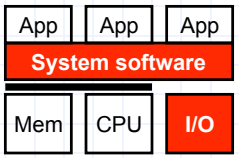
CIS 371
Computer Organization and Design

Unit 11: Virtual Memory & I/O

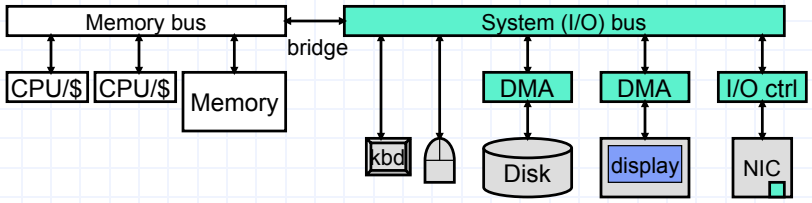
This Unit: Virtualization and the OS



- The operating system (OS)
  - A super-application
  - Hardware support for an OS
- Virtual memory
  - Page tables and address translation
  - TLBs and memory hierarchy issues
- I/O
  - Devices and buses
  - Polling and interrupts
  - DMA

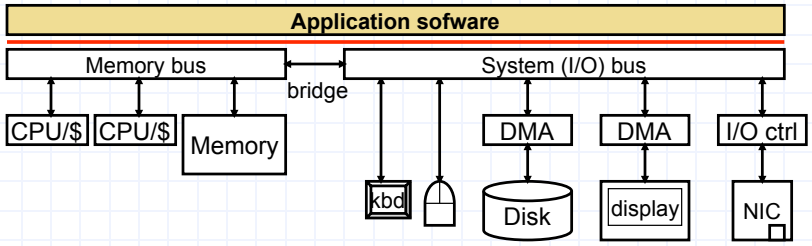
A Computer System: Hardware

- CPUs and memories
  - Connected by memory bus
- **I/O peripherals**: storage, input, display, network, …
  - With separate or built-in DMA
  - Connected by **system bus** (which is connected to memory bus)



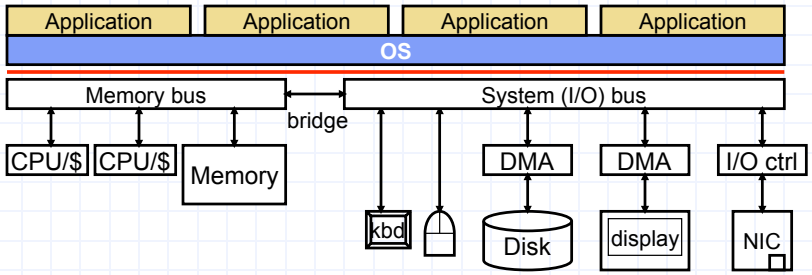
A Computer System: + App Software

- **Application software**: computer must do something



A Computer System: + OS

- **Operating System (OS):** virtualizes hardware for apps
  - **Abstraction**: provides **services** (e.g., threads, files, etc.)
    + Simplifies app programming model, raw hardware is nasty
  - **Isolation**: gives each app illusion of private CPU, memory, I/O
    + Simplifies app programming model
    + Increases hardware resource utilization



Operating System (OS) and User Apps

- Sane system development requires a split
  - Hardware itself facilitates/enforces this split
- **Operating System (OS)**: a super-privileged process
  - Manages hardware resource allocation/revocation for all processes
  - Has direct access to resource allocation features
  - Aware of many nasty hardware details
  - Aware of other processes
  - Talks directly to input/output devices
- **User-level apps**: ignorance is bliss
  - Unaware of most nasty hardware details
  - Unaware of other apps (and OS)
  - Explicitly denied access to resource allocation features

System Calls

- Controlled transfers to/from OS
- **System Call**: a user-level app "function call" to OS
  - Leave description of what you want done in registers
  - SYSCALL instruction (also called TRAP or INT)
    - Can't allow user-level apps to invoke arbitrary OS code
    - Restricted set of legal OS addresses to jump to (**trap vector**)
  - Processor jumps to OS using trap vector
    - Sets privileged mode
  - OS performs operation
  - OS does a "return from system call"
    - Unsets privileged mode

Interrupts

- **Exceptions**: synchronous, generated by running app
  - E.g., illegal insn, /0, etc.
- **Interrupts**: asynchronous events generated externally
  - E.g., timer, I/O request/reply, etc.
- **"Interrupt" handling**: same mechanism for both
  - "Interrupts" are on-chip signals/bits
    - Either internal (e.g., timer, exceptions) or connected to pins
  - Processor continuously monitors interrupt status, when one is high…
  - Hardware jumps to some preset address in OS code (interrupt vector)
  - Like an asynchronous, non-programmatic SYSCALL
- **Timer**: programmable on-chip interrupt
  - Initialize with some number of micro-seconds
  - Timer counts down and interrupts when reaches 0

LC3/P37X OS Support

- Privilege modes
  - Two: user/kernel, encoded in `PSR[15]`
- Separate address spaces
  - Low 12K (x0000-x2FFF), high 12K (xC000-xFFFF) reserved for OS
  - 16-bit Memory Protection Register (`MPR`) enforces
- Call gates
  - TRAP (256 codes) and RTT
- Interrupts
  - Supported in ISA (256 codes), but not by simulator

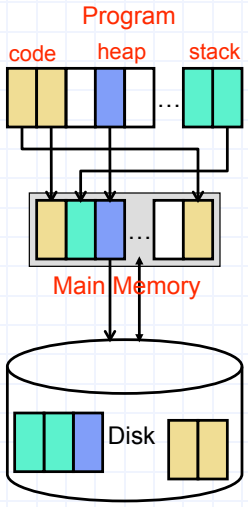
Virtualizing Processors

- How do multiple apps (and OS) share the processors?
  - **Goal: applications think there are an infinite # of processors**
- Solution: time-share the resource
  - Trigger a **context switch** at a regular interval (~1ms)
    - **Pre-emptive**: app doesn't yield CPU, OS forcibly takes it
      - + Stops greedy apps from starving others
  - **Architected state**: PC, registers
    - Save and restore them on context switches
    - Memory state?
  - **Non-architected state**: caches, branch predictor tables, etc.
    - Ignore or flush
- Operating responsible to handle context switching
  - Hardware support is just a timer interrupt

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
  - **Goal: each application thinks it has infinite memory**
- One app may want more memory than is in the system
  - App's insn/data footprint may be larger than main memory
  - Requires main memory to act like a cache (with disk as backup)
  - No notion of "program not fitting" in registers or caches
    - Why?
- Solution:
  - Part #1: treat memory as a "cache"
    - Store the overflowed blocks in "swap" space on disk
  - Part #2: add a level of indirection (address translation)

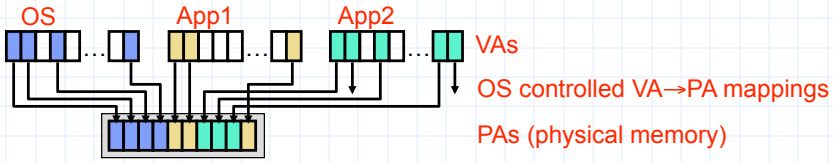
Virtual Memory (VM)



- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Alpha is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (typically M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

Virtual Memory (VM)

- **Virtual Memory (VM)**:
  - Level of indirection (like register renaming)
  - Application generated addresses are **virtual addresses (VAs)**
  - Memory accessed using **physical addresses (PAs)**
  - VAs translated to PAs at some coarse granularity
  - OS controls VA to PA mapping for itself and all other processes
  - Logically: translation performed before every insn fetch, load, store
  - Physically: hardware acceleration removes translation overhead



VM is an Old Idea: Older than Caches

- Original motivation: **single-program compatibility**
  - IBM System 370: a family of computers with one software suite
  - \+ Same program could run on machines with different memory sizes
  - – Prior, programmers explicitly accounted for memory size

- But also: **full-associativity + software replacement**
  - Memory t_{miss} is high: extremely important to reduce $\%_{miss}$

Parameter	I$/D$	L2	Main Memory
t_{hit}	2ns	10ns	**30ns**
t_{miss}	**10ns**	**30ns**	**10ms (10M ns)**
Capacity	8–64KB	128KB–2MB	**64MB–64GB**
Block size	16–32B	32–256B	**4+KB**
Assoc./Repl.	1–4, NMRU	4–16, NMRU	**Full, "working set"**

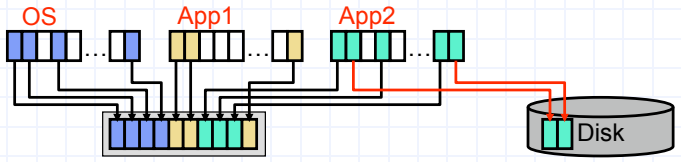
Uses of Virtual Memory

- More recently: **isolation** and **multi-programming**
  - Each app thinks it has 2^N B of memory, its stack starts 0xFFFFFFFF,…
  - Apps prevented from reading/writing each other's memory

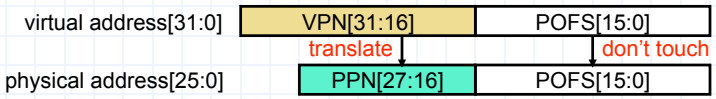
- Also: **protection**, **inter-process communication**, etc.

VM: The Basics

- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit, Alpha: 64-bit)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, especially if N=64
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP, unmapped VPs live on disk (swap)



Address Translation



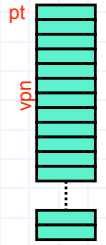
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Address Translation Mechanics I

- How are addresses translated?
  - In software (for now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
  - **Software data structure constructed by OS**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (pt[vpn].is_valid)
     return pt[vpn].ppn;
}
```



Page Table Size

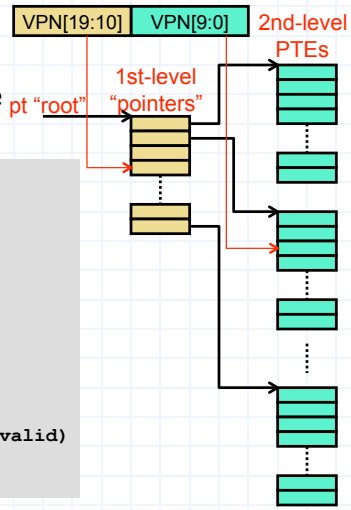
- How big is a page table on the following machine?
  - 32-bit machine
  - 4B page table entries (PTEs)
  - 4KB pages
  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB
- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get big
  - There are ways of making them smaller

Multi-Level Page Table

- One way: **multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels
- Example: two-level page table for machine on last slide
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs → 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) → 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers * 32-bit VA → 4KB → 1 upper level page

Multi-Level Page Table (PT)

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table

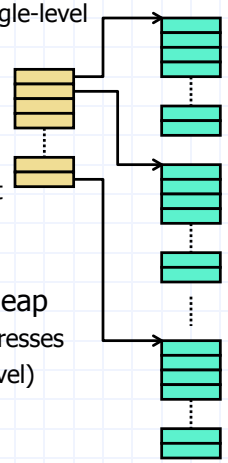



```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```


Multi-Level Page Table (PT)

- Have we saved any space?
  - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



Another Use of VM: Page-Level Protection

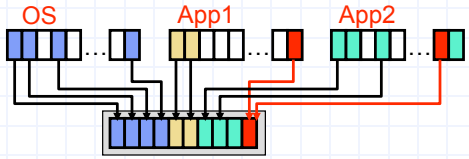
- **Page-level protection**
  - Piggy-back page-table mechanism
  - Map VPN to PPN + Read/Write/Execute permission bits
  - Attempt to execute data, to write read-only data?
    - Exception → OS terminates program
  - Useful (for OS itself actually)

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn, int action) {
   if (pt[vpn].is_valid && !(pt[vpn].permissions & action)) kill;
   …
}
```


Another: Inter-Process Communication

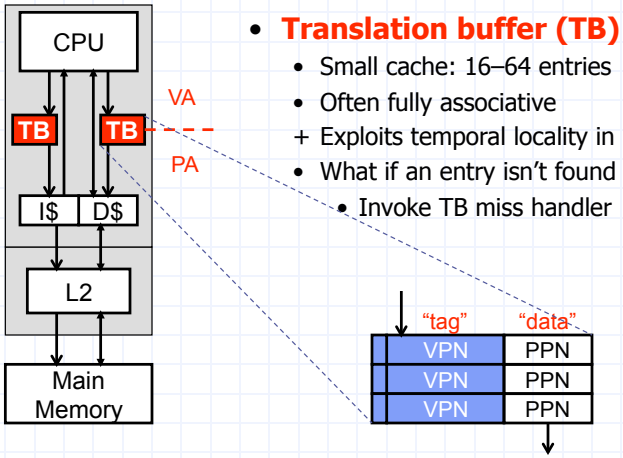
- **Inter-process communication (through memory)**
  - OS maps (different) VPNs from multiple processes to one PPN
  - Or share files via the UNIX `mmap()` call



Address Translation Mechanics II

- Conceptually
  - Translate VAs before every cache access
  - Walk the page table before every load/store/insn-fetch
  - – Would be terribly inefficient (even in hardware)
- In reality
  - **Translation Buffer (TB)**: cache translations
  - Only walk page table on TB miss
- Hardware truisms
  - Functionality problem? Add indirection (e.g., VM)
  - Performance problem? Add cache (e.g., Translation Buffer)

Translation Buffer



- **Translation buffer (TB)**
  - Small cache: 16–64 entries
  - Often fully associative
  - + Exploits temporal locality in page table
  - What if an entry isn't found in the TB?
    - Invoke TB miss handler

TB Misses

- **TB miss:** requested translation not in TB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Software-managed TB**: e.g., Alpha
  - Short (~10 insn) OS routine walks page table, updates TB
  - + Keeps PT format flexible
  - – Latency: one or two memory accesses + OS call (pipeline flush)
- **Hardware-managed TB**: e.g., x86
  - Page table root pointer in hardware register, FSM "walks" table
  - + Latency: saves cost of OS call (pipeline flush)
  - – Page table format is hard-coded

Page Faults

- **Page fault**: PTE not in TB or page table
  - → page not in memory
  - Starts out as a TB miss, detected by OS handler/hardware FSM
- **OS software routine**: no hardware FSM for this
  - Choose a physical page to replace
    - **"Working set"**: refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Requires another data structure: **frame map** (why?)

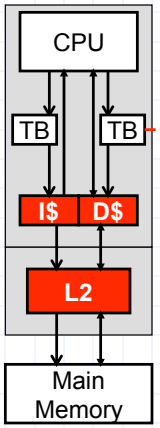
```
struct {
   int process, vpn;
} FME;
struct FME fm[NUM_PHYSICAL_PAGES];
```

Full-Associativity + Software Replacement

- VM treats main memory as a fully associative cache...
- ...with a sophisticated replacement policy
  - Why is this OK for main memory and not OK for caches?

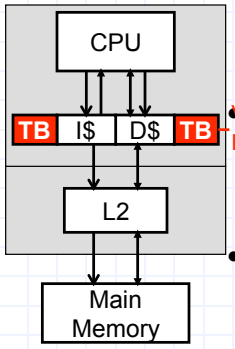
- Two aspects to full associativity
  - **Replacement**: the aspect of full-associativity you want
    - Algorithm only has to be as fast as t_{miss}
      - Memory: can take as long as you want to decide
      - D$: can take up to t_{hit-L2} to decide if you want
        - Why don't you? With 2-way associativity, will it matter?
  - **Lookup**: the aspect you don't want, but get anyway
    - Memory: lookup made fast by level of indirection (PT and TB)
      - Doesn't require searching all entries
    - D$: no indirection → slow lookup → no full-associativity

Serial TB & Cache Access



- **"Physical" caches**
  - Indexed and tagged by **physical addresses**
  - \+ Natural, "lazy" sharing of caches between apps/OS
    - VM ensures isolation (via **physical addresses**)
    - No need to do anything on context switches
    - Multi-threading works too
  - \+ Cached inter-process communication works
    - Single copy indexed by physical address
  - – Slow: adds at least one cycle to t_{hit}
- Note: **TBs are by definition virtual**
  - Indexed and tagged by virtual addresses
  - Flush across context switches
  - Or extend with process id tags

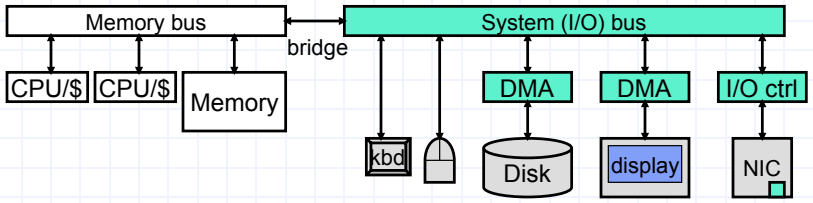
Parallel TB & Cache Access



- What about parallel access?
  - What if
    (cache size) / (associativity) ≤ page size
  - Index bits same in virt. and physical addresses!
- Access TB in parallel with cache
  - Cache access needs tag only at very end
  - \+ Fast: no additional t_{hit} cycles
  - \+ Still no context-switching/aliasing problems
  - Dominant organization used today
- Example: Pentium 4, 4KB pages, 8KB, 2-way SA L1 data cache
  - Implication: associativity allows bigger caches

A Computer System: I/O Subsystem

- I/O subsystem: kind of boring, kind of important
  - **I/O devices**: storage, input, display, network, …
  - **I/O bus**
- Software:
  - Virtualized by OS
    - Device drivers
  - Presents synchronous interface for asynchronous devices

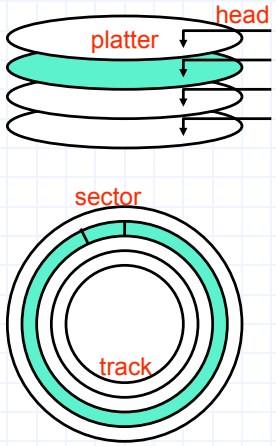


I/O Devices

- Primary characteristic: **data rate (bandwidth)**
  - Latency really only an issue for disk (and network)
  - Contributing factors: **input-output-both? partner?**
  - "Interesting" devices have high data rates

Device	Partner	I/O	Data Rate (KB/s)
Keyboard	Human	I	2 B/key * 10 key/s = 0.02
Mouse	Human	I	2 B/sample * 10 sample/s = 0.02
0.5Mp DVD recorder	Human	I	4 B/pixel * 0.5M pixel/disp * 60 disp/s = 120,000.00
Speaker	Human	O	0.60
Printer	Human	O	200.00
SVGA display	Human	O	4 B/pixel * 1M pixel/disp * 60 disp/s = 240,000.00
Ethernet card	Machine	I/O	10,000.00
Disk	Machine	I/O	10,000.00–100,000.00

An Important I/O Device: Disk



- **Disk**: like stack of record players
- Collection of **platters**
  - Each with read/write head
- Platters divided into concentric **tracks**
  - Head seeks to track
  - All heads move in unison
- Each track divided into **sectors**
  - More sectors on outer tracks
  - Sectors rotate under head
- **Controller**
  - Seeks heads, waits for sectors
  - Turns heads on/off
  - May have its own cache (a few MBs)
    - Exploit spatial locality

Disk Latency

- Disk read/write latency has four components
  - **Seek delay (t_{seek})**: head seeks to right track
    - Average of ~5ms - 15ms
    - Less in practice because of shorter seeks)
  - **Rotational delay ($t_{rotation}$)**: right sector rotates under head
    - On average: time to go halfway around disk
      - Based on rotation speed (RPM)
      - 10,000 to 15,000 RPMs
    - ~3ms
  - **Transfer time ($t_{transfer}$)**: data actually being transferred
    - Fast for small blocks
  - **Controller delay ($t_{controller}$)**: controller overhead (on either side)
    - Fast (no moving parts)
- **$t_{disk} = t_{seek} + t_{rotation} + t_{transfer} + t_{controller}$**

Disk Latency Example

- Example: time to read a 4KB chunk assuming…
  - 128 sectors/track, 512 B/sector, 6000 RPM, 10 ms t_{seek}, 1 ms $t_{controller}$
  - 6000 RPM → 100 R/s → 10 ms/R → $t_{rotation}$ = 10 ms / 2 = 5 ms
  - 4 KB page → 8 sectors → $t_{transfer}$ = 10 ms * 8/128 = 0.6 ms
  - $t_{disk} = t_{seek} + t_{rotation} + t_{transfer} + t_{controller}$ = 16.6 ms
  - t_{disk} = 10 + 5 + 0.6 + 1 = 16.6 ms

Disk Bandwidth: Sequential vs Random

- Disk is bandwidth-inefficient for page-sized transfers
  - Sequential vs random accesses
- **Random accesses:**
  - One read each disk access latency (~10ms)
  - Randomly reading 4KB pages
    - 10ms is 0.01 seconds → 100 access per second
    - 4KB * 100 access/sec → 400KB/second bandwidth
- **Sequential accesses:**
  - Stream data from disk (no seeks)
  - 128 sectors/track, 512 B/sector, 6000 RPM
    - 64KB per rotation, 100 rotation/per sec
    - 6400KB/sec → 6.4MB/sec
- Sequential access is ~10x or more bandwidth than random
  - Still no where near the 1GB/sec to 10GB/sec of memory

Some (Old) Example Disks (Hitachi)

	Ultrastar	Travelstar	Microdrive
Diameter	3.5"	2.5"	1.0"
Capacity	300 GB	40 GB	4 GB
Cache	8 MB	2 MB	128KB
RPM	10,000 RPM	4200 RPM	3600 RPM
Seek	4.5 ms	12 ms	12 ms
Sustained Data Rate	100 MB/s	40 MB/s	10 MB/s
Interface	ATA or SCSI	ATA	ATA
Cost	$450	$120	$70
Use	Desktop	Notebook	some iPods

- **Flash**: non-volatile CMOS storage
  - The "new disk": replacing disk in many

Typical I/O Device Interface

- Operating system talks to the I/O device
  - Send commands, query status, etc.
  - Software uses special uncached load/store operations
  - Hardware sends these reads/writes across I/O bus to device
- Direct Memory Access (DMA)
  - For big transfers, the I/O device accesses the memory directly
  - Example: DMA used to transfer an entire block to/from disk
- Interrupt-driven I/O
  - The I/O device tells the software its transfer is complete
  - Tells the hardware to raise an "interrupt" (door bell)
  - Processor jumps into the OS
  - Inefficient alternative: polling

Brief History of DRAM

- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - More closely resembles today's disks than memory
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

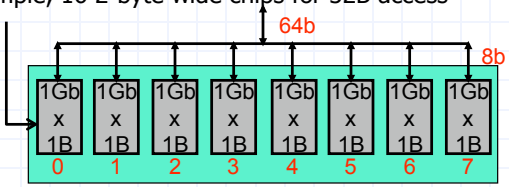
Memoirs of a Computer Programmer, 1985

DRAM (Dynamic RAM) Technology

- DRAM optimized for density (bits per chip)
  - Capacitor & **one transistor**
    - In contrast, SRAM has six transistors
  - Capacitor stores charge (for 1) or no charge (for 0)
  - Transistor controls access to the charge
  - Analogy of balloon + valve
- Destructive read
  - Sensing if the capacitor is charged or not destroy value
  - Solution: every read is immediately followed by a write
- Refresh
  - Charge leaks away
  - Occasionally read then write back the values
  - Not needed for SRAM
- High latency (50ns to 150ns)

Main Memory (DRAM) Bandwidth

- Use multiple memory chips to increase bandwidth
  - Recall, access are the same size as second-level cache
  - Example, 16 2-byte wide chips for 32B access



- DRAM density increasing faster than demand
  - Result: number of memory chips per system decreasing
- Need to increase the **bandwidth per chip**
  - SDRAM ➔ DDR ➔ DDR2
  - Rambus - high-bandwidth memory
    - Used by several game consoles. Why?

Disk & Memory Bandwidth (vs Latency)

- **Latency determined by technology**
  - Mechanical disks (~5ms)
  - Slow off-chip DRAM access (~50ns)
- **Bandwidth is mostly determined by cost**
  - "You can always buy bandwidth"
  - Need more memory bandwidth (bytes/second)?
    - Buy more memory chips, use more pins to connect them to chip
    - Typical "DIMMs" have several DRAM chips
  - Need more disk I/O bandwidth (operations/second)
    - Buy more disks
  - Use the extra resources in parallel
    - Yet another example of parallelism

Designing an I/O System for Bandwidth

- Approach
  - Find bandwidths of individual components
  - Configure components you can change…
  - To match bandwidth of bottleneck component you can't
  - Caveat: real I/O systems modeled with simulation
- Example parameters
  - 300 MIPS CPU, 100 MB/s I/O bus
  - 150K insns per I/O operation, random 64 KB reads
  - Disk controllers (20 MB/s): each accommodates up to 7 disks
  - Disks with $t_{seek} + t_{rotation}$ = 10 ms, random 64 KB reads
- Determine
  - What is the maximum sustainable I/O rate?
  - How many disk controllers and disks does it require?

Designing an I/O System for Bandwidth

- First: determine I/O rates of components we can't change
  - CPU: (300M insn/s) / (150K Insns/IO) = 2000 IO/s
  - I/O bus: (200M B/s) / (64K B/IO) = 3124 IO/s
  - Peak I/O rate determined by cpu: **2000 IO/s**
- Second: configure remaining components to match rate
  - Disk: 10ms is 0.01 seconds, so each disk is 100 IO/s
  - How many disks?
    - (2000 IO/s) / (100 IO/s) = **20 disks**
  - How many controllers?
    - For 100MB/s we need **five 20MB/s controllers**
    - Four disks per controller
- How would this change for sequential reads?