Memory and I/O buses

- CPU accesses physical memory over a bus
- Devices access memory over I/O bus with DMA
- Devices can appear to be a region of memory

Realistic ~2005 PC architecture

Communicating with a device

- Memory-mapped device registers
  - Certain physical addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory
- Device memory – device may have memory OS can write to directly on other side of I/O bus
- Special I/O instructions
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports at byte granularity
- DMA – place instructions to card in main memory
  - Typically then need to “poke” card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus

What is memory?

- SRAM – Static RAM
  - Like two NOT gates circularly wired input-to-output
  - 4-6 transistors per bit, actively holds its value
  - Very fast, used to cache slower memory
- DRAM – Dynamic RAM
  - A capacitor + gate, holds charge to indicate bit value
  - 1 transistor per bit – extremely dense storage
  - Charge leaks – need slow comparator to decide if bit 1 or 0
  - Must re-write charge after reading, and periodically refresh
- VRAM – “Video RAM”
  - Dual ported DRAM, can write while someone else reads

What is I/O bus? E.g., PCI

- Memory-mapped device registers
  - Certain physical addresses correspond to device registers
  - Load/store gets status/sends instructions – not real memory
- Device memory – device may have memory OS can write to directly on other side of I/O bus
- Special I/O instructions
  - Some CPUs (e.g., x86) have special I/O instructions
  - Like load & store, but asserts special I/O pin on CPU
  - OS can allow user-mode access to I/O ports at byte granularity
- DMA – place instructions to card in main memory
  - Typically then need to “poke” card by writing to register
  - Overlaps unrelated computation with moving data over (typically slower than memory) I/O bus
**x86 I/O instructions**

```c
static inline uint8_t
l sb (uint10_t port)
{
  static inline void
  outb (uint16_t port, uint8_t data) {
  }
  data;
  asm volatile ("im b %w1, %b0" : "=a" (data) : "&d" (port));
  return data;
}

static inline void
outb (uint16_t port, uint8_t_t data) {
  asm volatile ("out b %b0, %w1" : : "a" (data), "&d" (port));
}

static inline void
insv (uint16_t port, void *addr, size_t cnt) {
  asm volatile ("re p insv" : : "a" (addr), "c" (cnt)
        : "&d" (port), "memory");
}
```

---

**Example: parallel port (LPT1)**

- **Simple hardware has three control registers:**
  ```
  | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
  |-------------------------|-------------------------|
  | BSY | ACK | PAP | OFON | ERR | -  | -  | -  |
  |-------------------------|-------------------------|
  | -  | -   | -   | -    | -   | -  | -  | -  |
  ``

- **Every bit except IRQ corresponds to a pin on 25-pin connector:**
  ![Port Connectors Image](imagecredits:Wikipedia)

---

**Writing bit to parallel port [osdev]**

```c
void
 sendbyte (uint8_t byte) {
  /* Wait until BSY bit is 1. */
  while ((inb (0x378) & 0x80) == 0)
    delay (1);
  /* Put the byte we wish to send on pins D7-0. */
  outb (0x378, byte);
  /* Pulse STR (strobe) line to inform the printer */
  if (a byte is available */
    \* uint8_t ctrlval = inb (0x37A);
    outb (0x37A, ctrlval | 0x01);
    delay (1);
    outb (0x37A, ctrlval);
}
```

---

**IDE disk driver**

```c
void IDE_ReadSector (int disk, int off, void *buf) {
  outb (0x1F6, disk << 7 | 0x80); // Select Drive
  IDEWait();
  outb (0x1F8, 1); // Read length (1 sector = 512 B)
  outb (0x1F9, off); // IBA low
  outb (0x1F4, off >> 8); // IBA mid
  outb (0x1F6, off >> 16); // IBA high
  outb (0x1F7, 0x200); // Read command
  insv (0x1FC, buf, 266); // Read 266 words
}

void IDEWait () {
  // Discard status 4 times
  inb (0x1F7); inb (0x1F7);
  inb (0x1F7); inb (0x1F7);
  // Wait for status BUSY flag to clear
  while (((inb (0x1F7) & 0x80) != 0)
```

---

**Memory-mapped IO**

- **Instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows 2<sup>nd</sup> different port numbers
  - Per-port access control turns out not to be useful
    (any port access allows you to disable all interrupts)
- **Devices can achieve same effect with physical addresses, e.g.:**
  ```c
  volatile int32_t *device_control
      = (int32_t *) (0xc0100 + PHYS_BASE);
  *device_control = 0x80;
  int32_t status = *device_control;
  ```
- **Assign physical addresses at boot to avoid conflicts. PCI:**
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device
  - OS must map physical to virtual addresses, ensure non-cachable
  - Idea: only use CPU to transfer control requests, not data
  - Include list of buffer locations in main memory
  ```c
  - Device reads list and accesses buffers through DMA
    - Descriptions sometimes allow for scatter/gather I/O
  ```

---

**DMA buffers**

```c
<table>
<thead>
<tr>
<th>Type</th>
<th>Memory buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>101</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>102</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>103</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>....</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>104</td>
<td>Memory buffers</td>
</tr>
</tbody>
</table>
```

---

**IDE disk driver**

```c
void IDE_ReadSector (int disk, int off, void *buf) {
  outb (0x1F6, disk << 7 | 0x80); // Select Drive
  IDEWait();
  outb (0x1F8, 1); // Read length (1 sector = 512 B)
  outb (0x1F9, off); // IBA low
  outb (0x1F4, off >> 8); // IBA mid
  outb (0x1F6, off >> 16); // IBA high
  outb (0x1F7, 0x200); // Read command
  insv (0x1FC, buf, 266); // Read 266 words
}
```

---

**Memory-mapped IO**

- **Instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows 2<sup>nd</sup> different port numbers
  - Per-port access control turns out not to be useful
    (any port access allows you to disable all interrupts)
- **Devices can achieve same effect with physical addresses, e.g.:**
  ```c
  volatile int32_t *device_control
      = (int32_t *) (0xc0100 + PHYS_BASE);
  *device_control = 0x80;
  int32_t status = *device_control;
  ```
- **Assign physical addresses at boot to avoid conflicts. PCI:**
  - Slow/clunky way to access configuration registers on device
  - Use that to assign ranges of physical addresses to device
  - OS must map physical to virtual addresses, ensure non-cachable
  - Idea: only use CPU to transfer control requests, not data
  - Include list of buffer locations in main memory
  ```c
  - Device reads list and accesses buffers through DMA
  ```

---

**DMA buffers**

```c
<table>
<thead>
<tr>
<th>Type</th>
<th>Memory buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>101</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>102</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>103</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>....</td>
<td>Memory buffers</td>
</tr>
<tr>
<td>104</td>
<td>Memory buffers</td>
</tr>
</tbody>
</table>
```
**Example: Network Interface Card**

- Link interface talks to wire/fiber/antenna
  - Typically does framing, link-layer CRC
- FIFOs on card provide small amount of buffering
- Bus interface logic uses DMA to move packets to and from buffers in main memory

**Example: IDE disk read w. DMA**

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

**Driver architecture**

- Device driver provides several entry points to kernel
  - Reset, ioctl, output, interrupt, read, write, strategy ...
- How should driver synchronize with card?
  - E.g., Need to know when transmit buffers free or packets arrive
  - Need to know when disk request complete
- One approach: **Polling**
  - Sent a packet? Loop asking card when buffer is free
  - Waiting to receive? Keep asking card if it has packet
  - Disk I/O? Keep looping until disk ready bit set
- Disadvantages of polling?
  - Can't use CPU for anything else while polling
  - Schedule poll in future? High latency to receive packet or process disk block bad for response time

**Interrupt driven devices**

- Instead, ask card to interrupt CPU on events
  - Interrupt handler runs at high priority
  - Asks card what happened (xmit buffer free, new packet)
  - This is what most general-purpose OSES do
- Bad under high network packet arrival rate
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – receive livelock
  - Best: Adaptive switching between interrupts and polling
- Very good for disk requests
- Rest of today: Disks (network devices in 3 lectures)

**Anatomy of a disk [Ruemmler]**

- Stack of magnetic platters
  - Rotate together on a central spindle @ 3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions
- Disk arm assembly
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - One disk head per recording surface (2 × platters)
  - Sensitive to motion and vibration [Gregg](demo on youtube)
• Platters divided into concentric tracks
• A stack of tracks of fixed radius is a cylinder
• Heads record and sense data along cylinders
  - Significant fractions of encoded stream for error correction
• Generally only one head active at a time
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned

Cylinders, tracks, & sectors

- Move head to specific track and keep it there
  - Resist physical shocks, imperfect tracks, etc.
- A seek consists of up to four phases:
  - speedup–accelerate arm to max speed or half way point
  - coast–at max speed (for long seeks)
  - slowdown–stops arm near destination
  - settle–adjusts head to actual desired track
- Very short seeks dominated by settle time (~1 ms)
- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of 40g
Seek details

- Head switches comparable to short seeks
  - May also require head adjustment
  - Settles take longer for writes than for reads – Why?
- Disk keeps table of pivot motor power
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic “thermal recalibration”
  - But, e.g., ~500 ms recalibration every ~25 min bad for AV
- “Average seek time” quoted can be many things
  - Time to seek 1/3 disk, 1/3 time to seek whole disk

Disk interface

- Controls hardware, mediates access
- Computer, disk often connected by bus (e.g., ATA, SCSI, SATA)
  - Multiple devices may contend for bus
- Possible disk/interface features:
  - Disconnect from bus during requests
  - Command queuing: Give disk multiple requests
    - Disk can schedule them using rotational information
- Disk cache used for read-ahead
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can’t stop a head-switch
- Some disks support write caching
  - But data not stable—not suitable for all requests

Sectors

- Disk interface presents linear array of sectors
  - Historically 512 B, but 4 KiB in “advanced format” disks
  - Written atomically (even if there is a power failure)
- Disk maps logical sector #s to physical sectors
  - Zoning—puts more sectors on longer tracks
  - Track skewing—sector 0 pos. varies by track (sequential access speed)
  - Spaming—flawed sectors remapped elsewhere
- OS doesn’t know logical to physical sector mapping
  - Larger logical sector # difference means longer seek time
  - Highly non-linear relationship (and depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times

SCSI overview [Schmidt]

- SCSI domain consists of devices and an SDS
  - Devices: host adapters & SCSI controllers
  - Service Delivery Subsystem connects devices—e.g., SCSI bus
- SCSI-2 bus (SDS) connects up to 8 devices
  - Controllers can have > 1 “logical units” (LUNs)
  - Typically, controller built into disk and 1 LUN/target, but “bridge controllers” can manage multiple physical devices
- Each device can assume role of initiator or target
  - Traditionally, host adapter was initiator, controller target
  - Now controllers act as initiators (e.g., COPY command)
  - Typical domain has 1 initiator, ≥ 1 targets
**SCSI requests**

- A request is a command from initiator to target
  - Once transmitted, target has control of bus
  - Target may disconnect from bus and later reconnect (very important for multiple targets or even multitasking)

- Commands contain the following:
  - Task identifier—initiator ID, target ID, LUN, tag
  - Command descriptor block—e.g., read 10 blocks at pos. N
  - Optional task attribute—SIMPLE, ORDERED, HEAD OF QUEUE
  - Optional: output/input buffer, sense data
  - Status byte—GOOD, CHECK CONDITION, INTERMEDIATE, ...

**Executing SCSI commands**

- Each LUN maintains a queue of tasks
  - Each task is DORMANT, BLOCKED, ENABLED, OR ENDED
  - SIMPLE tasks are dormant until no ordered/head of queue
  - ORDERED tasks dormant until no HoQ/more recent ordered
  - HoQ tasks begin in enabled state

- Task management commands available to initiator
  - Abort/terminate task, Reset target, etc.

- Linked commands
  - Initiator can link commands, so no intervening tasks
  - E.g., could use to implement atomic read-modify-write
  - Intermediate commands return status byte INTERMEDIATE

**SCSI exceptions and errors**

- After error stop executing most SCSI commands
  - Target returns with CHECK CONDITION status
  - Initiator will eventually notice error
  - Must read specifics w. REQUEST SENSE

- Prevents unwanted commands from executing
  - E.g., initiator may not want to execute 2nd write if 1st fails

- Simplifies device implementation
  - Don’t need to remember more than one error condition

- Same mechanism used to notify of media changes
  - I.e., ejected tape, changed CD-ROM

**Disk performance**

- Placement & ordering of requests a huge issue
  - Sequential I/O much, much faster than random
  - Long seeks much slower than short ones
  - Power might fail any time, leaving inconsistent state

- Must be careful about order for crashes
  - More on this in next two lectures

- Try to achieve contiguous accesses where possible
  - E.g., make big chunks of individual files contiguous

- Try to order requests to minimize seek times
  - OS can only do this if it has a multiple requests to order
  - Requires disk I/O concurrency
  - High-performance apps try to maximize I/O concurrency

- Next: How to schedule concurrent requests

**Scheduling: FCFS**

- “First Come First Served”
  - Process disk requests in the order they are received

- Advantages

- Disadvantages

- “First Come First Served”
  - Process disk requests in the order they are received

- Advantages
  - Easy to implement
  - Good fairness

- Disadvantages
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput
**Shortest positioning time first (SPTF)**

- **Advantages**
  - Exploits locality of disk requests
  - Higher throughput

- **Disadvantages**
  - Starvation
  - Don’t always know what request will be fastest

- **Improvement:** Aged SPTF
  - Give older requests higher priority
  - Adjust “effective” seek time with weighting factor:
    \[ T_{ef} = T_{pe} - W \cdot T_{w}s \]
**“Elevator” scheduling (SCAN)**

- Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

**Advantages**
- Takes advantage of locality
- Bounded waiting

**Disadvantages**
- Cylinders in the middle get better service
- Might miss locality SPTF could exploit

CSCAN: Only sweep in one direction
Very commonly used algorithm in Unix

Also called LOOK/CLOOK in textbook
- (Textbook uses [C]SCAN to mean scan entire disk uselessly)

**CSCAN example**

- Continuum between SPTF and SCAN
  - Like SPTF, but slightly changes “effective” positioning time
    - If request in same direction as previous seek: \( T_{eфф} = T_{пр} \)
    - Otherwise: \( T_{eфф} = T_{пр} + r \cdot T_{мин} \)
  - when \( r = 0 \), get SPTF, when \( r = 1 \), get SCAN
  - E.g., \( r = 0.2 \) works well

**Advantages and disadvantages**
- Those of SPTF and SCAN, depending on how \( r \) is set

See [Worthington] for good description and evaluation of various disk scheduling algorithms

**Flash memory**

- Today, people increasingly using flash memory
  - Completely solid state (no moving parts)
    - Remembers data by storing charge
    - Lower power consumption
    - No mechanical seek times to worry about

**Limited # overwrites possible**
- Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
- Requires flash translation layer (FTL) to provide wear leveling, so repeated writes to logical block don’t wear out physical block
- FTL can seriously impact performance
- In particular, random writes very expensive [Birrell]

**Limited durability**
- Charge wears out over time
- Turn off device for a year, you can potentially lose data

**Types of flash memory**

- **NAND flash (most prevalent for storage)**
  - Higher density (most used for storage)
  - Faster erase and write
  - More errors internally, so need error correction

- **NOR flash**
  - Faster reads in smaller data units
  - Can execute code straight out of NOR flash
  - Significantly slower erases

- **Single-level cell (SLC) vs. Multi-level cell (MLC)**
  - MLC encodes multiple bits in voltage level
  - MLC slower to write than SLC
  - MLC has lower durability (bits decay faster)

**NAND Flash Overview**

- Flash device has 2112-byte pages
  - 2048 bytes of data + 64 bytes metadata & ECC
- Blocks contain 64 (SLC) or 128 (MLC) pages
- Blocks divided into 2–4 planes
  - All planes contend for same package pins
  - But can access their blocks in parallel to overlap latencies
- Can read one page at a time
  - Takes 25 \( \mu \text{sec} \) + time to get data off chip

**Must erase whole block before programming**
- Erase sets all bits to 1—very expensive (2 msec)
- Programming pre-erased block requires moving data to internal buffer, then 200 (SLC)–800 (MLC) \( \mu \text{sec} \)
<table>
<thead>
<tr>
<th>Parameter</th>
<th>SLC</th>
<th>MLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density Per Die (GB)</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Page Size (Bytes)</td>
<td>2048+32</td>
<td>2048+64</td>
</tr>
<tr>
<td>Block Size (Pages)</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>Read Latency (µs)</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write Latency (µs)</td>
<td>200</td>
<td>800</td>
</tr>
<tr>
<td>Erase Latency (µs)</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>40MHz, 16-bit bus Read b/w (MB/s)</td>
<td>75.8</td>
<td>75.8</td>
</tr>
<tr>
<td>133MHz Program b/w (MB/s)</td>
<td>20.1</td>
<td>5.0</td>
</tr>
<tr>
<td>133MHz Read b/w (MB/s)</td>
<td>126.4</td>
<td>126.4</td>
</tr>
<tr>
<td>133MHz Program b/w (MB/s)</td>
<td>20.1</td>
<td>5.0</td>
</tr>
</tbody>
</table>
File systems: traditionally hardest part of OS
- More papers on FSes than any other single topic

Main tasks of file system:
- Don’t go away (ever)
- Associate bytes with name (files)
- Associate names with each other (directories)
- Can implement file systems on disk, over network, in memory, in non-volatile ram (NVRAM), on tape, w/ paper.
- We’ll focus on disk and generalize later

Today: files, directories, and a bit of performance

Disk = First state we’ve seen that doesn’t go away
- So: Where all important state ultimately resides

Slow (milliseconds access vs. nanoseconds for memory)
- Processor speed: $\times \frac{1}{18000}$
- Disk access time: 7% / yr

Huge (100–1,000x bigger than memory)
- How to organize large collection of ad hoc information?
- File System: Hierarchical directories, Metadata, Search

Disk vs. Memory

<table>
<thead>
<tr>
<th>Disk</th>
<th>MLC NAND</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest write sector</td>
<td>sector</td>
<td>byte</td>
</tr>
<tr>
<td>Atomic write sector</td>
<td>sector</td>
<td>byte/word</td>
</tr>
<tr>
<td>Random read 8 ms</td>
<td>3-10 $\mu$s</td>
<td>50 ns</td>
</tr>
<tr>
<td>Random write 8 ms</td>
<td>9-11 $\mu$s</td>
<td>50 ns</td>
</tr>
<tr>
<td>Sequential read 100 MB/s</td>
<td>550–2500 MB/s</td>
<td>&gt; 1 GB/s</td>
</tr>
<tr>
<td>Sequential write 100 MB/s</td>
<td>520–1500 MB/s*</td>
<td>&gt; 1 GB/s</td>
</tr>
<tr>
<td>Cost $0.03/GB$</td>
<td>$0.35/GB$*</td>
<td>$6/GB$</td>
</tr>
<tr>
<td>Persistence Non-volatile</td>
<td>Non-volatile</td>
<td>Volatile</td>
</tr>
</tbody>
</table>

*Flash write performance degrades over time

Disk review

- Disk reads/writes in terms of sectors, not bytes
  - Read/write single sector or adjacent groups
- How to write a single byte? “Read-modify-write”
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk
  - Key: If cached, don’t need to read in
- Sector = unit of atomicity.
  - Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)
- Larger atomic units have to be synthesized by OS

Some useful trends

- Disk bandwidth and cost/bit improving exponentially
  - Similar to CPU speed, memory size, etc.
- Seek time and rotational delay improving very slowly
  - Why? require moving physical object (disk arm)
- Disk accesses a huge system bottleneck & getting worse
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.
- Desktop memory size increasing faster than typical workloads
  - More and more of workload fits in file cache
  - Disk traffic changes: mostly writes and new data
- Memory and CPU resources increasing
  - Use memory and CPU to make better decisions
  - Complex prefetching to support more IO patterns
  - Delay data placement decisions reduce random IO

Files: named bytes on disk

- File abstraction:
  - User’s view: named sequence of bytes
  - FS’s view: collection of disk blocks
  - File system’s job: translate name & offset to disk blocks:
    {file, offset} \( \rightarrow \) disk address

- File operations:
  - Create a file, delete a file
  - Read from file, write to file
- Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)
What’s hard about grouping blocks?

- Like page tables, file system metadata are simply data structures used to construct mappings
  - Page table: map virtual page # to physical page #
    - Page table: 23 → 33
  - File metadata: map byte offset to disk block address
    - File metadata: 512 → Unix inode → 8003121
  - Directory: map name to disk address or file #
    - Directory: foo.c → directory → 44

FS vs. VM

- In both settings, want location transparency
  - Application shouldn’t care about particular disk blocks or physical memory locations
- In some ways, FS has easier job than than VM:
  - CPU time to do FS mappings not a big deal (no TLB)
  - Page tables deal with sparse address spaces and random access, files often denser (file # / block # - 1), ~sequentially accessed
- In some ways FS’s problem is harder:
  - Each layer of translation = potential disk access
  - Space a huge premium! (But disk is huge???) Reason? Cache space never enough; amount of data you can get in one fetch never enough
  - Range very extreme: Many files <10 KB, some files many GB

Some working intuitions

- FS performance dominated by # of disk accesses
  - Say each access costs ~10 milliseconds
  - Touch the disk 100 extra times = 1 second
  - Can do a billion ALU ops in same time!
- Access cost dominated by movement, not transfer:
  - seek time + rotational delay + # bytes/disk bw
    - 1 sector: 5ms + 4ms + 5µs (≈ 512 B/(40 MB/s)) = 9ms
    - 50 sectors: 5ms + 4ms + .25ms = 9.25ms
    - Can get 50x the data for only ~3% more overhead!
- Observations that might be helpful:
  - All blocks in file tend to be used together, sequentially
  - All files in a directory tend to be used together
  - All names in a directory tend to be used together

Common addressing patterns

- Sequential:
  - File data processed in sequential order
  - By far the most common mode
  - Example: editor writes out new file, compiler reads in file, etc
- Random access:
  - Address any block in file directly without passing through predecessors
  - Examples: data set for demand paging, databases
- Keyed access
  - Search for block with particular values
  - Examples: associative data base, index
  - Usually not provided by OS

Problem: how to track file’s data

- Disk management:
  - Need to keep track of where file contents are on disk
  - Must be able to use this to map byte offset to disk block
  - Structure tracking a file’s sectors is called an index node or inode
  - Inodes must be stored on disk, too
- Things to keep in mind while designing file structure:
  - Most files are small
  - Much of the disk is allocated to large files
  - Many of the I/O operations are made to large files
  - Want good sequential and good random access (what do these require?)

Straw man: contiguous allocation

- “Extent-based”: allocate files like segmented memory
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size

- Example: IBM OS/360
  - Pros?
  - Cons? (Think of corresponding VM scheme)
Straw man: contiguous allocation

- “Extent-based”: allocate files like segmented memory
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size

Example: IBM OS/360

Pros?
- Simple, fast access, both sequential and random

Cons? (Think of corresponding VM scheme)
- External fragmentation

Straw man #2: Linked files

- Basically a linked list on disk.
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file’s first block
  - In each block, keep a pointer to the next one

Examples (sort-of): Alto, TOPS-10, DOS FAT

Pros?

Cons?

FAT discussion

- Entry size = 16 bits
  - What’s the maximum size of the FAT?
    - Given a 512 byte block, what’s the maximum size of FS?
    - One solution: go to bigger blocks. Pros? Cons?

- Space overhead of FAT is trivial:
  - 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)

- Reliability: how to protect against errors?
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability

- Bootstrapping: where is root directory?
  - Fixed location on disk:

Example: DOS FS (simplified)

- Linked files with key optimization: puts links in fixed-size “file allocation table” (FAT) rather than in the blocks.

  Directory (5)
<table>
<thead>
<tr>
<th>FAT (16-bit entries)</th>
<th>file a</th>
<th>file b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 6</td>
<td>free</td>
<td></td>
</tr>
<tr>
<td>b: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

  Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access

FAT discussion

- Entry size = 16 bits
  - What’s the maximum size of the FAT? 65,536 entries
  - Given a 512 byte block, what’s the maximum size of FS? 32 MiB
  - One solution: go to bigger blocks. Pros? Cons?

- Space overhead of FAT is trivial:
  - 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)

- Reliability: how to protect against errors?
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability

- Bootstrapping: where is root directory?
  - Fixed location on disk:
Another approach: Indexed files

- Each file has an array holding all of its block pointers
  - Just like a page table, so will have similar issues
  - Max file size fixed by array’s size (static or dynamic?)
  - Allocate array to hold file’s block pointers on file creation
  - Allocate actual blocks on demand using free list

  - Pros?
  - Cons?

Indexed files

- Issues same as in page tables
  - Large possible file size = lots of unused entries
  - Large actual size? Table needs large contiguous disk chunk

  - Solve identically: small regions with index array, this array with another array, … Downside?

Multi-level indexed files (old BSD FS)

- Solve problem of first block access slow
- inode = 14 block pointers + “stuff”

  - Pros:
    - Simple, easy to build, fast access to small files
    - Maximum file length fixed, but large.
  
  - Cons:
    - What is the worst case # of accesses?
    - What is the worst-case space overhead? (e.g., 13 block file)

  - An empirical problem:
    - Because you allocate blocks by taking them off unordered freelist, metadata and data get strewn across disk

Old BSD FS discussion

- Inodes are stored in a fixed-size array
  - Size of array fixed when disk is initialized; can’t be changed
  - Lives in known location, originally at one side of disk:

    ```
    Inode array  file blocks ...
    ```

    - Now is smeared across it (why?)

    - The index of an inode in the inode array called an i-number
    - Internally, the OS refers to files by inumber
    - When file is opened, inode brought in memory
    - Written back when modified and file closed or time elapses

More about inodes

- Inodes are stored in a fixed-size array
  - Size of array fixed when disk is initialized; can’t be changed
  - Lives in known location, originally at one side of disk:

    ```
    Inode array  file blocks ...
    ```

    - Now is smeared across it (why?)

    - The index of an inode in the inode array called an i-number
    - Internally, the OS refers to files by inumber
    - When file is opened, inode brought in memory
    - Written back when modified and file closed or time elapses

  - Size of array fixed when disk is initialized; can’t be changed
  - Lives in known location, originally at one side of disk:

    ```
    Inode array  file blocks ...
    ```

    - Now is smeared across it (why?)

    - The index of an inode in the inode array called an i-number
    - Internally, the OS refers to files by inumber
    - When file is opened, inode brought in memory
    - Written back when modified and file closed or time elapses
Directories

- Problem:
  - “Spend all day generating data, come back the next morning, want to use it.” – F. Corbato, on why files/dirs invented

- Approach 0: Users remember where on disk their files are
  - E.g., like remembering your social security or bank account #
  - Yuck. People want human digestible names
  - We use directories to map names to file blocks
  - Next: What is in a directory and why?

A short history of directories

- Approach 1: Single directory for entire system
  - Put directory at known location on disk
  - Directory contains (name, inode) pairs
  - If one user uses a name, no one else can
  - Many ancient personal computers work this way

- Approach 2: Single directory for each user
  - Still clumsy, and 1,800,000 files is a real pain

- Approach 3: Hierarchical name spaces
  - Allow directory to map names to files or other dirs
  - File system forms a tree (or graph, if links allowed)
  - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)

Hierarchical Unix

- Used since CTSS (1960s)
  - Unix picked up and used really nicely

- Directories stored on disk just like regular files
  - Special inode type byte set to directory
  - User’s can read just like any other file
  - Only special syscalls can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree (what needed to make a DAG?)

  Simple, plus speeding up file ops speeds up dir ops!

Naming magic

- Bootstrapping: Where do you start looking?
  - Root directory always inode #2 (0 and 1 historically reserved)

- Special names:
  - Root directory: “/”
  - Current directory: “.”
  - Parent directory: “..”

- Some special names are provided by shell, not FS:
  - User’s home directory: “~”
  - Globbing: “foo *” expands to all files starting “foo.”

- Using the given names, only need two operations to navigate the entire name space:
  - cd name: move into (change context to) directory name
  - ls: enumerate all names in current directory (context)

Unix example: /a/b/c.c

- Cumbersome to constantly specify full path names
  - In Unix, each process has a “current working directory” (cwd)
  - File names not beginning with “/” are assumed to be relative to cwd; otherwise translation happens as before
  - Editorial: root, cwd should be regular fds (like stdin, stdout, ...) with openat syscall instead of open

- Shells track a default list of active contexts
  - A “search path” for programs you run
  - Given a search path A : B : C, a shell will check in A, then check in B, then check in C
  - Can escape using explicit paths: “./foo”

- Example of locality
**Hard and soft links (synonyms)**

- More than one dir entry can refer to a given file
  - Unix stores count of pointers ("hard links") to inode
  - To make: `ln foo bar` creates a synonym (bar) for file foo
- Soft/symbolic links = synonyms for names
  - Point to a file (or dir) name, but object can be deleted from underneath it (or never even exist).
  - Unix implements like directories: inode has special "symlink" bit set and contains name of link target

**Case study: speeding up FS**

- **Original Unix FS:** Simple and elegant:
  - Components:
    - Data blocks
    - Inodes (directories represented as files)
    - Hard links
    - Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
  - Problem: slow
    - Only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!

**A plethora of performance costs**

- Blocks too small (512 bytes)
  - File index too large
  - Too many layers of mapping indirection
  - Transfer rate low (get one block at time)
- Poor clustering of related objects:
  - Consecutive file blocks not close together
  - Inodes far from data blocks
  - Inodes for directory not close together
  - Poor enumeration performance: e.g., "ls", "grep foo *.c"
- Usability problems
  - 14-character file names a pain
  - Can’t atomically update file in crash-proof way
- Next: how FFS fixes these (to a degree) [McKusic]

**Problem: Internal fragmentation**

- Block size was too small in Unix FS
- Why not just make block size bigger?

<table>
<thead>
<tr>
<th>Block size</th>
<th>space wasted</th>
<th>file bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>6.9%</td>
<td>2.6%</td>
</tr>
<tr>
<td>1024</td>
<td>11.8%</td>
<td>3.3%</td>
</tr>
<tr>
<td>2048</td>
<td>22.4%</td>
<td>6.4%</td>
</tr>
<tr>
<td>4096</td>
<td>45.6%</td>
<td>12.0%</td>
</tr>
<tr>
<td>1MB</td>
<td>99.0%</td>
<td>97.2%</td>
</tr>
</tbody>
</table>

- Bigger block increases bandwidth, but how to deal with wastage ("internal fragmentation")?
  - Use idea from malloc: split unused portion.

**Solution: fragments**

- BSD FFS:
  - Has large block size (4096 or 8192)
  - Allow large blocks to be chopped into small ones ("fragments")
  - Used for little files and pieces at the ends of files

- Best way to eliminate internal fragmentation?
  - Variable sized splits of course
  - Why does FFS use fixed-sized fragments (1024, 2048)?

**Clustering related objects in FFS**

- Group sets of consecutive cylinders into "cylinder groups"
  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group
Clustering in FFS

- Tries to put sequential blocks in adjacent sectors
  - (Access one block, probably access next)
- Tries to keep inode in same cylinder as file data:
  - (If you look at inode, most likely will look at data too)
- Tries to keep all inodes in a dir in same cylinder group
  - Access one name, frequently access many, e.g., “/three.pnum/three.pnum”

What does disk layout look like?

- Each cylinder group basically a mini-Unix file system:
  - cylinder groups
  - superblocks
  - inodes
  - data blocks
  - bookkeeping information
- How to ensure there’s space for related stuff?
  - Place different directories in different cylinder groups
  - Keep a “free space reserve” so can allocate near existing things
  - When file grows too big (/one.pnumMB) send its remainder to different cylinder group.

Finding space for related objs

- Old Unix (& DOS): Linked list of free blocks
  - Just take a block off of the head. Easy.
  - Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow
- FFS: switch to bit-map of free blocks
  - 10101010111111000001111100101100
  - Easier to find contiguous blocks.
  - Small, so usually keep entire thing in memory
  - Time to find free block increases if fewer free blocks

Using a bitmap

- Usually keep entire bitmap in memory:
  - 4G disk / 4K byte blocks. How big is map?
- Allocate block close to block x?
  - Check for blocks near \( \log_2 (x/32) \)
  - If disk almost empty, will likely find one near
  - As disk becomes full, search becomes more expensive and less effective
- Trade space for time (search time, file access time)
- Keep a reserve (e.g., 10%) of disk always free, ideally scattered across disk
  - Don’t tell users (as can get to 110% full)
  - Only root can allocate blocks once FS 100% full
  - With 10% free, can almost always find one of them free

So what did we gain?

- Performance improvements:
  - Able to get 20-40% of disk bandwidth for large files
  - 10-20x original Unix file system!
  - Better small file performance (why?)
- Is this the best we can do? No.
- Block based rather than extent based
  - Could have named contiguous blocks with single pointer and length (Linux ext2/fs, XFS)
- Writes of metadata done synchronously
  - Really hurts small file performance
  - Make asynchronous with write-ordering (“soft updates”) or logging/journaling... more next lecture
  - Play with semantics (/tmp file systems)

Other hacks

- Obvious:
  - Big file cache
- Fact: no rotation delay if get whole track.
  - How to use?
- Fact: transfer cost negligible.
  - Recall: Can get 50x the data for only ~3% more overhead
  - 1 sector: 5ms + 4ms + 5µs (= 512 B/(10 MB/s)) ≈ 9ms
  - 50 sectors: 5ms + 4ms + 2.5ms = 9.25ms
  - How to use?
- Fact: if transfer huge, seek + rotation negligible
- LFS: Hoard data, write out MB at a time
- Next lecture:
  - FFS in more detail
  - More advanced, modern file systems