

File Systems

Operating System Design – M1 Info

Instructor: Vincent Danjean
Class Assistants: Florent Bouchez, Nicolas Fournel

October 21, 2014

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

Outline

I/O Systems

Organization

Communicating With a Device

Hard drives

Flash Memory

File System

File

Inode Organization

Directory Organization

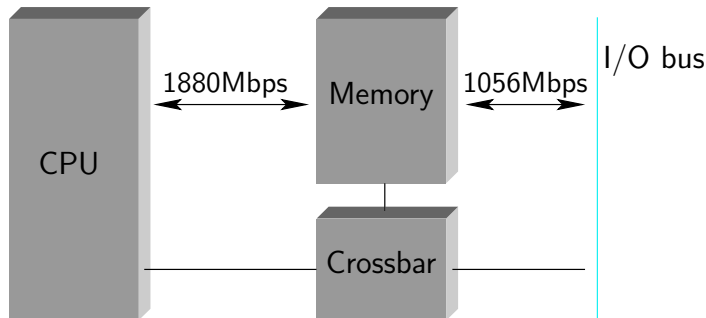
Speeding Up: FFS

Recovering from failures

Ordered Updates

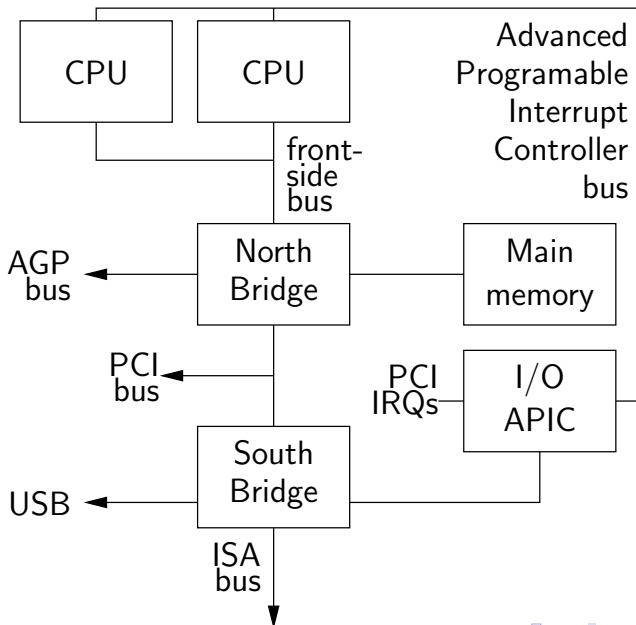
Journaling

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 PC architecture



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

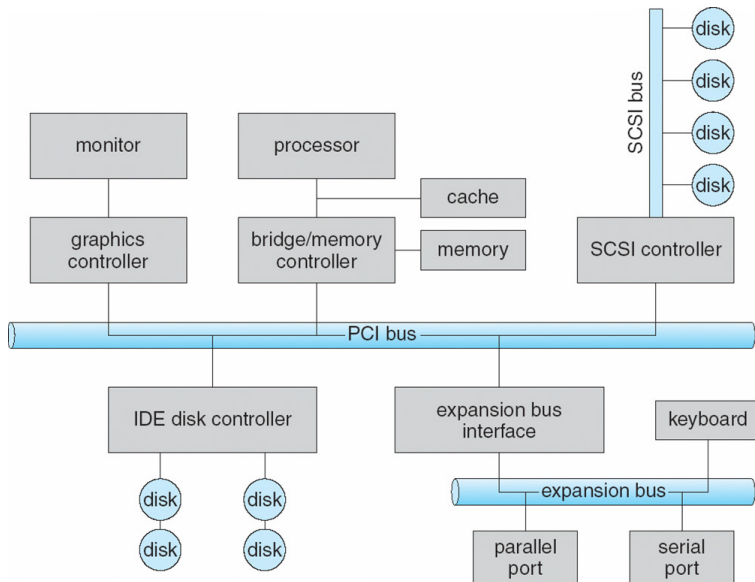
▶ **DRAM – Dynamic RAM**

- ▶ 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, can write while someone else reads

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



Outline

I/O Systems

- Organization

- Communicating With a Device**

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

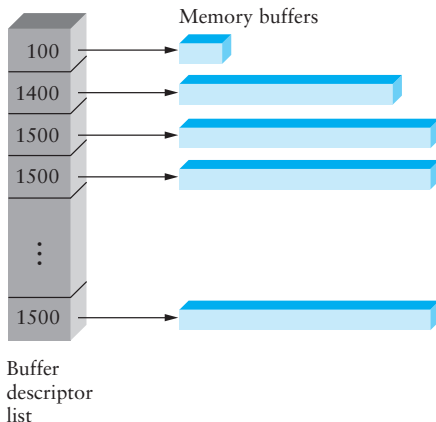
- Ordered Updates

- Journaling

Communicating with a device

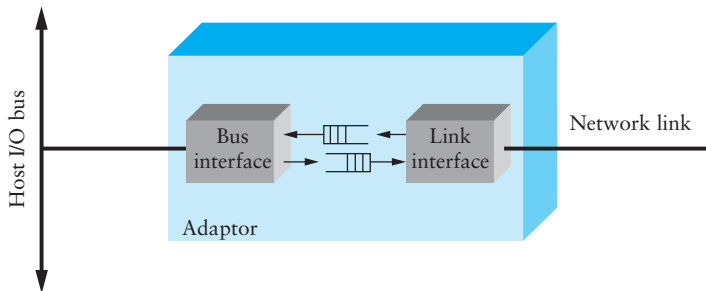
- ▶ **Memory-mapped device registers**
 - ▶ Certain *physical* addresses correspond to device registers
 - ▶ Load/store gets status/sends instructions – not real memory
- ▶ **Device memory – device may have memory OS can write to directly on other side of I/O bus**
- ▶ **Special I/O instructions**
 - ▶ 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 with finer granularity than page
- ▶ **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

DMA buffers



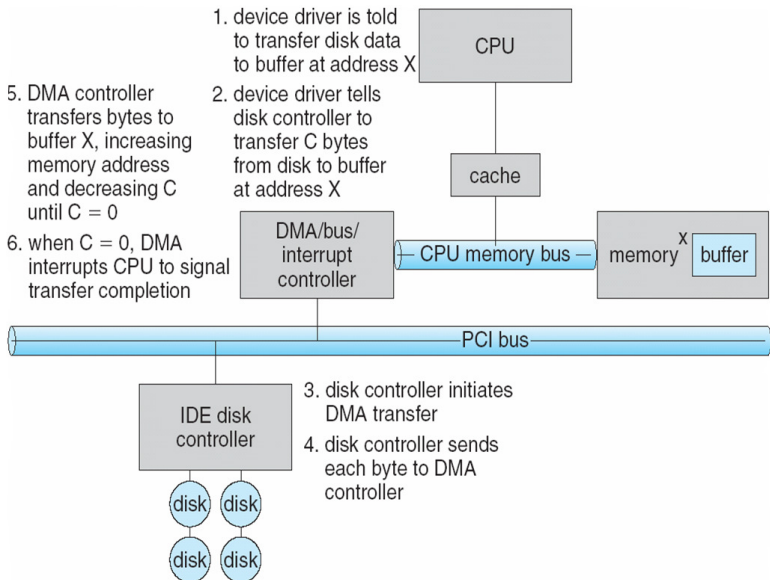
- ▶ **Include list of buffer locations in main memory**
- ▶ **Card reads list then accesses buffers (w. DMA)**
 - ▶ Descriptors sometimes allow for scatter/gather I/O

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



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?**

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
 - ▶ Or schedule poll in future and do something else, but then high latency to receive packet or process disk block

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 1.5 weeks)**

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives**

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

Anatomy of a disk

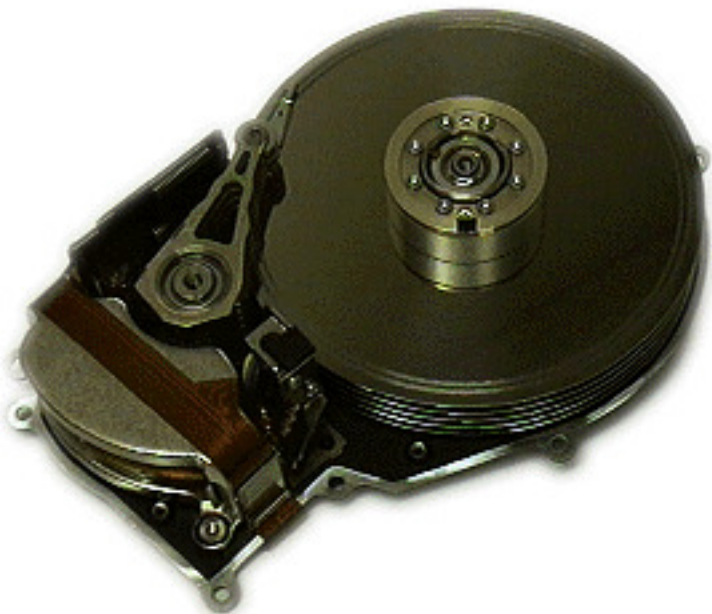
- ▶ **Stack of magnetic platters**

- ▶ 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
- ▶ Arms contain disk heads—one for each recording surface
- ▶ Heads read and write data to platters

Disk



Disk



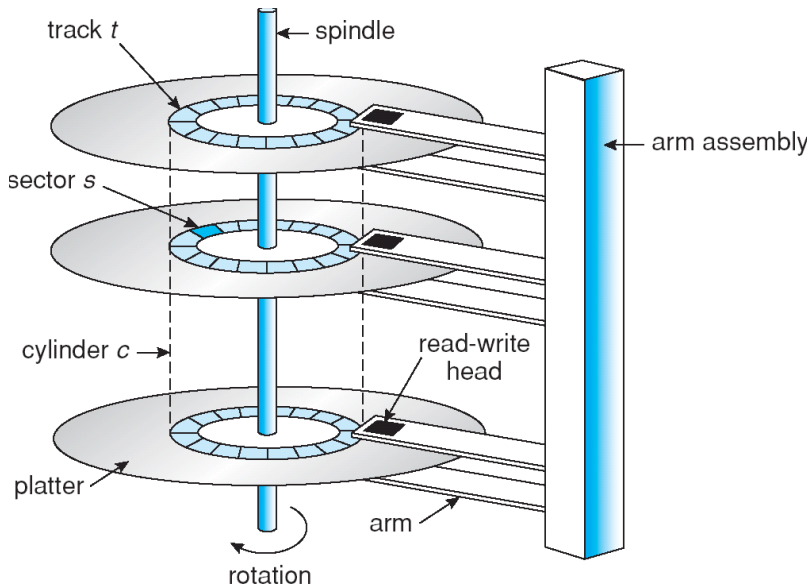
Disk



Storage on a magnetic platter

- ▶ **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



Disk positioning system

- ▶ **Move head to specific track and keep it there**
 - ▶ Resist physical shocks, imperfect tracks, etc.
- ▶ **Seek time depends on:**
 - ▶ Inertial power of the arm actuator motor
 - ▶ Distance between outer-disk recording radius and inner-disk recording radius (data-band)
 - ▶ Depends on platter-size
- ▶ **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?

Seek details

- ▶ **Head switches comparable to short seeks**
 - ▶ May also require head adjustment
 - ▶ Settles take longer for writes than for reads
 - If read strays from track, catch error with checksum, retry
 - If write strays, you've just clobbered some other track
- ▶ **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

Sectors

- ▶ **Bits are grouped into sectors: generally 512 bytes + overhead information**
 - ▶ Error Correcting Codes
 - ▶ Servo fields to properly position the head
- ▶ **Disk interface presents linear array of sectors**
 - ▶ Also 512 bytes, written atomically (even if power failure)
- ▶ **Disk maps logical sector #s to physical sectors**
 - ▶ *Zoning*—puts more sectors on longer tracks
 - ▶ *Track and Cylinder skewing*—sector 0 pos. varies by track (why?)
 - ▶ *Sparing*—flawed sectors remapped elsewhere
- ▶ **OS doesn't know logical to physical sector mapping**
 - ▶ Larger logical sector # difference means larger seek
 - ▶ Highly non-linear relationship (*and* depends on zone)
 - ▶ OS has no info on rotational positions
 - ▶ Can empirically build table to estimate times

Sectors

- ▶ **Bits are grouped into sectors: generally 512 bytes + over-head information**
 - ▶ Error Correcting Codes
 - ▶ Servo fields to properly position the head
- ▶ **Disk interface presents linear array of sectors**
 - ▶ Also 512 bytes, written atomically (even if power failure)
- ▶ **Disk maps logical sector #s to physical sectors**
 - ▶ *Zoning*—puts more sectors on longer tracks
 - ▶ *Track and Cylinder skewing*—sector 0 pos. varies by track (known head and cylinder switch time \leadsto sequential access speed optimization)
 - ▶ *Sparing*—flawed sectors remapped elsewhere
- ▶ **OS doesn't know logical to physical sector mapping**
 - ▶ Larger logical sector # difference means larger seek
 - ▶ Highly non-linear relationship (*and* depends on zone)
 - ▶ OS has no info on rotational positions
 - ▶ Can empirically build table to estimate times

Disk review

- ▶ **Disk reads/writes in terms of sectors, not bytes**
 - ▶ Read/write single sector or adjacent groups (cluster)



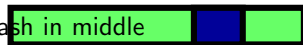
- ▶ **How to write a single byte? “Read-modify-write”**

- ▶ 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**

Disk interface

- ▶ **Controls hardware, mediates access**
- ▶ **Computer, disk often connected by bus (e.g., SCSI)**
 - ▶ 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

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 lecture
- ▶ **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**

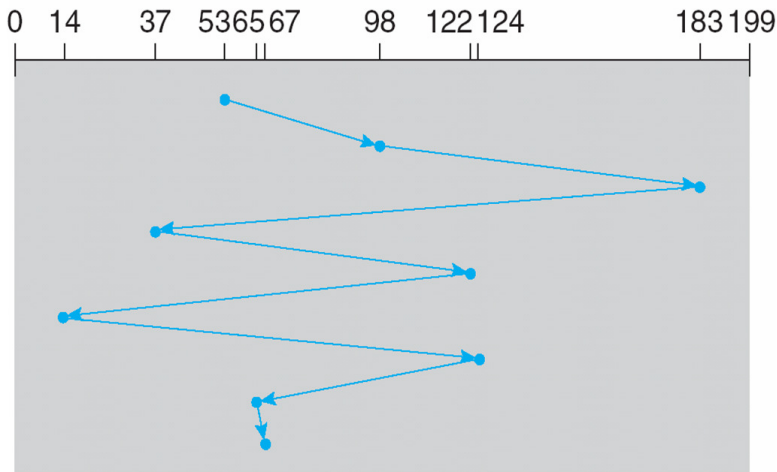
Scheduling: FCFS

- ▶ **“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

FCFS example

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Shortest positioning time first (SPTF)

- ▶ **Shortest positioning time first (SPTF)**
 - ▶ Always pick request with shortest seek time
- ▶ **Advantages**
- ▶ **Disadvantages**
- ▶ **Improvement**

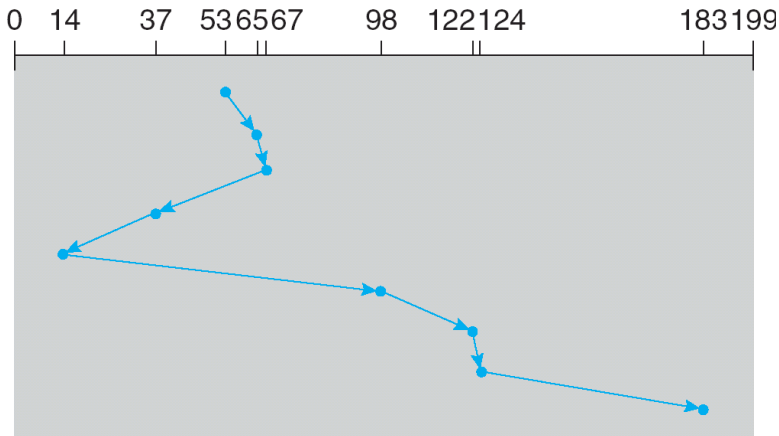
Shortest positioning time first (SPTF)

- ▶ **Shortest positioning time first (SPTF)**
 - ▶ Always pick request with shortest seek time
- ▶ **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:
$$T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}}$$
- ▶ **Also called Shortest Seek Time First (SSTF)**

SPTF example

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



“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**
- ▶ **Disadvantages**

“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)

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory**

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

Flash memory

- ▶ **Today, people increasingly using flash memory**
- ▶ **Completely solid state (no moving parts)**
 - ▶ Remembers data by storing charge
 - ▶ Lower power consumption and heat
 - ▶ 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 easily lose data

Disk vs. Memory

| | Disk | MLC NAND Flash | DRAM |
|------------------|--------------|-------------------|------------|
| Smallest write | sector | sector | byte |
| Atomic write | sector | sector | byte/word |
| Random read | 8 ms | 75 μ s | 50 ns |
| Random write | 8 ms | 300 μ s* | 50 ns |
| Sequential read | 100 MB/s | 250 MB/s | > 1 GB/s |
| Sequential write | 100 MB/s | 170 MB/s* | > 1 GB/s |
| Cost | \$.08–1/GB | \$3/GB | \$10-25/GB |
| Persistence | Non-volatile | Non-volatile | Volatile |

*Flash write performance degrades over time

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

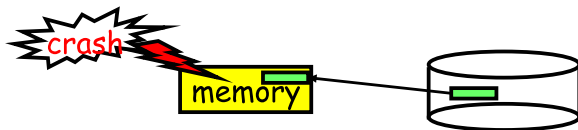
- Journaling

File system fun

- ▶ **File systems = the 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**

The medium is the message

- ▶ **Disk = First thing we've seen that doesn't go away**



- ▶ So: Where all important state ultimately resides
- ▶ **Slow (ms access vs ns for memory)**



- ▶ **Huge (100–1,000x bigger than memory)**
 - ▶ How to organize large collection of ad hoc information?
 - ▶ Taxonomies! (Basically FS = general way to make these)

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

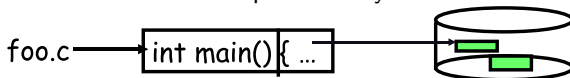
- Ordered Updates

- Journaling

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:



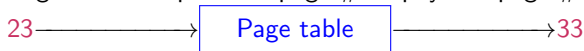
► File operations:

- Create a file, delete a file
 - Read from file, write to file
 - Repositioning within a file
 - Truncating a file, append, rename, ...
- **File meta-informations** (size, owner, access rights, timestamps, ...)
 - **Want: operations to have as few disk accesses as possible & have minimal space overhead**

What's hard about grouping blocks?

- ▶ Like page tables, file system meta data are simply data structures used to construct mappings

- ▶ Page table: map virtual page # to physical page #



- ▶ File meta data: map byte offset to disk block address



- ▶ Directory: map name to disk address or file #



- ▶ Inode stores meta-information (not name!) and file bytes location.

FS vs. VM

- ▶ **In both settings, want location transparency**
- ▶ **In some ways, FS has easier job 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 ($0 \dots \text{filesize} - 1$) & \sim 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

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

Some working intuitions

- ▶ **FS performance dominated by # of disk accesses**
 - ▶ Each access costs ~ 10 milliseconds
 - ▶ Touch the disk 100 extra times = 1 *second*
 - ▶ Can easily do 100s of millions of ALU ops in same time
- ▶ **Access cost dominated by movement, not transfer:**

seek time + **rotational delay** + # bytes/disk-bw

- ▶ Can get 50x the data for only $\sim 3\%$ more overhead
 - ▶ 1 sector: $10\text{ms} + 8\text{ms} + 10\mu\text{s}$ ($= 512\text{ B}/(50\text{ MB/s})$) $\approx 18\text{ms}$
 - ▶ 50 sectors: $10\text{ms} + 8\text{ms} + .5\text{ms} = 18.5\text{ms}$
- ▶ **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**
- ▶ File descriptors 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?)

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**
- ▶ File descriptors must be stored on disk, too

- ▶ **Things to keep in mind while designing file structure:**

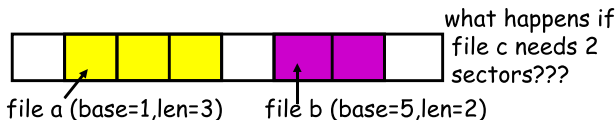
- ▶ Most files are small
- ▶ Much of the disk is allocated to large files
- ▶ Many of the I/O operations are made to large files
- ▶ Want good sequential and good random access (what do these require?)

- ▶ **Just like VM: good data structures**

- ▶ Arrays, linked list, trees (of arrays), hash tables.

Straw man: contiguous allocation

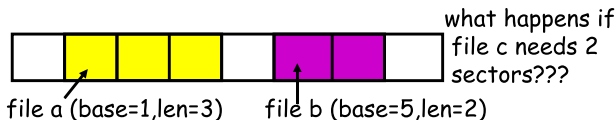
- ▶ **“Extent-based”**: allocate files like segmented memory
 - ▶ When creating a file, make the user specify pre-specify its length and allocate all space at once
 - ▶ Inode contents: location and size



- ▶ **Example: IBM OS/360**
- ▶ **Pros?**
- ▶ **Cons? (What VM scheme does this correspond to?)**

Straw man: contiguous allocation

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

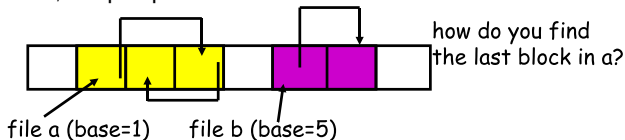


- ▶ **Example: IBM OS/360**
- ▶ **Pros?**
 - ▶ Simple, fast access, both sequential and random
- ▶ **Cons? (What VM scheme does this correspond to?)**
 - ▶ External fragmentation

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**

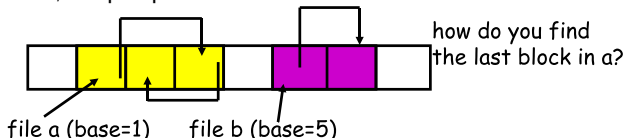
- ▶ **Pros?**

- ▶ **Cons?**

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?**

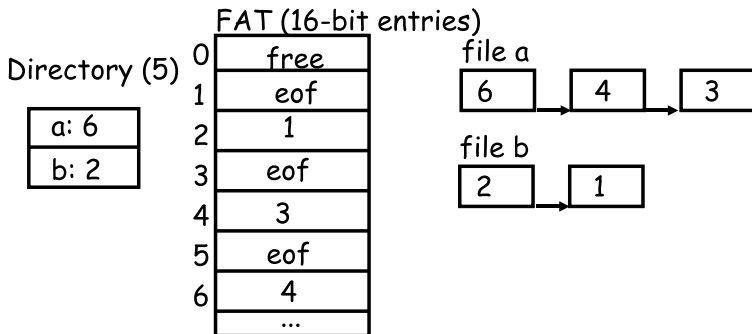
- ▶ Easy dynamic growth & sequential access, no fragmentation

- ▶ **Cons?**

- ▶ Linked lists on disk a bad idea because of access times
- ▶ Pointers take up room in block, skewing alignment
- ▶ If one pointer is ever damaged, the rest of the file is lost.

Example: DOS FS (simplified)

- Uses linked files. Cute: links reside in fixed-sized “file allocation table” (FAT) rather than in the blocks.



- 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?
 - ▶ Given a 512 byte block, what's the maximum size of FS?
 - ▶ One attack: go to bigger blocks. Pros? Cons?
- ▶ **Space overhead of FAT is trivial:**
 - ▶ 2 bytes / 512 byte block = $\sim 0.4\%$ (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:



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 MB
- ▶ One attack: go to bigger blocks. Pros? Cons?

- ▶ **Space overhead of FAT is trivial:**

- ▶ 2 bytes / 512 byte block = $\sim 0.4\%$ (Compare to Unix)

- ▶ **Reliability: how to protect against errors?**

- ▶ Create duplicate copies of FAT on disk.
- ▶ State duplication a very common theme in reliability

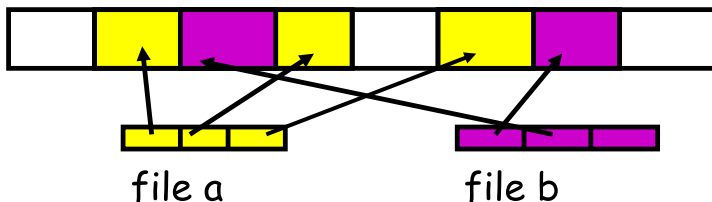
- ▶ **Bootstrapping: where is root directory?**

- ▶ Fixed location on disk:



Indexed files

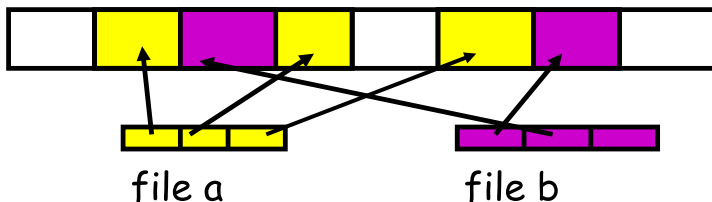
- ▶ **Each file has an array holding all of it's block pointers**
 - ▶ Just like a page table, so will have similar issues
 - ▶ Max file size fixed by array's size (static or dynamic?)
 - ▶ Allocate array to hold file's block pointers on file creation
 - ▶ Allocate actual blocks on demand using free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



- ▶ **Pros?**
- ▶ **Cons?**

Indexed files

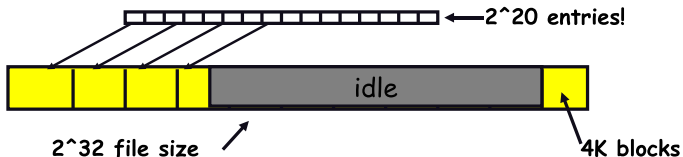
- ▶ **Each file has an array holding all of it's block pointers**
 - ▶ Just like a page table, so will have similar issues
 - ▶ Max file size fixed by array's size (static or dynamic?)
 - ▶ Allocate array to hold file's block pointers on file creation
 - ▶ Allocate actual blocks on demand using free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



- ▶ **Pros?**
 - ▶ Both sequential and random access easy
- ▶ **Cons?**
 - ▶ Mapping table requires large chunk of contiguous space
... Same problem we were trying to solve initially

Indexed files

- ▶ Issues same as in page tables

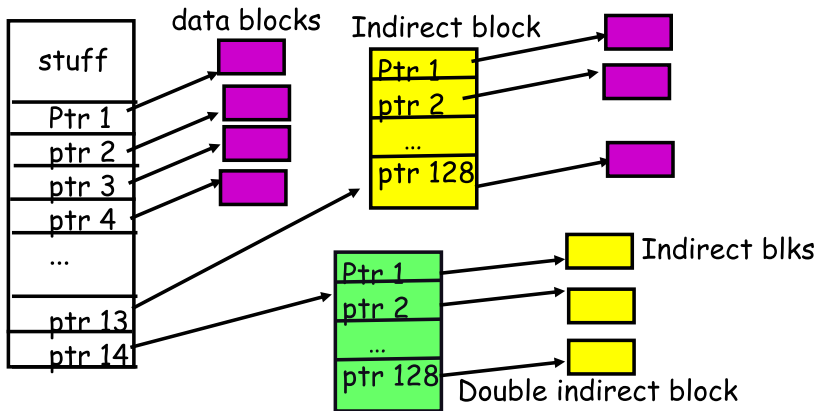


- ▶ Large possible file size = lots of unused entries
- ▶ Large actual size? table needs large contiguous disk chunk
- ▶ **Solve identically: small regions with index array, this array with another array, ... Downside?**



Multi-level indexed files (old BSD FS)

- inode = 14 block pointers + “stuff” (meta-informations)



Old BSD FS discussion

- ▶ **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 free-list, meta data and data get strewn across disk

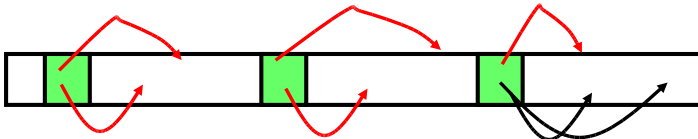
More about inodes

- ▶ **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:



- ▶ 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

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization**

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

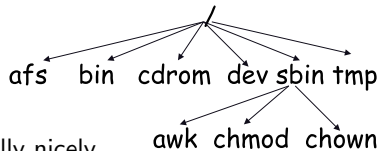
Directories

- ▶ **Problem:**
 - ▶ “Spend all day generating data, come back the next morning, want to use it.” F. Corbato, on why files/dirs invented.
- ▶ **Approach 0: Have 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 $\langle \text{name}, \text{inumber} \rangle$ 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 1s on 10,000 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**

- ▶ Inode contains special flag bit set dir
- ▶ User's can read just like any other file
- ▶ Only special programs 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?)

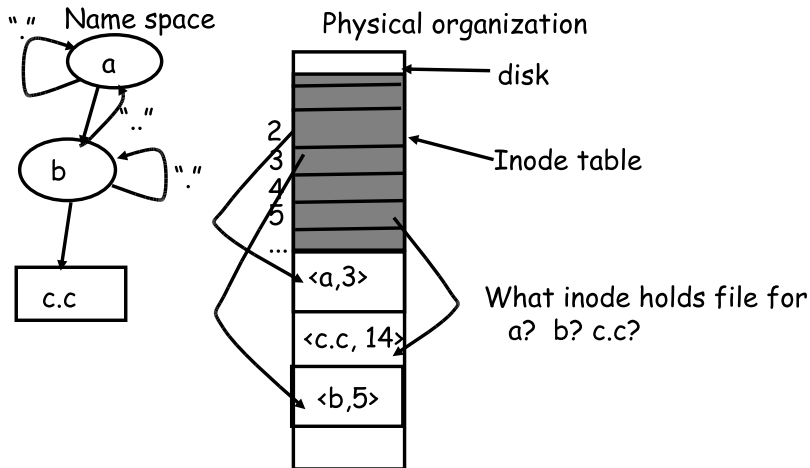
| <code><name,inode#></code> |
|----------------------------------|
| <code><afs,1021></code> |
| <code><tmp,1020></code> |
| <code><bin,1022></code> |
| <code><cdrom,4123></code> |
| <code><dev,1001></code> |
| <code><sbin,1011></code> |
| <code>:</code> |

- ▶ **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: `“..”`
- ▶ **Special names not implemented in 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



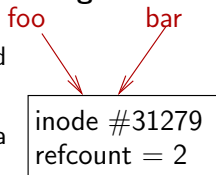
Default context: working directory

- ▶ **Cumbersome to constantly specify full path names**
 - ▶ In Unix, each process associated with a “current working directory”
 - ▶ File names that do not begin with “/” are assumed to be relative to the working directory, otherwise translation happens as before
- ▶ **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”

Hard and soft links (synonyms)

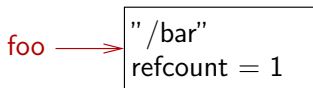
► More than one dir entry can refer to a given file

- Unix stores count of pointers (“hard links”) to inode
- To make: “ln foo bar” creates a synonym (bar) for *file* foo



► Soft 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 “sym link” bit set and contains pointed to name



- To make: “ln -sf bar baz
- When the file system encounters a symbolic link it automatically translates it (if possible).

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

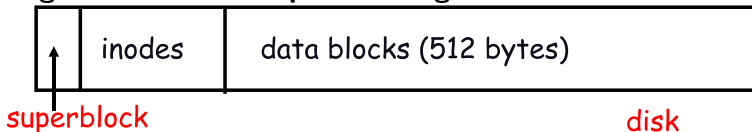
Recovering from failures

- Ordered Updates

- Journaling

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)
- ▶ **Sucky 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”
- ▶ **Next: how FFS fixes these problems (to a degree)**

Problem: Internal fragmentation

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

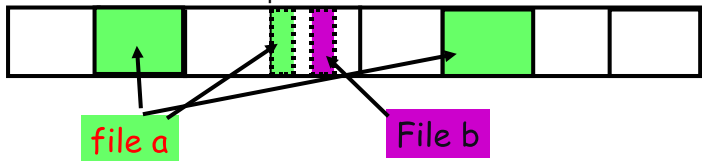
| Block size | space wasted | file bandwidth |
|------------|--------------|----------------|
| 512 | 6.9% | 2.6% |
| 1024 | 11.8% | 3.3% |
| 2048 | 22.4% | 6.4% |
| 4096 | 45.6% | 12.0% |
| 1MB | 99.0% | 97.2% |

- ▶ 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

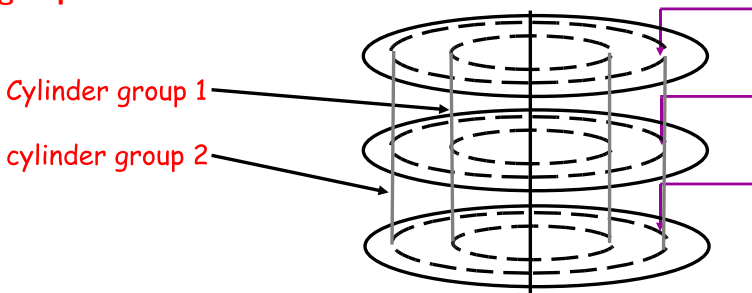


► Best way to eliminate internal fragmentation?

- Variable sized splits of course
- Why does FFS use fixed-sized fragments (1024, 2048)?

Clustering related objects in FFS

- ▶ **Group 1 or more consecutive cylinders into a “cylinder group”**

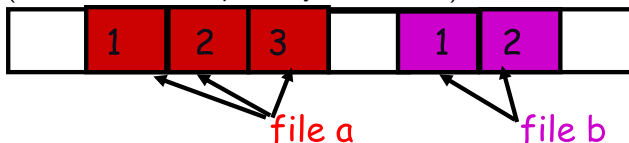


- ▶ Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
- ▶ Tries to put everything related in same cylinder group
- ▶ Tries to put everything not related in different group (?!)

Clustering in FFS

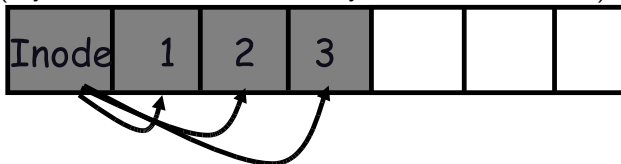
- Tries to put sequential blocks in adjacent sectors

- (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)

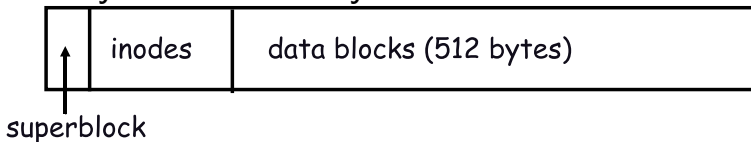


- Tries to keep all inodes in a dir in same cylinder group

- Access one name, frequently access many, e.g., "ls -l"

What does a cyl. group look like?

- ▶ **Basically a mini-Unix file system:**



- ▶ **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 (1MB) 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**
 - ▶ 1010101111111000001111111000101100
 - ▶ Easier to find contiguous blocks.
 - ▶ Small, so usually keep entire thing in memory
 - ▶ Key: keep a reserve of free blocks. Makes finding a close block easier

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 $\text{bmap}[x/32]$
 - ▶ 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 ($\text{df} \rightarrow 110\%$ 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**
 - ▶ Name contiguous blocks with single pointer and length
 - ▶ (Linux ext2fs)
- ▶ **Writes of meta data done synchronously**
 - ▶ Really hurts small file performance
 - ▶ Make asynchronous with write-ordering (“soft updates”) or logging (the episode file system, ~LFS)
 - ▶ 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 $\sim 3\%$ more overhead
 - ▶ 1 sector: $10\text{ms} + 8\text{ms} + 10\mu\text{s}$ ($= 512\text{ B}/(50\text{ MB/s})$) $\approx 18\text{ms}$
 - ▶ 50 sectors: $10\text{ms} + 8\text{ms} + .5\text{ms} = 18.5\text{ms}$
 - ▶ How to use?
- ▶ **Fact: if transfer huge, seek + rotation negligible**
 - ▶ How to use ?

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 $\sim 3\%$ more overhead
 - ▶ 1 sector: $10\text{ms} + 8\text{ms} + 10\mu\text{s}$ ($= 512\text{ B}/(50\text{ MB/s})$) $\approx 18\text{ms}$
 - ▶ 50 sectors: $10\text{ms} + 8\text{ms} + .5\text{ms} = 18.5\text{ms}$
 - ▶ How to use?
- ▶ **Fact: if transfer huge, seek + rotation negligible**
 - ▶ How to use ?

Use read ahead + cluster read/write (hoard data, write out MB at a time)

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

Fixing corruption – fsck

- ▶ **Must run FS check (fsck) program after crash**
- ▶ **Summary info usually bad after crash**
 - ▶ Scan to check free block map, block/inode counts
- ▶ **System may have corrupt inodes (not simple crash)**
 - ▶ Bad block numbers, cross-allocation, etc.
 - ▶ Do sanity check, clear inodes with garbage
- ▶ **Fields in inodes may be wrong**
 - ▶ Count number of directory entries to verify link count, if no entries but count $\neq 0$, move to lost+found
 - ▶ Make sure size and used data counts match blocks
- ▶ **Directories may be bad**
 - ▶ Holes illegal, “.” and “..” must be valid, ...
 - ▶ All directories must be reachable

Crash recovery permeates FS code

- ▶ **Have to ensure fsck can recover file system**
- ▶ **Example: Suppose all data written asynchronously**
- ▶ **Delete/truncate a file, append to other file, crash**
 - ▶ New file may reuse block from old
 - ▶ Old inode may not be updated
 - ▶ Cross-allocation!
 - ▶ Often inode with older mtime wrong, but can't be sure
- ▶ **Append to file, allocate indirect block, crash**
 - ▶ Inode points to indirect block
 - ▶ But indirect block may contain garbage

Ordering of updates

- ▶ **Must be careful about order of updates**
 - ▶ Write new inode to disk before directory entry
 - ▶ Remove directory name before deallocating inode
 - ▶ Write cleared inode to disk before updating CG free map
- ▶ **Solution: Many metadata updates synchronous**
 - ▶ Doing one write at a time ensures ordering
 - ▶ Of course, this hurts performance
 - ▶ E.g., `untar` much slower than disk bandwidth
- ▶ **Note: Cannot update buffers on the disk queue**
 - ▶ E.g., say you make two updates to same directory block
 - ▶ But crash recovery requires first to be synchronous
 - ▶ Must wait for first write to complete before doing second

Performance vs. consistency

- ▶ **FFS crash recoverability comes at huge cost**
 - ▶ Makes tasks such as `untar` easily 10-20 times slower
 - ▶ All because you *might* lose power or reboot at any time
- ▶ **Even while slowing ordinary usage, recovery slow**
 - ▶ If `fsck` takes one minute, then disks get $10\times$ bigger ...
- ▶ **One solution: battery-backed RAM**
 - ▶ Expensive (requires specialized hardware)
 - ▶ Often don't learn battery has died until too late
 - ▶ A pain if computer dies (can't just move disk)
 - ▶ If OS bug causes crash, RAM might be garbage
- ▶ **Better solution: Advanced file system techniques**
 - ▶ Topic of rest of lecture

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

First attempt: Ordered updates

- ▶ **Must follow three rules in ordering updates:**
 1. Never write pointer before initializing the structure it points to
 2. Never reuse a resource before nullifying all pointers to it
 3. Never clear last pointer to live resource before setting new one
- ▶ **If you do this, file system will be recoverable**
- ▶ **Moreover, can recover quickly**
 - ▶ Might leak free disk space, but otherwise correct
 - ▶ So start running after reboot, scavenge for space in background
- ▶ **How to achieve?**
 - ▶ Keep a partial order on buffered blocks

Ordered updates (continued)

- ▶ **Example: Create file A**
 - ▶ Block X contains an inode
 - ▶ Block Y contains a directory block
 - ▶ Create file A in inode block X , dir block Y
- ▶ **We say $Y \rightarrow X$, pronounced “ Y depends on X ”**
 - ▶ Means Y cannot be written before X is written
 - ▶ X is called the **dependee**, Y the **depender**
- ▶ **Can delay both writes, so long as order preserved**
 - ▶ Say you create a second file B in blocks X and Y
 - ▶ Only have to write each out once for both creates


Problem: Cyclic dependencies

- ▶ **Suppose you create file *A*, unlink file *B***
 - ▶ Both files in same directory block & inode block
- ▶ **Can't write directory until inode *A* initialized**
 - ▶ Otherwise, after crash directory will point to bogus inode
 - ▶ Worse yet, same inode # might be re-allocated
 - ▶ So could end up with file name *A* being an unrelated file
- ▶ **Can't write inode block until dir entry *B* cleared**
 - ▶ Otherwise, *B* could end up with too small a link count
 - ▶ File could be deleted while links to it still exist
- ▶ **Otherwise, fsck has to be very slow**
 - ▶ Check every directory entry and inode link count


Cyclic dependencies illustrated

Inode Block Directory Block



 = in use

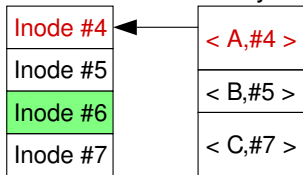
 Original

 = free

 Modified

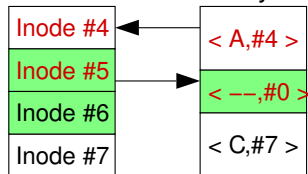
(a) Original Organization

Inode Block Directory Block



(b) Create File A

Inode Block Directory Block



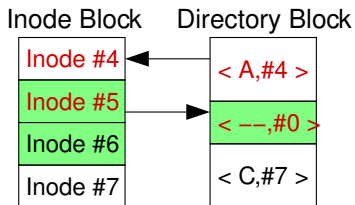
(c) Remove file B

More problems

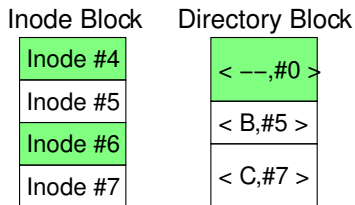
- ▶ **Crash might occur between ordered but related writes**
 - ▶ E.g., summary information wrong after block freed
- ▶ **Block aging**
 - ▶ Block that always has dependency will never get written back
- ▶ **Solution: “Soft updates” [Ganger]**
 - ▶ Write blocks in any order
 - ▶ But keep track of dependencies
 - ▶ When writing a block, temporarily roll back any changes you can't yet commit to disk

Breaking dependencies w. rollback

Main Memory



Disk

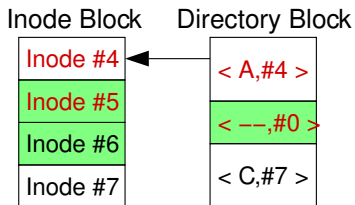


(a) After Metadata Updates

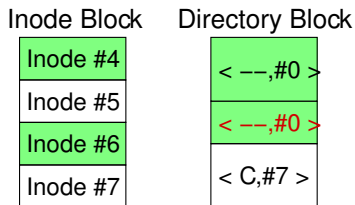
- ▶ Now say we decide to write directory block...
- ▶ Can't write file name A to disk—has dependee

Breaking dependencies w. rollback

Main Memory



Disk

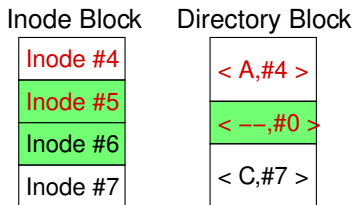


(b) Safe Version of Directory Block Written

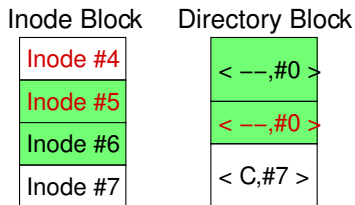
- ▶ **Undo file A before writing dir block to disk**
 - ▶ Even though we just wrote it, directory block still
- ▶ **But now inode block has no dependees**
 - ▶ Can safely write inode block to disk as-is. . .

Breaking dependencies w. rollback

Main Memory



Disk

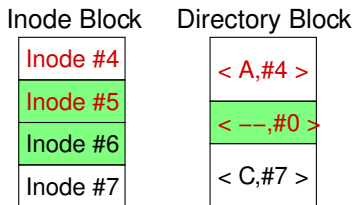


(c) Inode Block Written

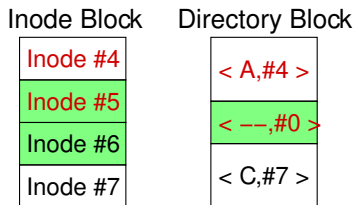
- ▶ Now inode block clean (same in memory as on disk)
- ▶ But have to write directory block a second time...

Breaking dependencies w. rollback

Main Memory



Disk



(d) Directory Block Written

- ▶ All data stably on disk
- ▶ Crash at any point would have been safe

Soft updates

- ▶ **Structure for each updated field or pointer, contains:**
 - ▶ old value
 - ▶ new value
 - ▶ list of updates on which this update depends (*dependees*)
- ▶ **Can write blocks in any order**
 - ▶ But must temporarily undo updates with pending dependencies
 - ▶ Must lock rolled-back version so applications don't see it
 - ▶ Choose ordering based on disk arm scheduling
- ▶ **Some dependencies better handled by postponing in-memory updates**
 - ▶ E.g., when freeing block (e.g., because file truncated), just mark block free in bitmap after block pointer cleared on disk

Simple example

- ▶ **Say you create a zero-length file A**
- ▶ **Depender: Directory entry for A**
 - ▶ Can't be written until dependees on disk
- ▶ **Dependees:**
 - ▶ Inode – must be initialized before dir entry written
 - ▶ Bitmap – must mark inode allocated before dir entry written
- ▶ **Old value: empty directory entry**
- ▶ **New value:** $\langle \text{filename } A, \text{inode } \# \rangle$
- ▶ **Can write directory block to disk any time**
 - ▶ Must substitute old value until inode & bitmap updated on disk
 - ▶ Once dir block on disk contains A , file fully created
 - ▶ Crash before A on disk, worst case might leak the inode

Operations requiring soft updates (1)

1. Block allocation

- ▶ Must write the disk block, the free map, & a pointer
- ▶ Disk block & free map must be written before pointer
- ▶ Use Undo/redo on pointer (& possibly file size)

2. Block deallocation

- ▶ Must write the cleared pointer & free map
- ▶ Just update free map after pointer written to disk
- ▶ Or just immediately update free map if pointer not on disk
- ▶ **Say you quickly append block to file then truncate**
 - ▶ You will know pointer to block not written because of the allocated dependency structure
 - ▶ So both operations together require no disk I/O!

Operations requiring soft updates (2)

3. Link addition (see **simple example**)

- ▶ Must write the directory entry, inode, & free map (if new inode)
- ▶ Inode and free map must be written before dir entry
- ▶ Use undo/redo on $i\#$ in dir entry (ignore entries w. $i\# = 0$)

4. Link removal

- ▶ Must write directory entry, inode & free map (if $nlinks == 0$)
 - ▶ Must decrement $nlinks$ only after pointer cleared
 - ▶ Clear directory entry immediately
 - ▶ Decrement in-memory $nlinks$ once pointer written
 - ▶ If directory entry was never written, decrement immediately (again will know by presence of dependency structure)
- ▶ **Note: Quick create/delete requires no disk I/O**

Soft update issues

- ▶ **fsync – syscall to flush file changes to disk**
 - ▶ Must also flush directory entries, parent directories, etc.
- ▶ **unmount – flush all changes to disk on shutdown**
 - ▶ Some buffers must be flushed multiple times to get clean
- ▶ **Deleting large directory trees frighteningly fast**
 - ▶ *unlink* syscall returns even if inode/indir block not cached!
 - ▶ Dependencies allocated faster than blocks written
 - ▶ Cap # dependencies allocated to avoid exhausting memory
- ▶ **Useless write-backs**
 - ▶ Syncer flushes dirty buffers to disk every 30 seconds
 - ▶ Writing all at once means many dependencies unsatisfied
 - ▶ Fix syncer to write blocks one at a time
 - ▶ Fix LRU buffer eviction to know about dependencies

Soft updates fsck

- ▶ **Split into foreground and background parts**
- ▶ **Foreground must be done before remounting FS**
 - ▶ Need to make sure per-cylinder summary info makes sense
 - ▶ Recompute free block/inode counts from bitmaps – very fast
 - ▶ Will leave FS consistent, but might leak disk space
- ▶ **Background does traditional fsck operations**
 - ▶ Do after mounting to recuperate free space
 - ▶ Can be using the file system while this is happening
 - ▶ Must be done in foreground after a media failure
- ▶ **Difference from traditional FFS fsck:**
 - ▶ May have many, many inodes with non-zero link counts
 - ▶ Don't stick them all in lost+found (unless media failure)

Outline

I/O Systems

- Organization

- Communicating With a Device

- Hard drives

- Flash Memory

File System

- File

- Inode Organization

- Directory Organization

- Speeding Up: FFS

Recovering from failures

- Ordered Updates

- Journaling

An alternative: Journaling

- ▶ **Reserve a portion of disk for write-ahead log**
 - ▶ Write any metadata operation first to log, then to disk
 - ▶ After crash/reboot, re-play the log (efficient)
 - ▶ May re-do already committed change, but won't miss anything
- ▶ **Performance advantage:**
 - ▶ Log is consecutive portion of disk
 - ▶ Multiple log writes very fast (at disk b/w)
 - ▶ Consider updates committed when written to log
- ▶ **Example: delete directory tree**
 - ▶ Record all freed blocks, changed directory entries in log
 - ▶ Return control to user
 - ▶ Write out changed directories, bitmaps, etc. in background (sort for good disk arm scheduling)

Journaling details

- ▶ **Must find oldest relevant log entry**
 - ▶ Otherwise, redundant and slow to replay whole log
- ▶ **Use checkpoints**
 - ▶ Once all records up to log entry N have been processed and affected blocks stably committed to disk. . .
 - ▶ Record N to disk either in reserved checkpoint location, or in checkpoint log record
 - ▶ Never need to go back before most recent checkpointed N
- ▶ **Must also find end of log**
 - ▶ Typically circular buffer; don't play old records out of order
 - ▶ Can include begin transaction/end transaction records
 - ▶ Also typically have checksum in case some sectors bad

Journaling vs. soft updates

- ▶ **Both much better than FFS alone**
- ▶ **Some limitations of soft updates**
 - ▶ Very specific to FFS data structures (E.g., couldn't easily complex data structures like B-trees in XFS—even directory rename not quite right)
 - ▶ Metadata updates may proceed out of order (E.g., create *A*, create *B*, crash—maybe only *B* exists after reboot)
 - ▶ Still need slow background fsck to reclaim space
- ▶ **Some limitations of journaling**
 - ▶ Disk write required for every metadata operation (whereas create-then-delete might require no I/O w. soft updates)
 - ▶ Possible contention for end of log on multi-processor
 - ▶ *fsync* must sync other operations' metadata to log, too