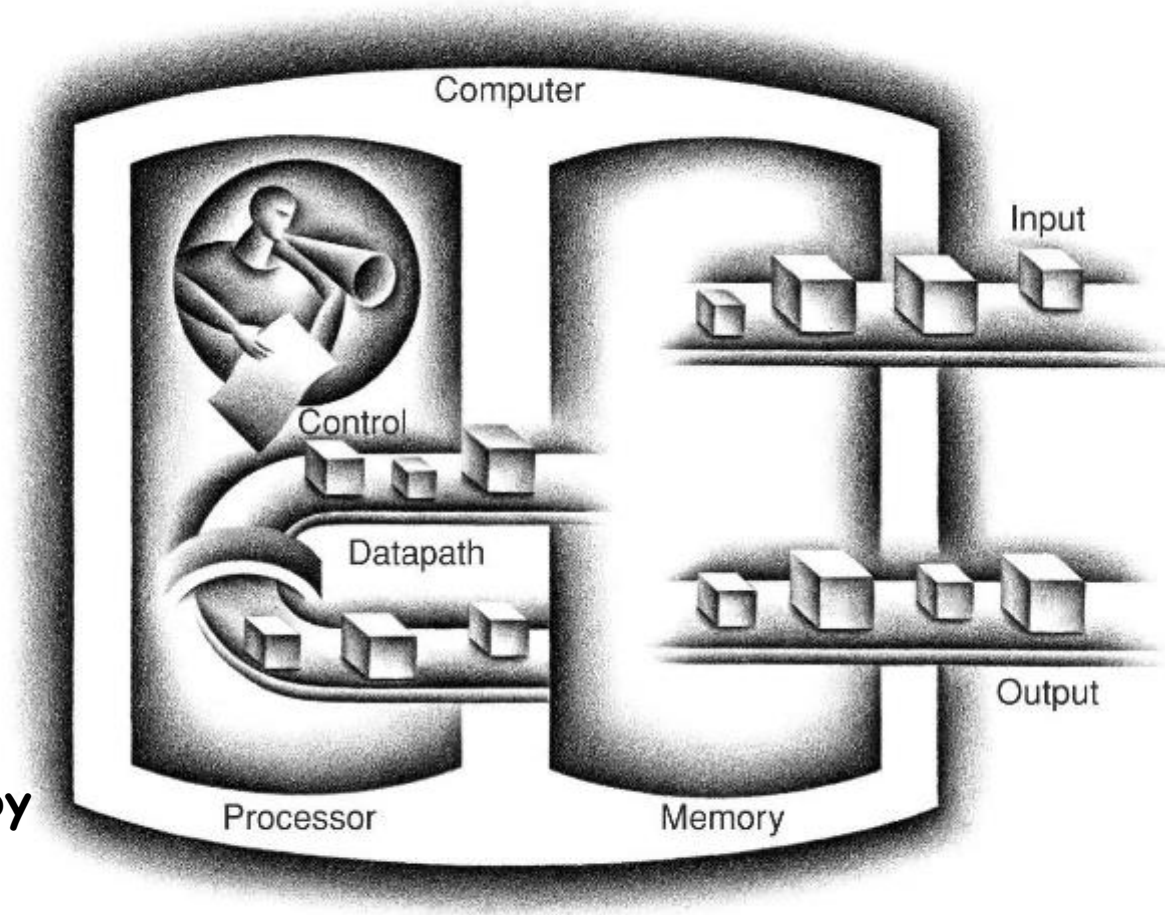
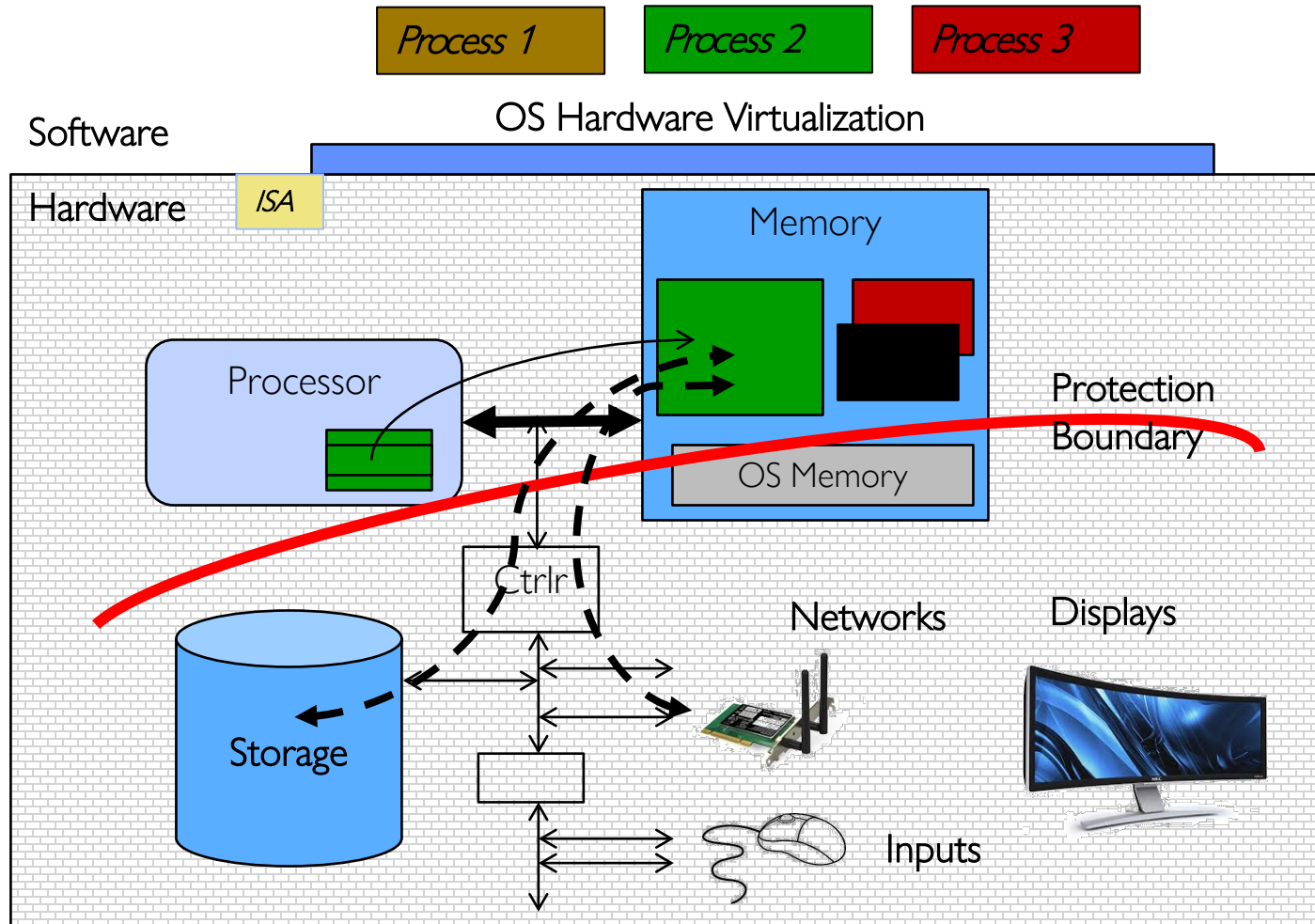


Diagram from "Computer Organization and Design" by Patterson and Hennessy

Requirements of I/O

- So far in CS 162, we have studied:
 - Abstractions: the APIs provided by the OS to applications running in a process
 - Synchronization/Scheduling: How to manage the CPU
- 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?

Recall: OS Basics: I/O



- OS provides common services in form of I/O

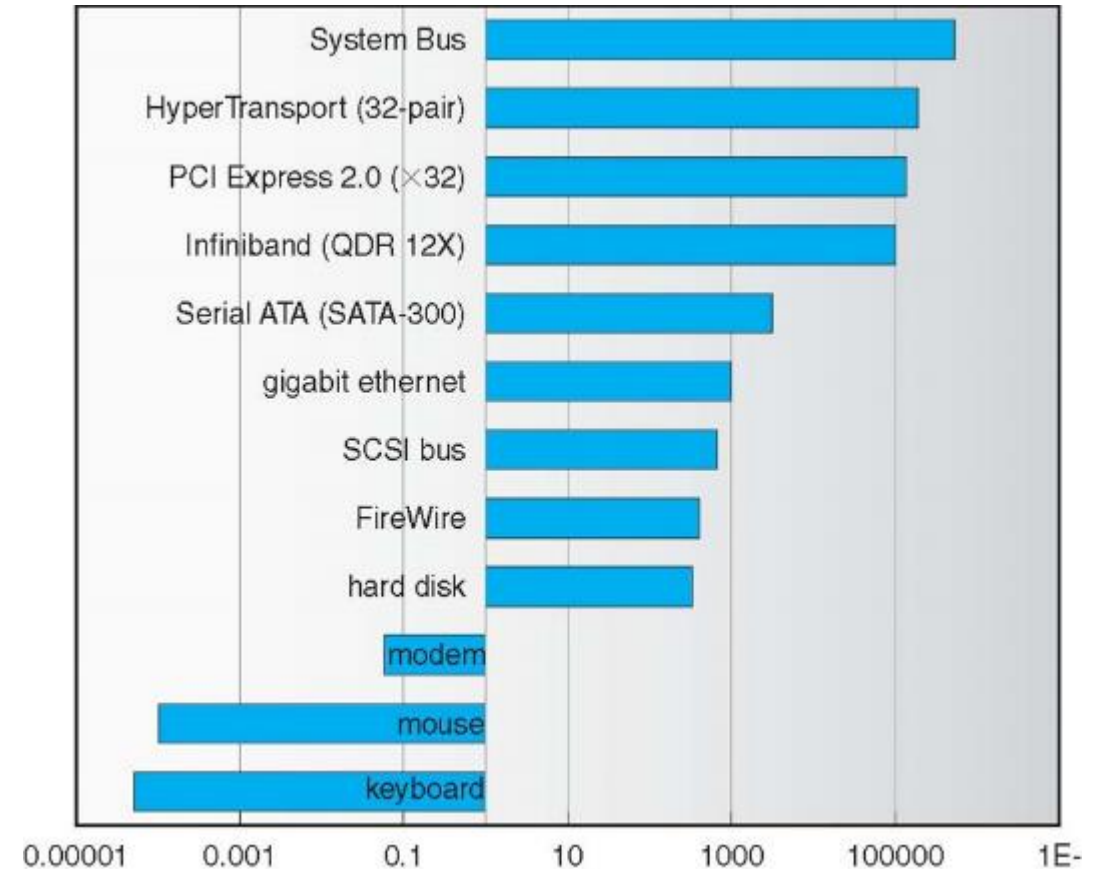
Recall: Range of Timescales

**Jeff Dean:
“Numbers
Everyone Should
Know”**

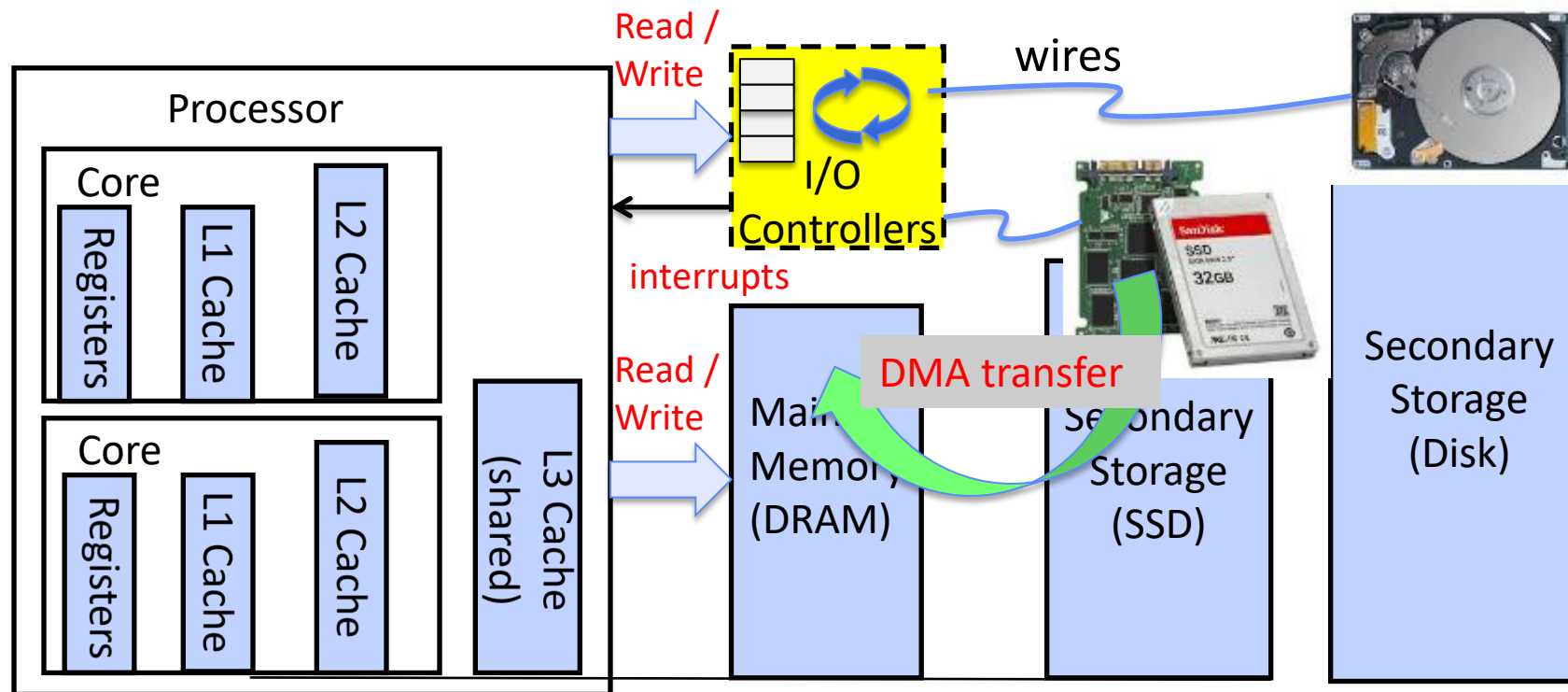
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

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

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

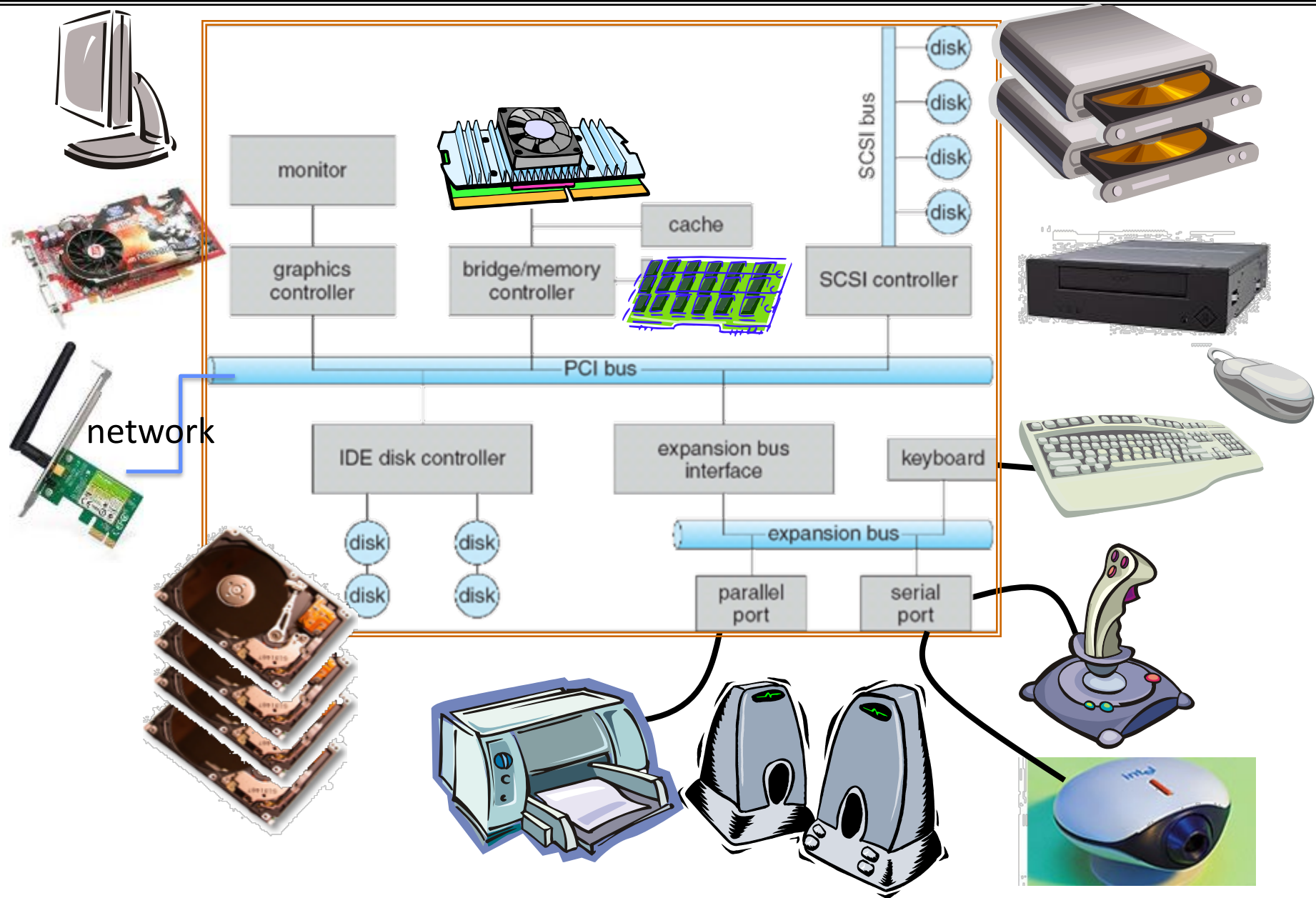


In a Picture

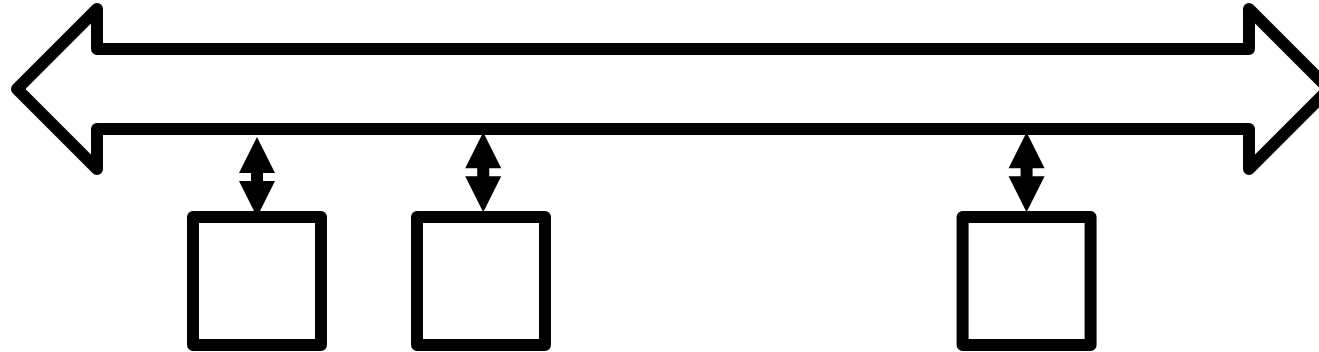


- I/O devices you recognize are supported by I/O Controllers
- Processors access them by reading and writing I/O registers as if they were memory
 - Write commands and arguments, read status and results

Modern I/O Systems



What's a bus?

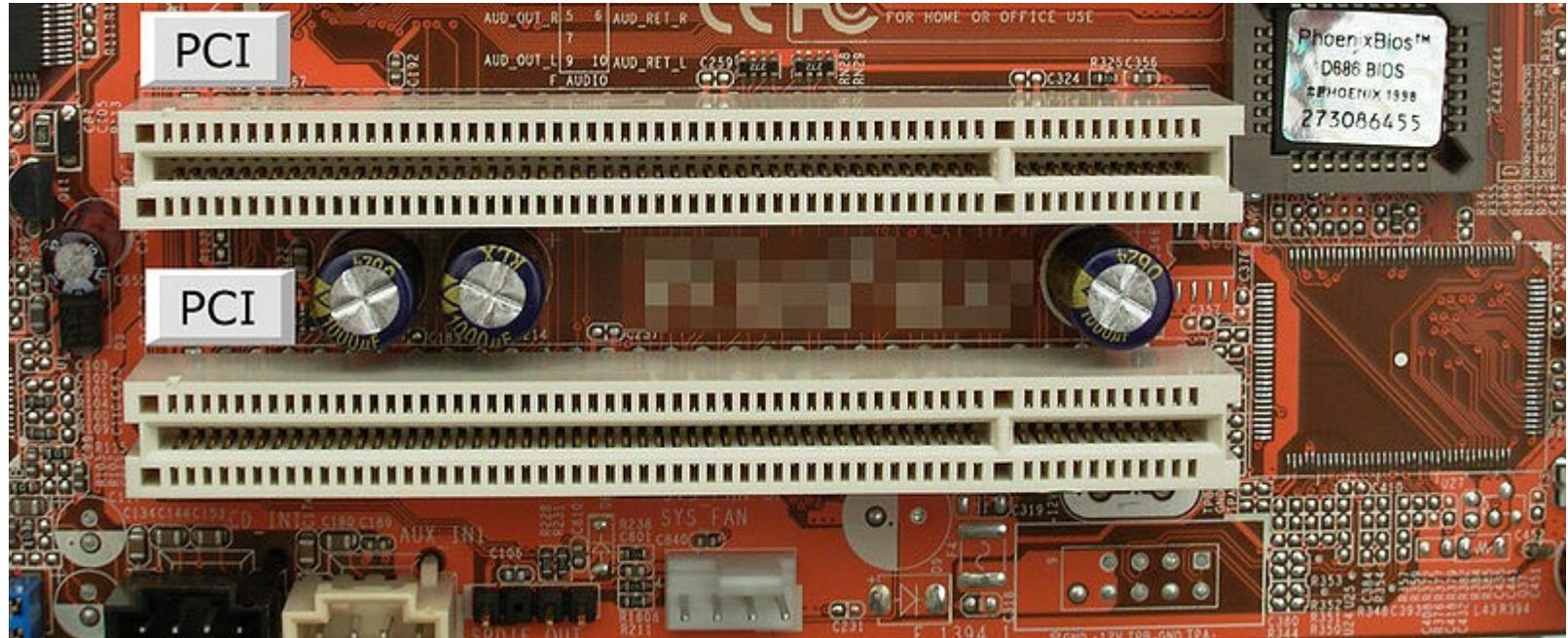


- Common set of wires for communication among hardware devices plus protocols for carrying out data transfer transactions
 - Operations: e.g., Read, Write
 - Control lines, Address lines, Data lines
 - Typically multiple devices
- Protocol: initiator requests access, arbitration to grant, identification 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

Why a Bus?

- Buses let us connect n devices over a single set of wires, connections, and protocols
 - $O(n^2)$ relationships with 1 set of wires (!)
- Downside: Only one transaction at a time
 - The rest must wait
 - “Arbitration” aspect of bus protocol ensures the rest wait

PCI Bus Evolution



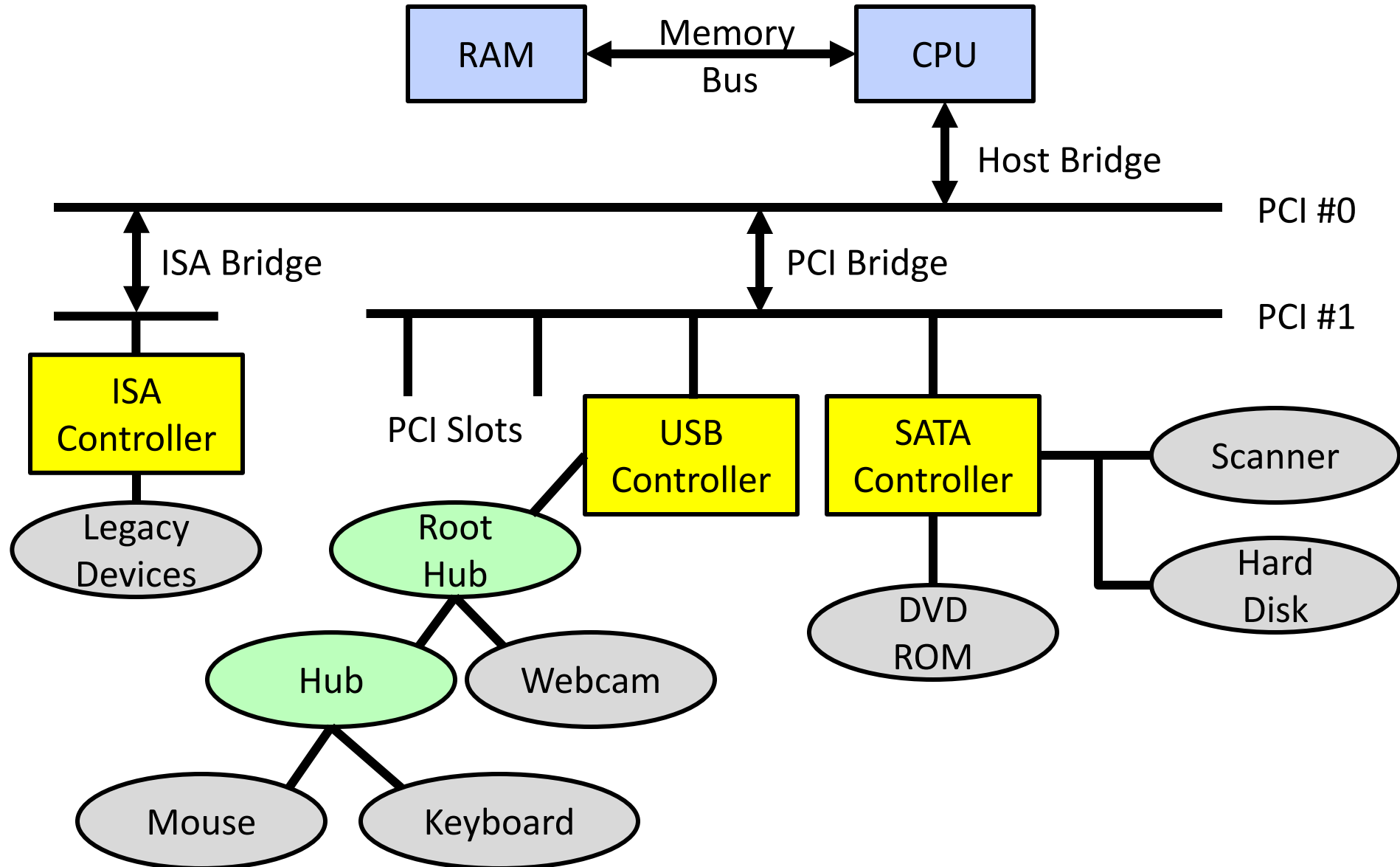
- PCI started life out as a bus
- But a parallel bus has many limitations
 - Multiplexing address/data for many requests
 - Slowest devices must be able to tell what's happening (e.g., for arbitration)
 - Bus speed is set to that of the slowest device

PCI Express “Bus”

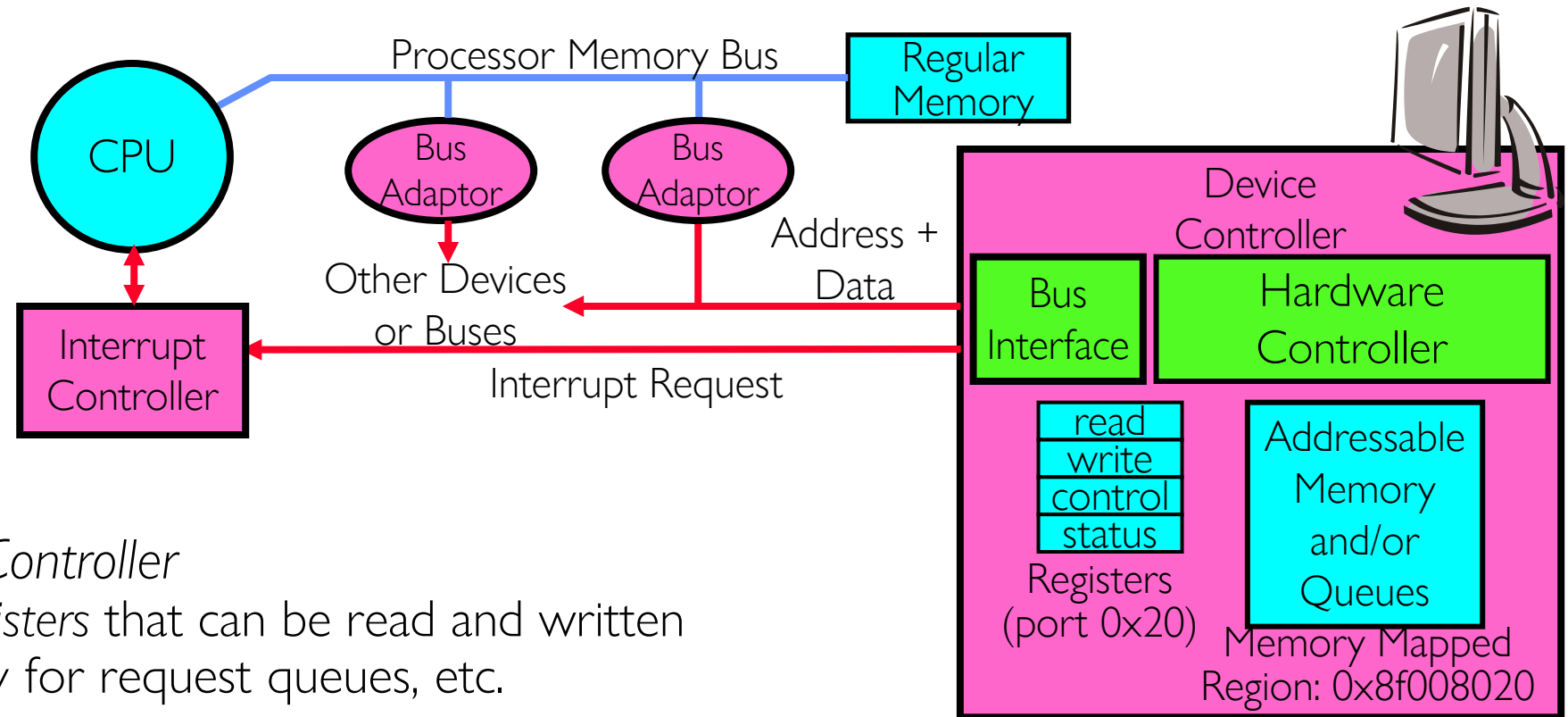
- No longer a parallel bus
- Really a **collection of fast serial channels** or “lanes”
- Devices can use as many as they need to achieve a desired bandwidth
- Slow devices don’t have to share with fast ones

- One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI Express
 - The physical interconnect changed completely, but the old API still worked

Example: PCI Architecture



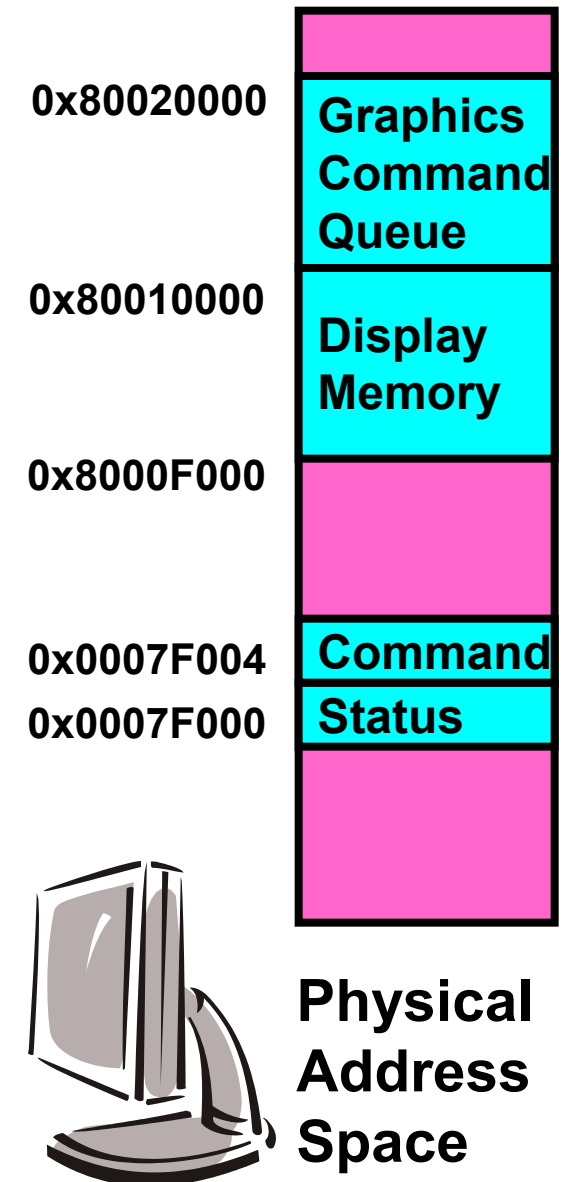
How does the Processor Talk to the Device?

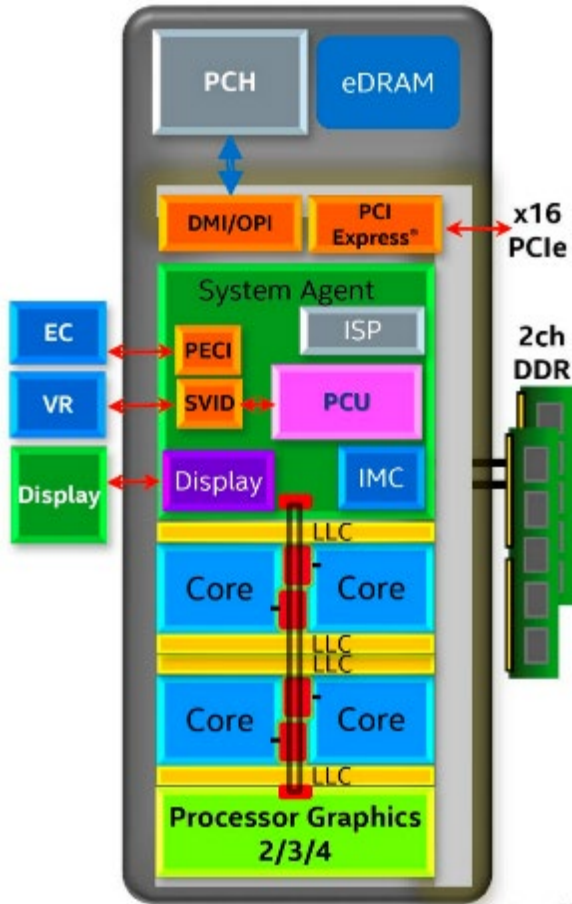



- CPU interacts with a *Controller*
 - Contains a set of *registers* that can be read and written
 - May contain memory for request queues, etc.
- Processor accesses registers in two ways:
 - **Port-Mapped I/O**: in/out instructions
 - » Example from the Intel architecture: **out 0x21, AL**
 - **Memory-mapped 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”)
 - » 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

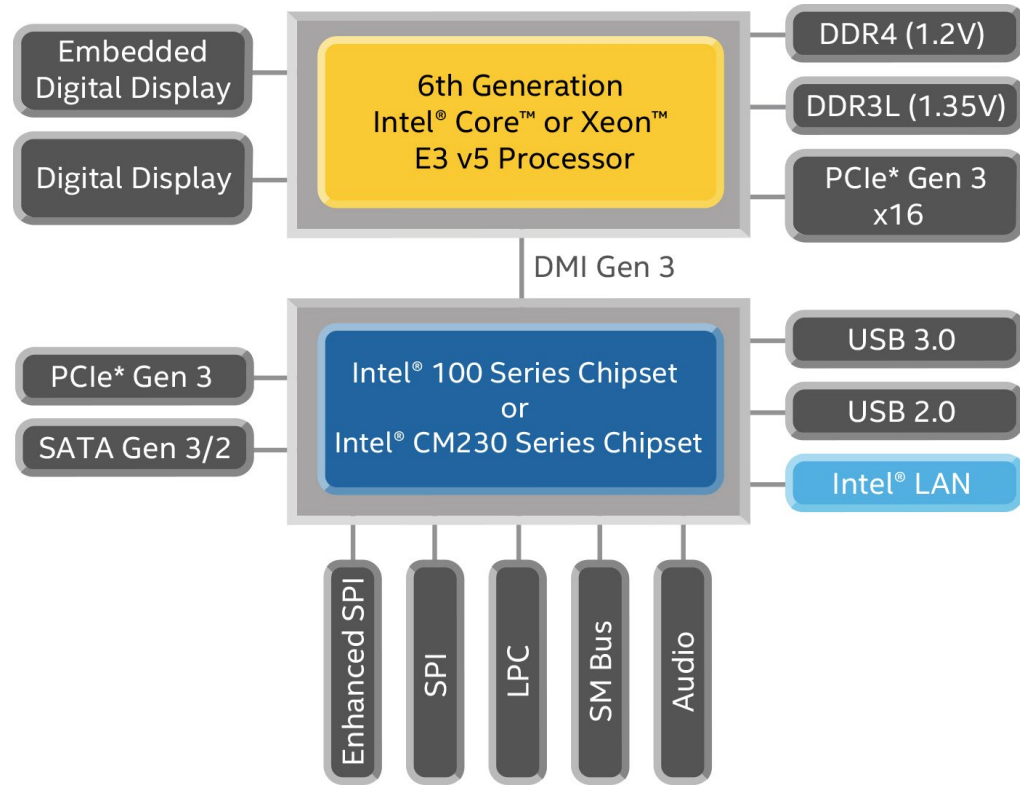




- Significant pieces:
 - Four OOO cores with deeper buffers
 - » Intel MPX (Memory Protection Extensions)
 - » Intel SGX (Software Guard Extensions)
 - » Issue up to 6 μ -ops/cycle
 - GPU, System Agent (Mem, Fast I/O)
 - Large shared L3 cache with on-chip ring bus
 - » 2 MB/core instead of 1.5 MB/core
 - » 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)
- 
- A photograph of an Intel Core i7-6700K processor. The processor is a square, silver-colored integrated circuit mounted on a green printed circuit board (PCB). The top surface of the processor is marked with the Intel logo, "INTEL® CORE™ i7", "i7-6700K", "SR2L0 4.00GHZ", and "X611A978 (e4)". The processor is connected to the PCB via numerous gold-plated pins visible around its perimeter.



Sky Lake I/O: PCH



Sky Lake System Configuration

- 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)

Operational Parameters for I/O

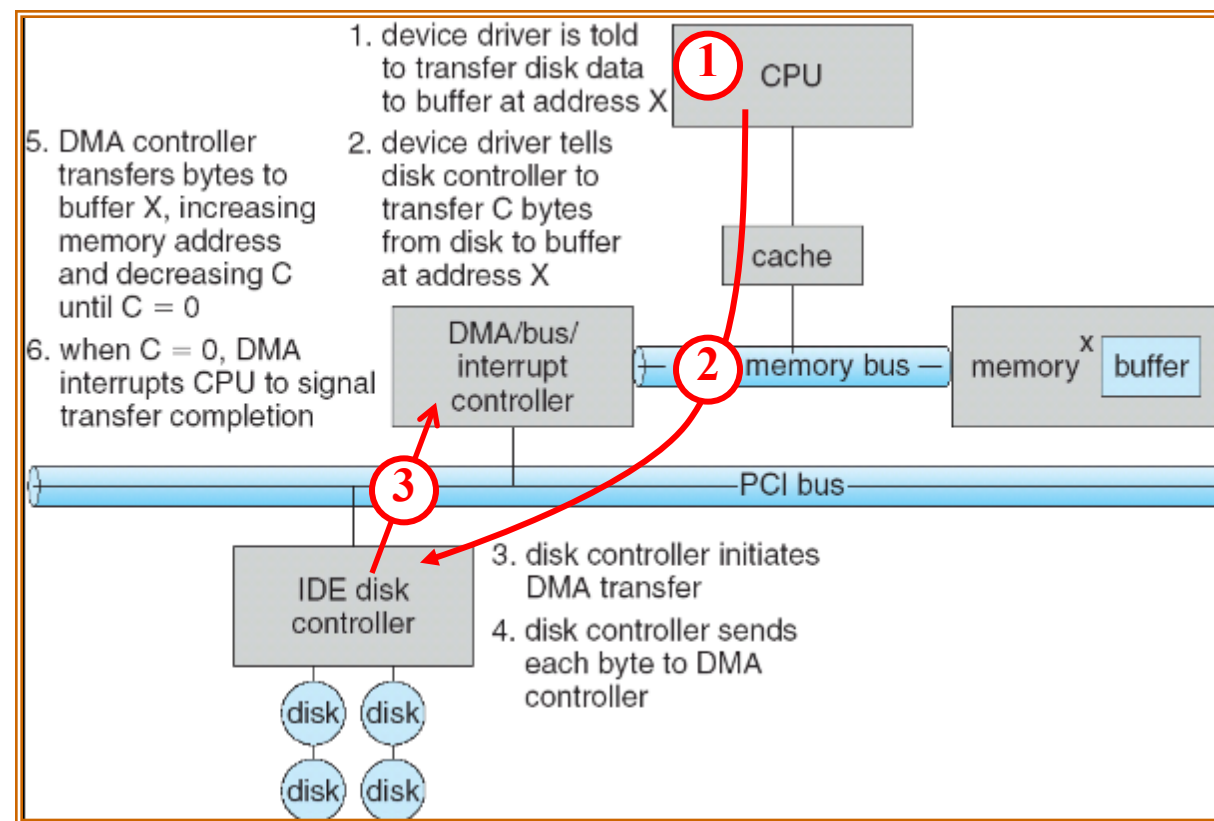
- Data granularity: Byte vs. Block
 - Some devices provide single byte at a time (e.g., keyboard)
 - Others provide whole blocks (e.g., disks, networks, etc.)
- Access pattern: Sequential vs. Random
 - Some devices must be accessed sequentially (e.g., tape)
 - Others can be accessed “randomly” (e.g., 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

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):

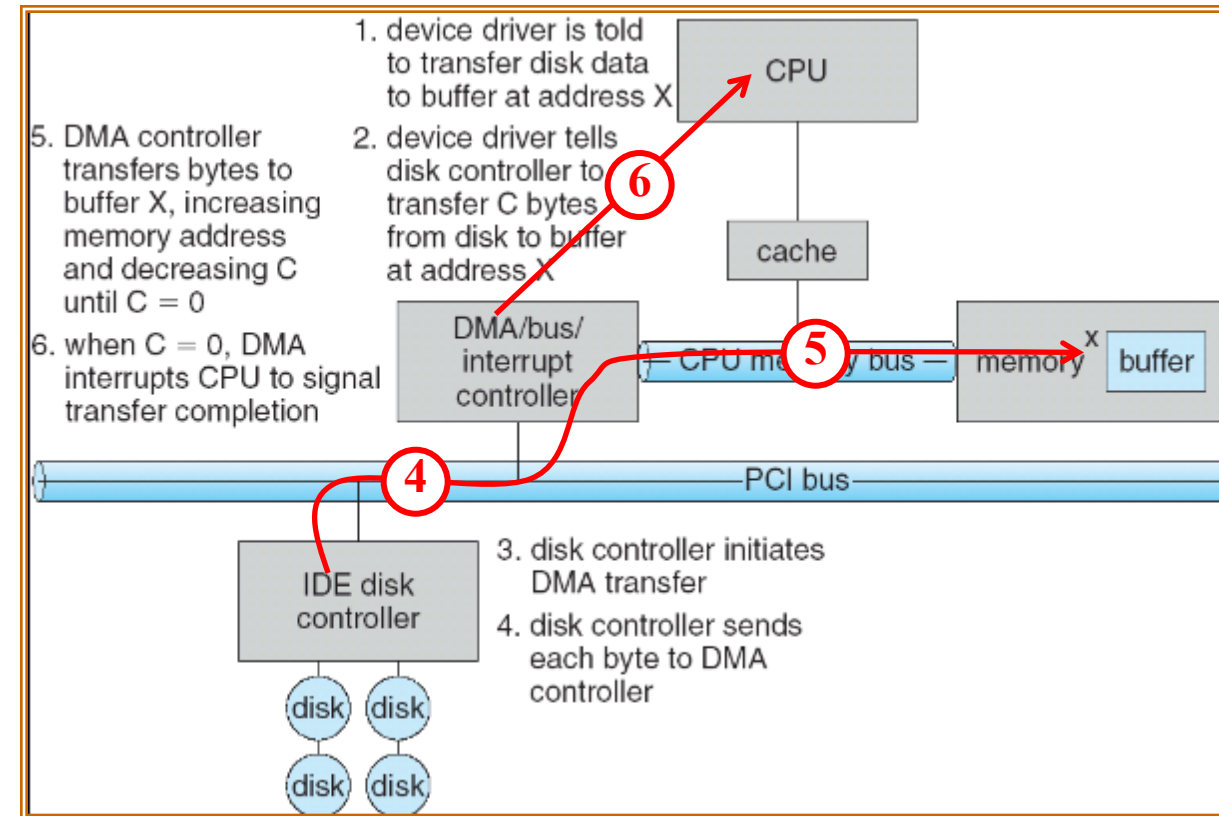


Transferring Data To/From Controller

- **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

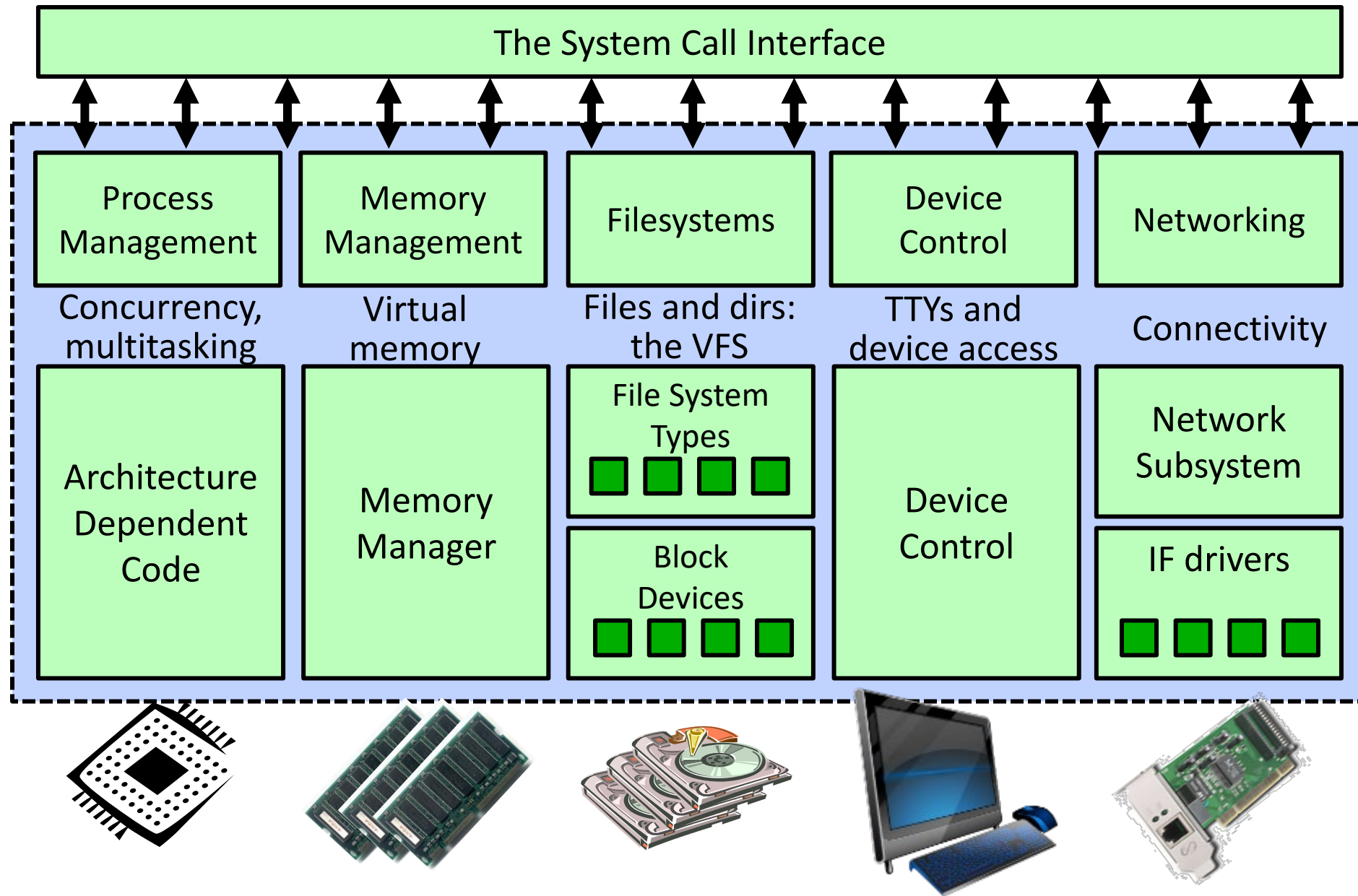
- Sample interaction with DMA controller (from OSC book):



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

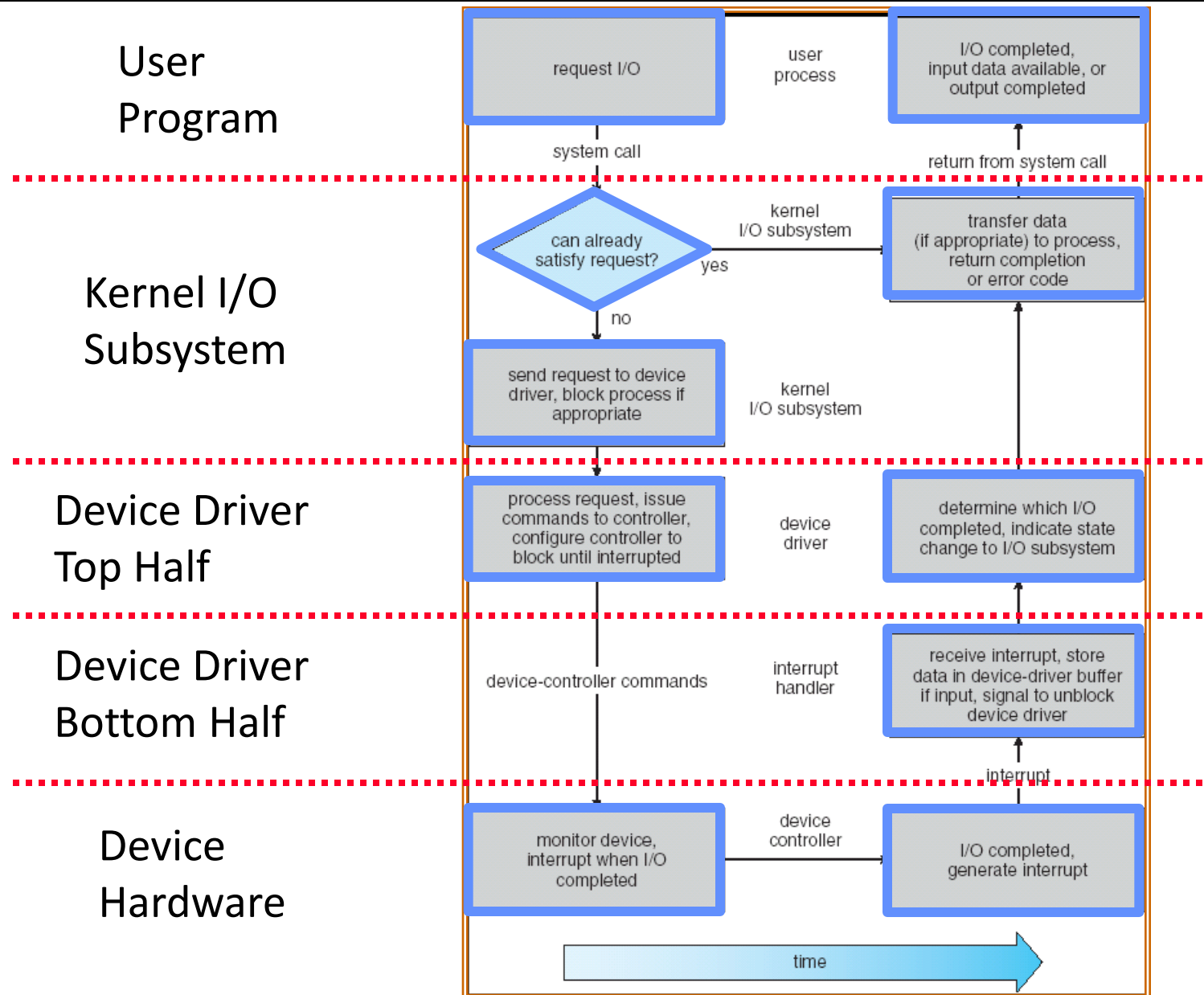
Kernel Device Structure



Recall: 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

Recall: Life Cycle of An I/O Request



Conclusion

- 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