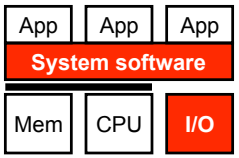
CIS 501
Computer Architecture

Unit 5: Virtual Memory & I/0

This Unit: Virtual Memory & I/O



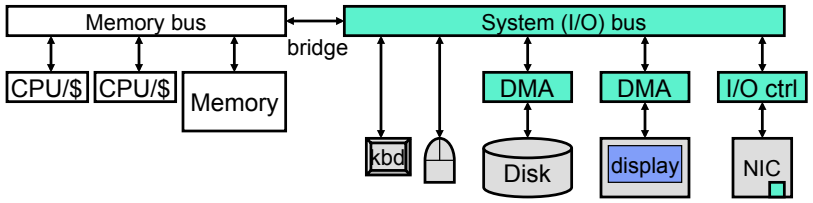
- The operating system (OS)
  - A super-application
  - Hardware support for an OS
- Virtual memory
  - Page tables and address translation
  - TLBs and memory hierarchy issues
- I/O
  - Disks
  - Flash memory

Readings

- P+H
  - Virtual Memory: C.4, C.5 starting with page C-53 (Opteron), C.6
  - Disks: 6.1, 6.2, 6.7-6.10

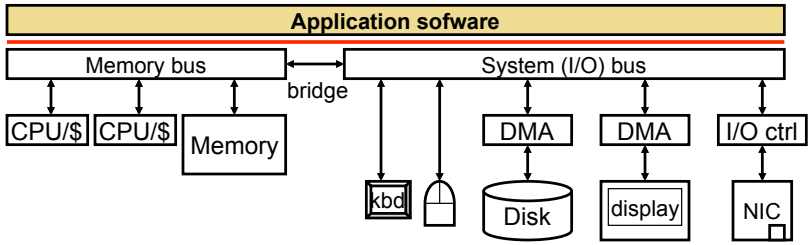
A Computer System: Hardware

- 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)



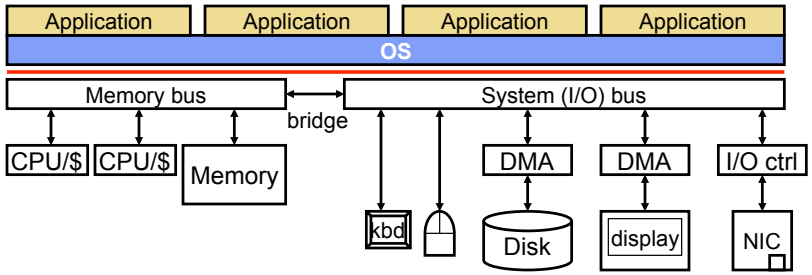
A Computer System: + App Software

- **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 (device driver software)
- **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, divide by zero, 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

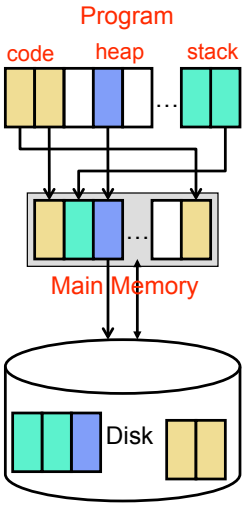
Virtualizing Processors

- 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 next level in memory hierarchy (slow)
    - Write-back, write-allocate, large blocks or "pages"
  - 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)

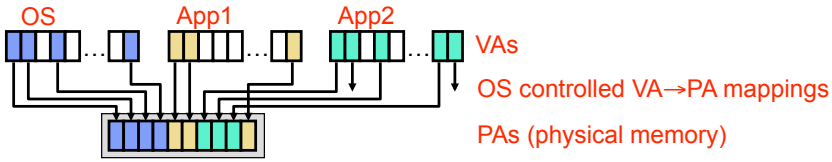
Virtual Memory (VM)



- 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)**
    - Each process ***thinks*** it has its own 2^N bytes of address space
  - 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

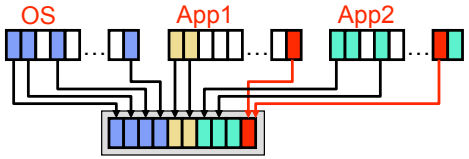
- Original motivation: **single-program compatibility**
  - IBM System 370: a family of computers with one software suite
  - \+ Same program could run on machines with different memory sizes
  - – Prior, programmers explicitly accounted for memory size

- 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"**

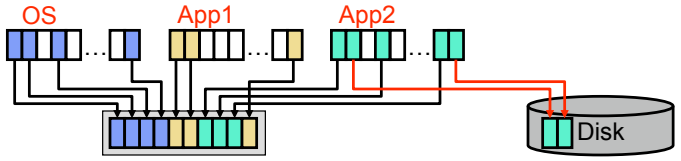
Uses of Virtual Memory

- 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
    - Can't even address the other program's memory!
- **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware
- **Inter-process communication**.
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX `mmap()` call

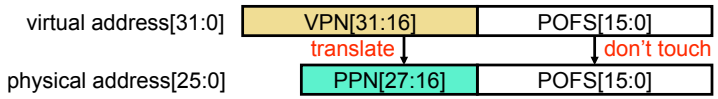


Virtual Memory: The Basics

- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-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) or nowhere (if not yet touched)



Address Translation



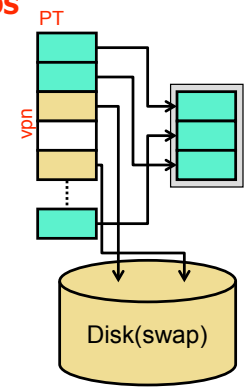
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine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

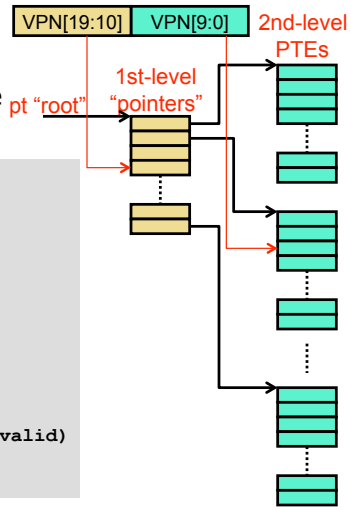
- How big is a page table on the following machine?
  - 32-bit machine
  - 4B page table entries (PTEs)
  - 4KB pages
  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB
- How big would the page table be with 64KB pages?
- How 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 (PT)

- 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

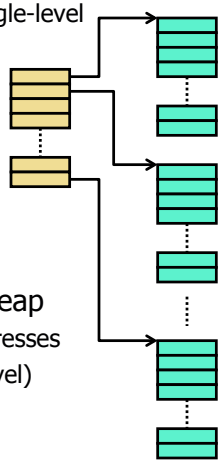



```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

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)



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)

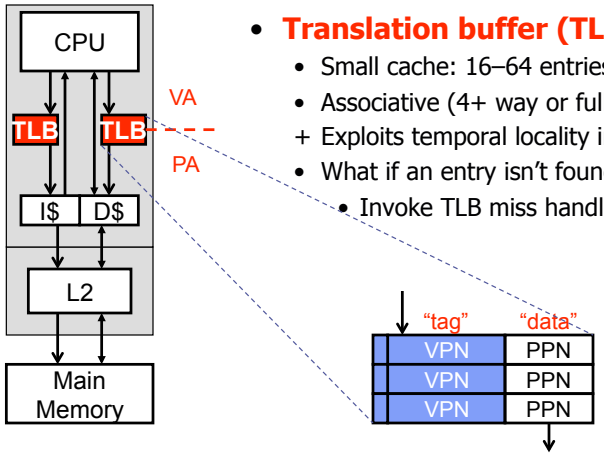
```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

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


Address Translation Mechanics II

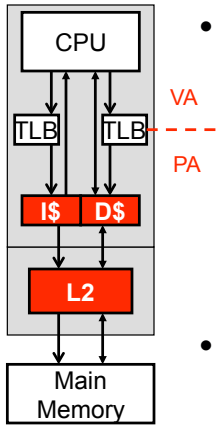
- Conceptually
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
  - – Would be terribly inefficient (even in hardware)
- In reality
  - **Translation Lookaside Buffer (TLB)**: cache translations
  - Only walk page table on TLB miss
- Hardware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

Translation Buffer



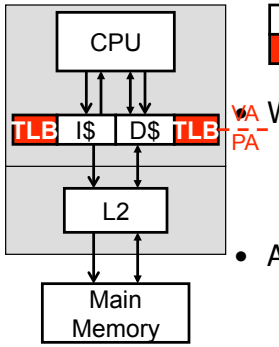
- **Translation buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  + Exploits temporal locality in page table
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler

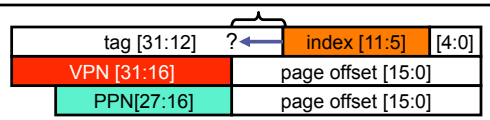
Serial TLB & Cache Access



- **"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: **TLBs are by definition virtual**
  - Indexed and tagged by virtual addresses
  - Flush across context switches
  - Or extend with process id tags

Parallel TLB & Cache Access





- What about parallel access?
  - What if
    (cache size) / (associativity) ≤ page size
  - Index bits same in virt. and physical addresses!
- Access TLB in parallel with cache
  - Cache access needs tag only at very end
  + Fast: no additional t_{hit} cycles
  + 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

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
  - **Like caches**: there can be L2 TLBs
- Example: AMD Opteron
  - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table
- **Rule of thumb**: TLB should "cover" L2 contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Think about relative miss latency in each…

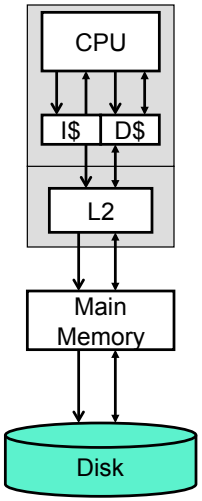
TLB Misses

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Software-managed TLB**: e.g., Alpha
  - Short (~10 insn) OS routine walks page table, updates TLB
  - \+ Keeps page table format flexible
  - – Latency: one or two memory accesses + OS call (pipeline flush)
- **Hardware-managed TLB**: 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

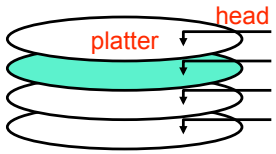
- **Page fault**: PTE not in TLB or page table
  - → page not in memory
  - Starts out as a TLB miss, detected by OS/hardware handler
- **OS software routine**:
  - Choose a physical page to replace
    - **"Working set"**: refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Requires yet another data structure: **frame map** (why?)
  - Treat like a normal TLB miss from here

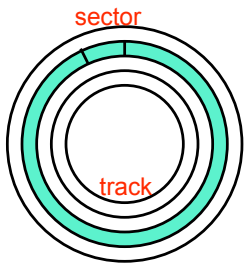
One Instance of I/O



- Have briefly seen one instance of I/O
  - **Disk**: bottom of memory hierarchy

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
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

Increasing Disk Performance

- **Software can help**
  - More sequential seeks (layout files on disk intelligently)
- **Raw latency/bandwidth determined by technology**
  - Mechanical disks (~5ms seek time), rotation time
  - Higher RPMs help bandwidth (but not really latency)
- **Bandwidth is mostly determined by cost**
  - "You can always buy bandwidth"
  - **Disk arrays**: buy more disks, **stripe data across multiple disks**
    - Increases both sequential and random access bandwidth
  - Use the extra resources in parallel
    - Yet another example of parallelism

Aside: Storage Backup

- **Data is more valuable than hardware!**
  - Almost always true
- **Protecting data - three aspects**
  - **User error** - accidental deletion
    - Aside: ".snapshot" on enaic/halfdome filesystem
  - **Disk failure** - mechanical, wears out over time
  - **Disaster recovery** - An entire site is disabled/destroyed
- **Approaches:**
  - Frequent tape backups, taken off site (most common today)
  - Handle each problem distinctly
    - File system, redundant disks (next), network-based remote backup

What If a Disk Fails? (Disk Reliability)

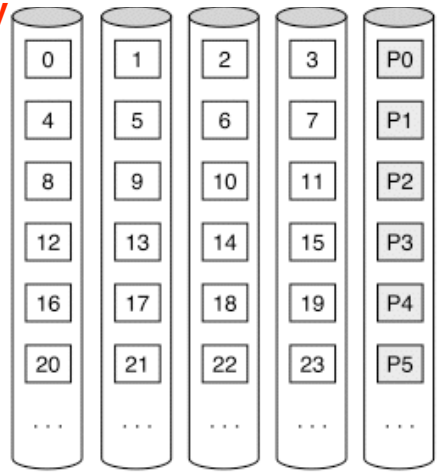
- **Error correction**: important for disk, too
  - Error correction/detection per block (handled by disk hardware)
  - Mechanical disk failures (entire disk lost) most common failure mode
    - Many disks means high failure rates
    - Entire file system can be lost if files striped across multiple disks
- **RAID (redundant array of inexpensive disks)**
  - Add redundancy
  - Similar to DRAM error correction, but…
  - Major difference: which disk failed is known
    - Even parity can be used to recover from single failures
    - Parity disk can be used to reconstruct data faulty disk
  - RAID design balances bandwidth and fault-tolerance
  - Implemented in hardware (fast, expensive) or software

Levels of RAID - Summary

- **RAID-0 - no redundancy**
  - Multiplies read and write bandwidth
- **RAID-1 - mirroring**
  - Pair disks together (write both, read one)
  - 2x storage overhead
  - Multiples only read bandwidth (not write bandwidth)
- **RAID-3 - bit-level parity** (dedicated parity disk)
  - N+1 disks, calculate parity (write all, read all)
  - Good sequential read/write bandwidth, poor random accesses
  - If N=8, only 13% overhead
- **RAID-4/5 - block-level parity**
  - Reads only data you need
  - Writes require read, calculate parity, write data&parity

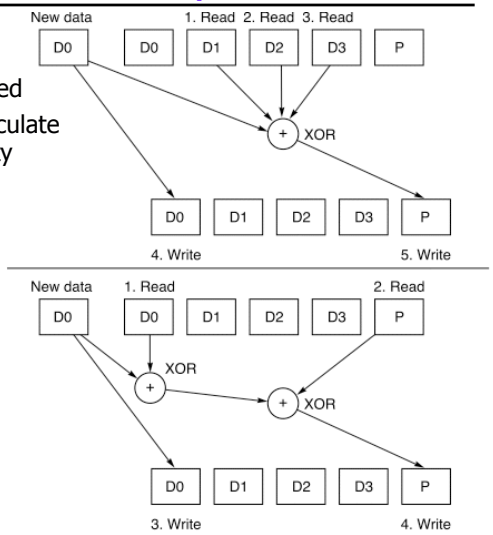
RAID-3: Bit-level parity

- **RAID-3 - bit-level parity**
  - dedicated parity disk
  - N+1 disks, calculate parity (write all, read all)
  - Good sequential read/write bandwidth, poor random accesses
  - If N=8, only 13% overhead



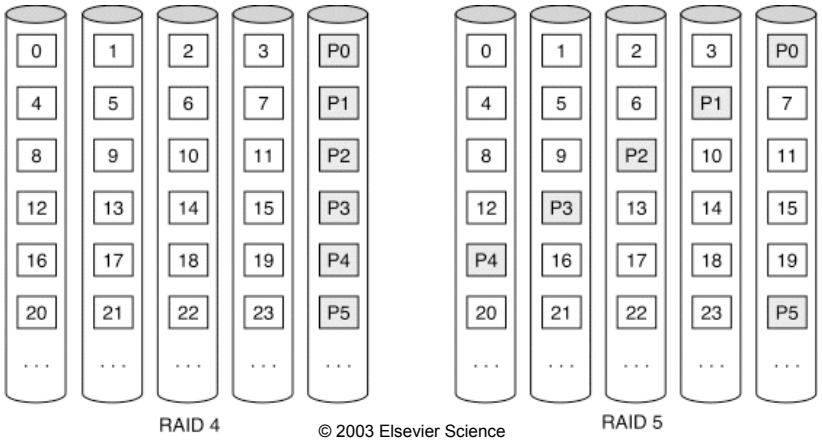
RAID 4/5 - Block-level Parity

- **RAID-4/5**
  - Reads only data you need
  - Writes require read, calculate parity, write data&parity
- Naïve approach
  1. Read all disks
  2. Calculate parity
  3. Write data&parity
- Better approach
  - Read data&parity
  - Calculate parity
  - Write data&parity
- Still worse for **writes** than RAID-3



RAID-4 vs RAID-5

- RAID-5 rotates the parity disk, avoid single-disk bottleneck



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, 150K insns per I/O operation, random reads
  - I/O bus: 200 MB/s Disk controllers: 30 MB/s, up to 7 disks each
  - Workload has multiple tasks to keep CPUs busy during I/O
- Determine
  - What is the access time for the random access of a specific size?
  - What is the maximum sustainable I/O rate for random accesses?
  - How many disk controllers and disks does it require?

Disk Access Time Calculation

- Time to read a random chunk assuming…
  - 1024 sectors/track, 512 B/sector, 12000 RPM
  - t_{seek} is 6.5 ms, $t_{controller}$ is 0.375 ms
- First, calculate $t_{rotation}$
  - 12000 RPM → 200 R/s → 5 ms/R → $t_{rotation}$ = 5 ms / 2 = 2.5 ms
- Next, calculate $t_{transfer}$
  - 4KB read → 8 sectors → $t_{transfer}$ = 5 ms * 8/1024 = 0.039 ms
  - 64KB read → 128 sectors → $t_{transfer}$ = 5 ms * 128/1024 = 0.625 ms
  - 256KB read → 512 sectors → $t_{transfer}$ = 5 ms * 512/1024 = 2.5 ms
- Finally, $t_{disk} = t_{seek} + t_{rotation} + t_{transfer} + t_{controller}$
  - t_{disk} for 4KB = 6.5 + 2.5 + **0.039** + 0.375 = 9.4 ms
  - t_{disk} for 64KB = 6.5 + 2.5 + **0.625** + 0.375 = 10 ms
  - t_{disk} for 256KB = 6.5 + 2.5 + **2.5** + 0.375 = 11.9 ms

Calculation for 4KB Random Reads

- First: determine I/O rates of components we can't change
  - CPU: (300M insn/s) / (150K Insns/IO) = 2000 IO/s
  - I/O bus: (200MB/s) / (**4K** B/IO) = 51200 IO/s
  - Peak I/O rate determined by cpu: **2000 IO/s**
    - 2000 IO/s * 4KB per IO = **7.8MB/s**
- Second: configure remaining components to match rate
  - Disk: **9.4** ms is 0.0094 sec, so each disk is 106 IO/s & 0.41MB/sec
  - How many disks?
    - (2000 IO/s) / (106 IO/s) = **19 disks**
  - How many controllers?
    - At 0.4 MB/sec per disk, 75 disks max for each 30MB/s controller
    - But each controller can only support 7 disks…
    - Thus, for 19 disks, we need **3 controllers**

Calculation for 64KB Random Reads

- First: determine I/O rates of components we can't change
  - CPU: (300M insn/s) / (150K Insns/IO) = 2000 IO/s
  - I/O bus: (200MB/s) / (**64K** B/IO) = 3200 IO/s
  - Peak I/O rate determined by cpu: **2000 IO/s**
    - 2000 IO/s * 64KB per IO = **128MB/s**
- Second: configure remaining components to match rate
  - Disk: **10** ms is 0.01 seconds, so each disk is 100 IO/s & 6.4MB/sec
  - How many disks?
    - (2000 IO/s) / (100 IO/s) = **20 disks**
  - How many controllers?
    - At 6.4 MB/sec per disk, 4 disks max for each 30MB/s controller
    - For 20 disks, we need **5 controllers**

Calculation for 256KB Random Reads

- First: determine I/O rates of components we can't change
  - CPU: (300M insn/s) / (150K Insns/IO) = 2000 IO/s
  - I/O bus: (200MB/s) / (**256K** B/IO) = 800 IO/s
  - Peak I/O rate determined by I/O bus: **800 IO/s**
    - 800 IO/s * 256KB per IO = **200MB/s**

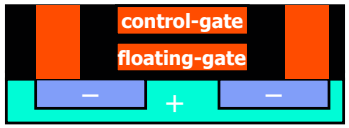
- Second: configure remaining components to match rate
  - Disk: **11.9** ms is 0.0119 sec, so each disk is 84 IO/s & 21MB/sec
  - How many disks?
    - (800 IO/s) / (84 IO/s) = **10 disks**
  - How many controllers?
    - At 21 MB/sec per disk, 1 disk max for each 30MB/s controller
    - For 10 disks, we need **10 controllers**

Designing an I/O System for Latency

- Previous system designed for bandwidth
- Some systems have latency requirements as well
  - E.g., database system may require maximum or average latency

- Latencies are actually harder to deal with than bandwidths
  - **Unloaded system**: few concurrent IO transactions
    - Latency is easy to calculate
  - **Loaded system**: many concurrent IO transactions
    - Contention can lead to queuing
    - Latencies can rise dramatically
    - Queuing theory can help if transactions obey fixed distribution
    - Otherwise simulation is needed

Flash Storage Technology

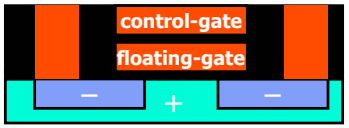


- **Floating-gate transistor**
  - CMOS transistor with an additional (floating) gate
  - 1/0 interpreted by whether transistor conducts or not

- How does this work?
  - F-gate completely encased in SiO_2 → no leakage → **non-volatile**
  - F-gate uncharged → c-gate can open/close channel → this is '1'
  - F-gate charged → cancels out c-gate → this is '0'

- Read latency is ~100ns (similar to DRAM, a little slower)

Flash Storage Technology cont'd



- How does f-gate charge and discharge?
  - By **F-N quantum tunneling** (aka "hot electron injection")
  - – Charging (writing a 0) requires huge voltage → slow
    - Write latency is ~30 μs (300X slower than reads)
  - – Discharging (re-writing a 1) requires huger voltage → very slow
    - Erase latency is ~500 ms (slower than disk)
  - – Higher voltage operations degrade Flash substrate
    - **Write fatigue**: a single cell can only be written about 1M times

- Slow-writes + fatigue → more programmable ROM than RAM
  - EEPROM: electrically-erasable programmable ROM

When To Use Flash vs. Disk?

- Cost (numbers somewhat out of date)
  - Flash: low-fixed + high-marginal ~ 10$ + 10$/GB
  - Disk: high-fixed + low-marginal: ~ 50$ + 1$/GB
  - Flash is cheaper for 4GB or less
- Form factor
  - For 4GB or less, Flash is also much smaller
- Environmental robustness
  - Flash doesn't have moving parts, so better fit for removable media
- Of course flash has write fatigue
- Flash is good for
  - Removable storage: digital cameras, memory sticks
  - Low-write file systems: cell phones, music players
  - Not large heavy-write file systems: desktop/laptop (may change)

Summary

- OS virtualizes memory and I/O devices
- Virtual memory
  - "infinite" memory, isolation, protection, inter-process communication
  - Page tables
  - Translation buffers
    - Parallel vs serial access, interaction with caching
  - Page faults
- I/O
  - Disk latency and bandwidth
  - Disk arrays & RAID
  - Flash memory