File Systems
SGG: Chapters 10 & 11

I. Overview & Background

A. Why do we care about data?
   1. It's half a Turing Machine
   2. Typically we're computing on data, and for data
      a) Sometimes {a lot, a little bit} of data goes in, {a small, a large} amount of data comes out
      b) E.g. Scientific computing: modeling large systems to answering simple questions (how does this protein fold?)
      c) In 2009:
         (1) Google processed 24 PB of data per day
         (2) Facebook claimed to store 1.5 PB of photo data
         (3) Internet Archive stored 2PB of data, growing 20TB per month
         (4) Avatar took up 1PB of storage for renderings

B. We need ways of efficiently accessing and organizing data.

C. How do we recover from failure?
   1. How do we maintain consistency in the face of failure?
   2. How do we model failures?

D. Mitigating the performance limitations of storage devices

E. How do we manage evolving schemas and data types, from increasingly diverse sources

F. How do we scale data systems to support 100-100M users?

G. How do we make it secure?

H. Fuzzy issues: reduce costs, simplify management, be compliant, limit liability

II. History of Storage

A. Punch Card

1. 1725: used to store instructions for controlling a textile loom
2. 1884: used to store partial computations for mechanical calculators
3. 1950s: used as means for storing persistent “digital” data

B. Tape

1. 1952: magnetic material to store data; used to store applications and shared libraries; often used in conjunction with punch cards
2. Very high capacity, low cost
3. Sequential access
4. Relatively durable

C. Disk

1. 1956: IBM RAMAC (Random Access Method of Accounting and Control)
   a) Weight: over 1 ton
   b) Capacity: 5MB
   c) Cost: $3200/month (1960) (lease); ~$28K/month (2013)
2. Continues to be the cheapest, (per-bit) random access device on the market (for now)

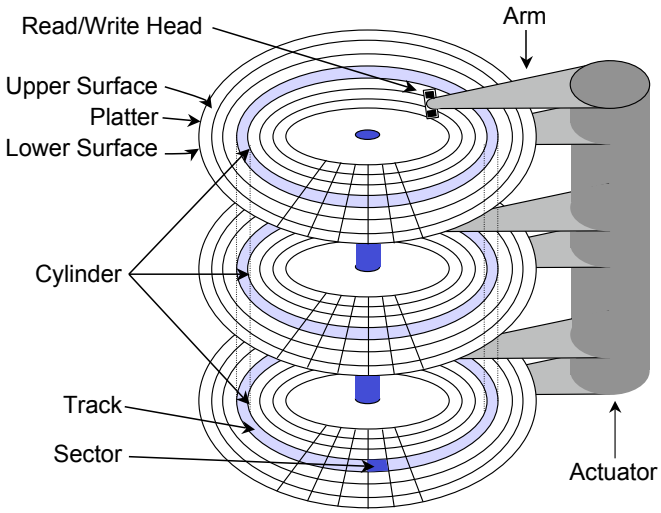
D. Solid State Drives

1. 1950s: Magnetic Core memory
2. 1960s: EEPROM
   a) fully modifiable (read, write, erase) at the byte level. Costs: additional circuitry is expensive and takes up space, writing is very slow, increases wear (minimally). Usually used in when capacity is not an issue: firmware.
3. 1980s: Toshiba introduced NOR Flash (flash meaning it could erased all at once).
   a) Greater capacities than EEPROM, decrease granularity in erasures. Read and written by the byte, but erased at a block. Further, this memory organization exhibits wear-leveling.
4. Today: NAND Flash Memory

a) Cheaper and smaller than NOR, NAND has superseded NOR in the SSD market. Current chips up to 200GB. Further decreases in granularity: reads and writes on a page/chunk (not byte), erases on a block. NAND pages have additional space (out-of-band) for EECs; this allows for a higher tolerance in defects, and cheaper production costs.

III. Hard Disk Drive

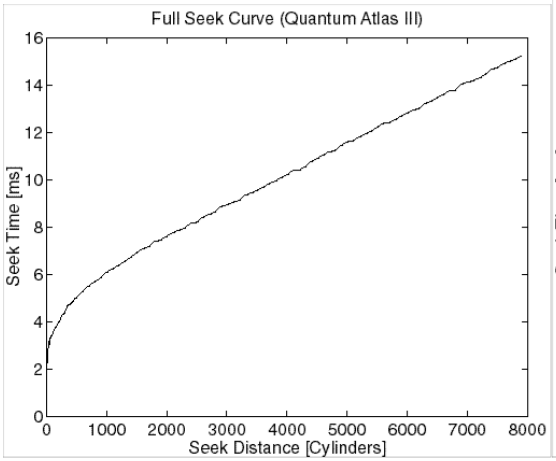
A. Internals

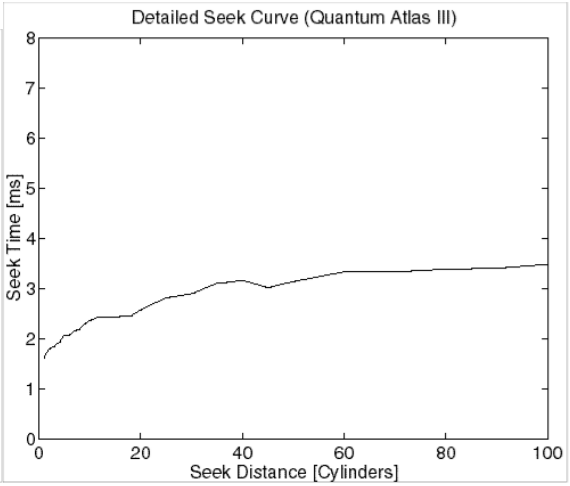


1. Sector: (512 bytes) smallest amount of IO you can do to an HDD
2. Tracks: 256-512K in size

B. Components of Access:

1. Command, Seek, Rotate, Transfer
2. Seek: accelerate, coast, settle (important to find that servo data)



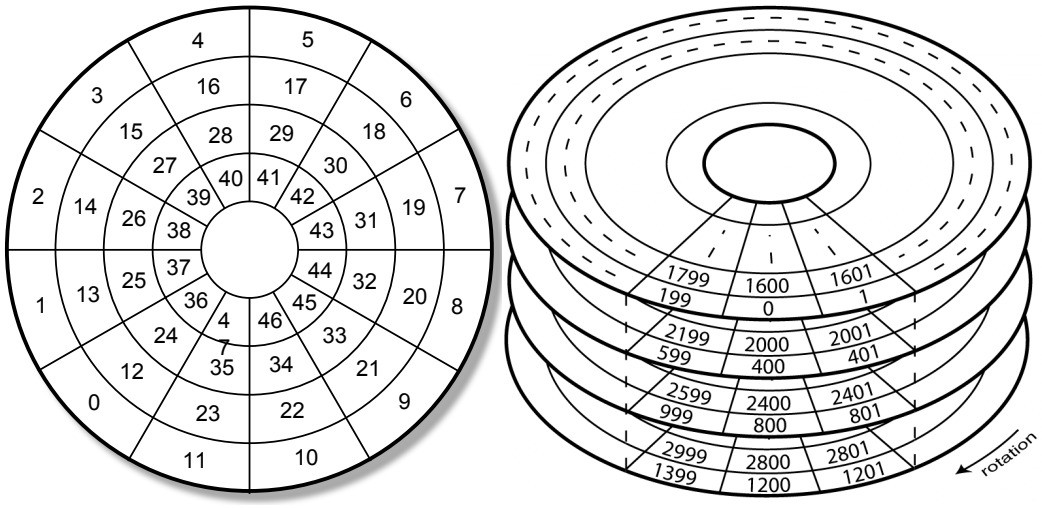


C. Rotate: a function of RPM; average rot. lat. 1/2 revolution; 7200RPM = 60(s/m)/7200(r/m) = 8.33ms; avg = 4.16ms

D. Transfer:

1. simple case (all data in one track): sectors desired * time for revolution / sectors per track
2. complex case (mult. tracks): above + ? (seeks, track & cyl. switches)

E. A Simple Organization



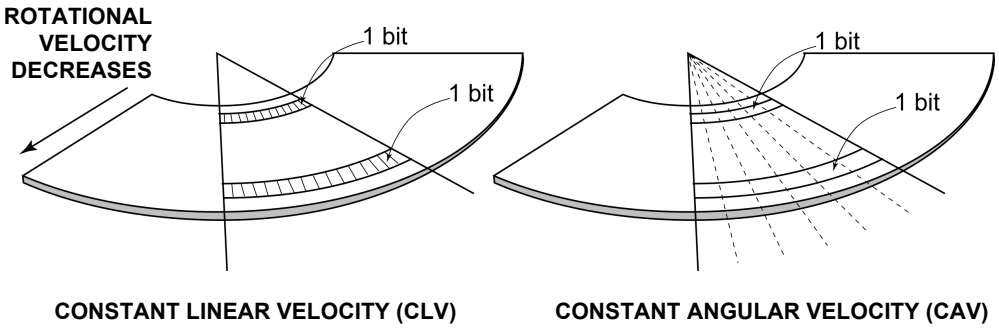
1. What's presented by the disk drive: a linear array of sectors
   a) This is a logical abstraction
2. Drive must map logical blocks to physical blocks; e.g. Let's

do some math:

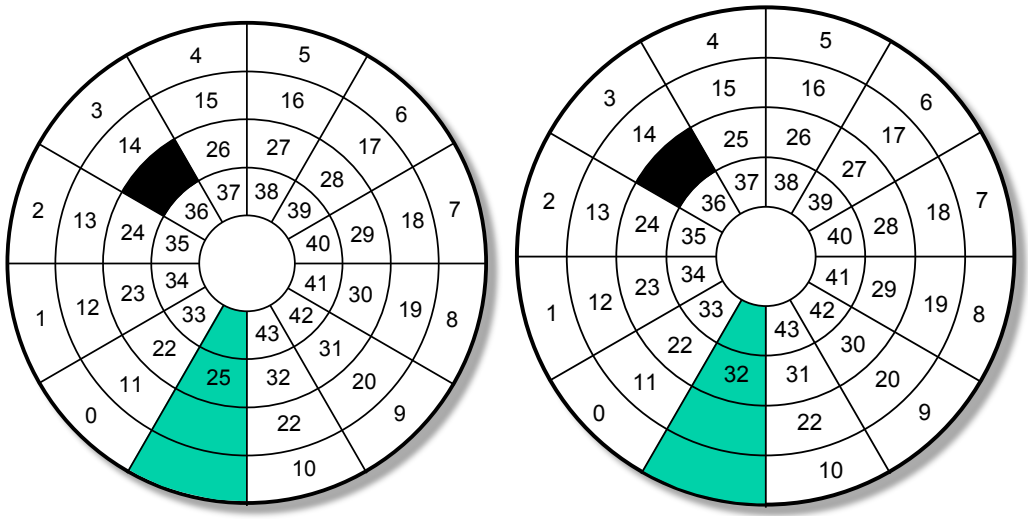
a) How would we compute an LBN->PBN mapping?
   (1) Cyl# = lbn/sectspercyl
   (2) Surface# = (lbn % secspercyl) / sectspertrack
   (3) sect# = lbn % sectspertrack

F. Complications

1. **Zones**: outer tracks store more sectors; complicates logic/bookkeeping; increases signal processing logic
   a) LBN -> PBN gets a little more complicated, but not unreasonable

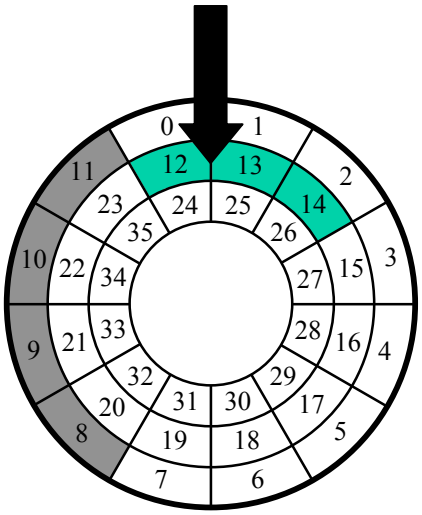


2. **Defects**: over time, portions of media become unusable (e.g. fail to hold charge, exhibit many errors);
   a) additional physical space for "Spares" (both tracks and sectors)
   b) Solutions: Remapping & Slipping

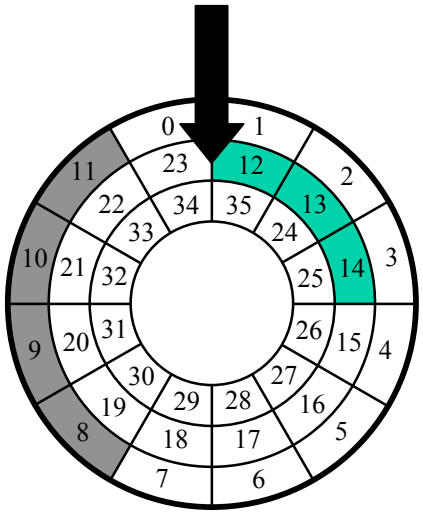


   c) Encoding: Error Correct Codes (ECC) & Run-Length Limited Encoding

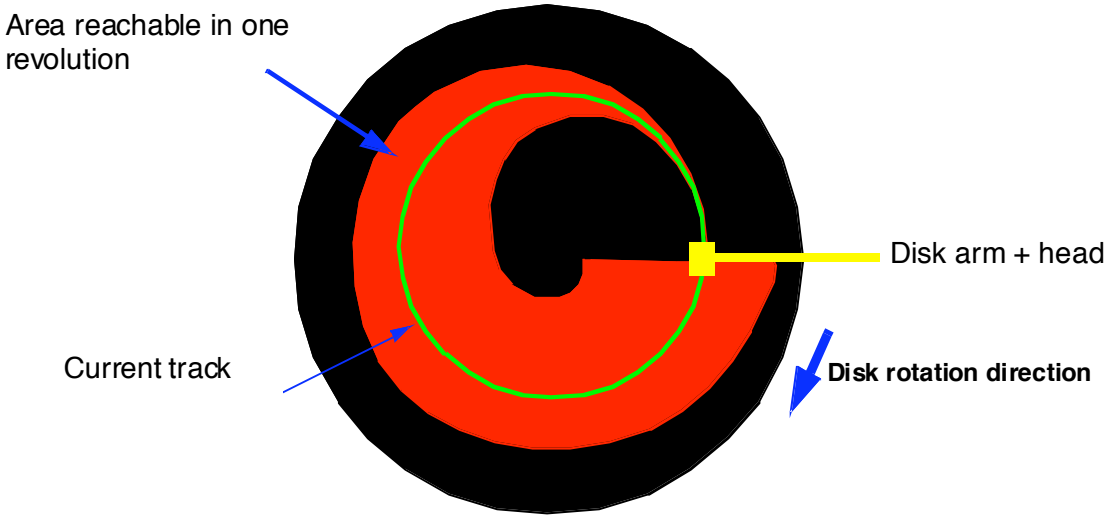
3. **Skew**: Cross-track & cylinder access: when data cross these boundaries, delay in head movement/switching can results in a "miss" and additional delay.
   a) Solution: LBN->PBN must take into account physical characteristics of the drive



After track switch



After track switch



4. **Users**: IO Requests are stochastic, and are rarely issued in an optimal order
   a) Solution: Disk scheduling algorithms (more later)

IV. File Systems

A. What we ultimately want from our file system:

1. To access, organize, and persist data
2. Provide protection and concurrency of data access
3. Implement optimizations that provide good performance

B. File systems provide an object abstraction and interface for data (yet another virtual layer)

1. Object: files, directories, links, metadata
2. Interfaces:lookup, read, write, rename, truncate, position, delete
3. Services: naming, object hierarchies, metadata, locking, allocation, layout, security and optimizations

V. Files

A. Files are the first-class, logical units of data with which processes interact

1. A file is more than the data that constitutes the file
2. What is a file's name?
3. What its type? What are its access permissions? Who owns it? When was it created? Or, last accessed?
4. Where does the data live on persistent storage device? How do we make updates to it?

B. Files are just an **abstraction**, and its the file system's job to provide it

1. shielding the user from having to directly manage all the above

C. File Extensions

1. Restrictions on how files are named are up to the OS
2. As is the enforce of extensions (e.g. .c, .exe, .txt)
   a) UNIX doesn't care: extensions are just hints for and by users
   b) Windows & OS X do care: extensions hint at the types of accesses and classes of processes allowed

D. File Structures

1. Most commodity OSes represent files as a logically contiguous array of bytes
   a) Very flexible from a user/process perspective
2. Some files are more heavily structured
   a) Databases only accept records
   b) Big Data file systems e.g. may only accept data blobs, or may requires a "document" (XML) structure

E. File Types

1. What type of data constitutes a file?
2. In UNIX: there a OS-supported file types
   a) Regular file (ASCII, binary)
      (1) e.g. plaintext, images, executables
   b) Directories (more on this later)
   c) Character special files (an abstraction of a character device)
      (1) e.g. /dev/tty, /dev/eth0
   d) Block special files (an abstraction of a block device)
      (1) e.g. /dev/sda

F. File Access

1. How do we want to access files?
2. Early OSes only supported **sequential access** (with rewind)
   a) This was sensible, because our IO devices (tape, punch cards) were also sequential
3. When disks were introduced, this enabled **random access**
   a) Enables a whole new class of applications and execution models (e.g. databases, virtual memory)

G. File Attributes

1. As mentioned, a file is more than just its data: all the attributes and other related information must be persistent as well
2. We call this **metadata** (data about data)
   a) Common examples: Access bits, creator, owner, size, pointers to blocks on disk,

H. File Operations

1. Create, Delete, Open, Close, Read, Write, Append, Truncate, Seek, GetATTR, SetATTR, Rename, Stat

VI. Directories

A. Single-Level Directory Systems

1. Only a root directory

B. Hierarchical Systems

C. Paths

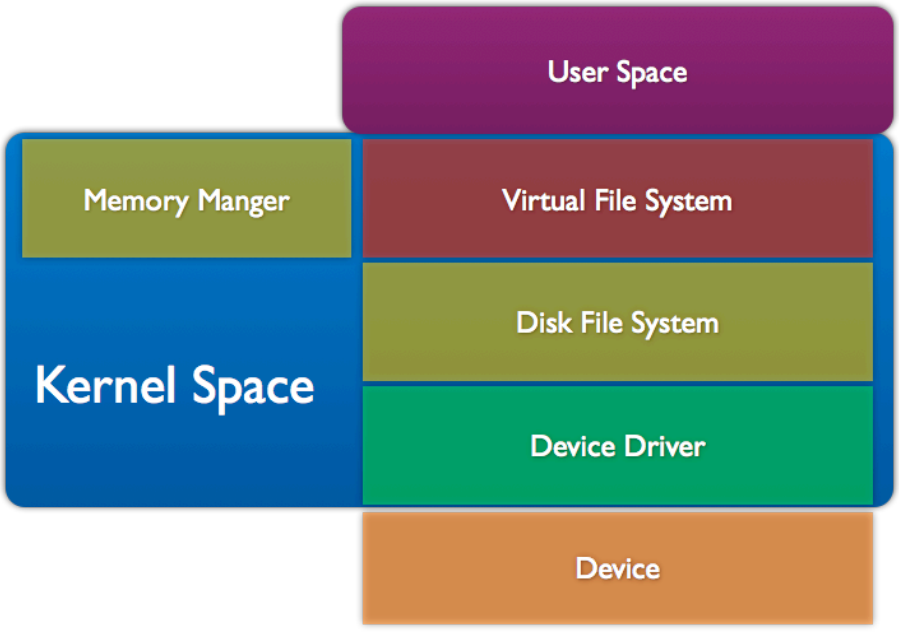
1. Absolute Paths
2. Relative Paths
   a) . and ..

D. Directory Operations

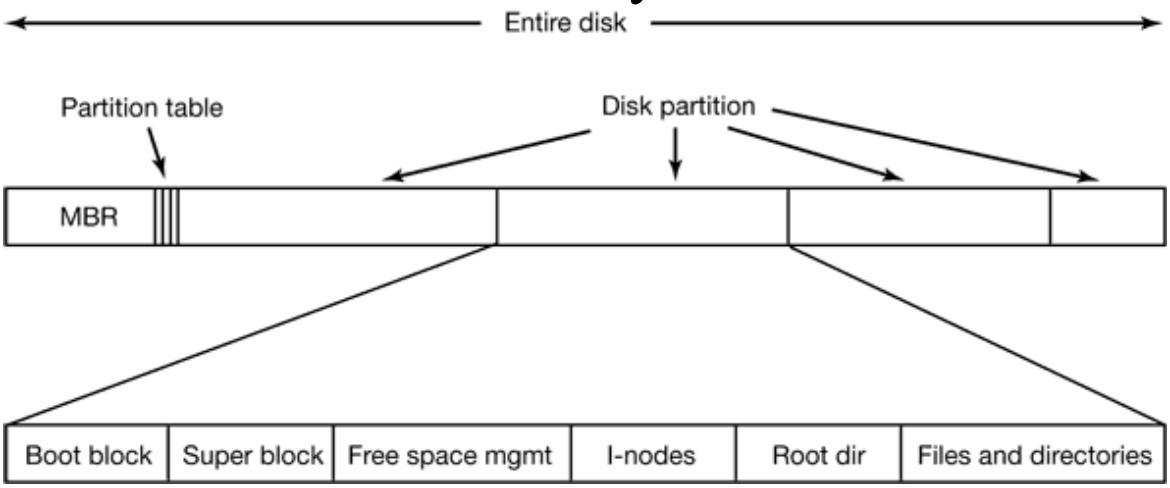
1. Create, Delete, Opendir, Closedir, Readdir, Rename, Link, Unlink
2. Hard links: Names points to a data structure
3. Soft links: Name points to another name

VII. File System Organization

A. The may layers between user space and the device



VIII.Common Disk File System Structures



A. **Master Boot Record** (MBR)
   1. Holds the information necessary to bootstrap the system's file systems
      a) Partition table
         (1) A logical mapping of the mountable file systems available on the device
      b) Machine code to necessary to read partition table and active partition
   2. By convention, MBR always at sector 0
   3. Read in by BIOS
      a) OS is typically stored in the **active** partitions
      b) First block in the active partition is the **boot block**
      c) This contains the machine code necessary to bootstrap the OS
      d) Every partition has a boot block, but only one is active

B. From here on, file system structures can vary greatly

C. Here we give some common examples and structures

D. Often, two major components: **metadata** blocks and **data** blocks
   1. Metadata contains information about the objects in the file system
   2. Data contains the actual information, stored by the user
   3. Remember: what follows are the **on-disk** structures; in-memory structures are handled by the VFS

E. Superblock

1. Metadata that contains file system-wide information, e.g.
   a) total number of files
   b) number of free/allocated data blocks
   c) file system type
   d) time and access information
   e) consistent or inconsistent
2. Since the superblock is so valuable, often file systems replicate it for redundancy

F. Inodes

1. Metadata that contains per-file information, e.g.
   a) Location of data blocks associated with this file
   b) Access Permissions or Control Lists
   c) Access, Creation, Modification times
   d) File sizes
   e) Reference count (how many hard links point to this inode)
   f) (most things available from stat())
   g) Does NOT contain the file's name
2. **Directories** are a special kind of inode, who data are <name, inode> pairs, called **directory entries** (dirents/dentry)
   a) Creating a name, is adding an entry
   b) Renaming an object, creates or replaces a name-inode mapping
   c) Deleting a name, removes or makes inactive the mapping from the parent directory
3. Typically directories store only the names of the direct children inodes
   a) Parent-child relationship
   b) By default, a directory contains . (self reference) and .. (parent)
4. Hard links are directory entries
5. Soft links are inodes whose data contains a name
6. Many names can point to the same inode
   a) This can result in cycles within our tree

IX. Lookup

A. One of the core operations of a file system, is the ability to find a file by its name: called a **lookup**

B. Directories provide a hierarchy of data

C. All file systems start at a logical /, called **root** (like the root of a tree)

D. The superblock tells the kernel where to find the inode for /

1. Almost always inode #2 (by convention)

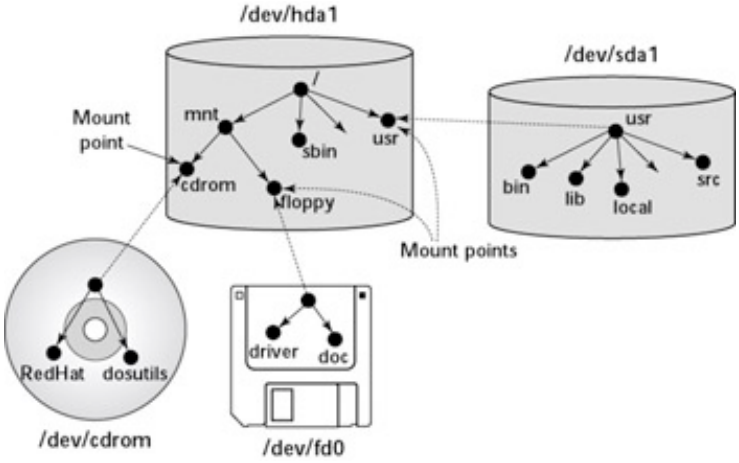
E. When I access a file, either by its relative or absolute path, I must do a lookup

1. A recursive traversal of the tree
2. e.g. the path /user/home/zachary/foo.txt
   a) Starts at /, lookups user directory inode, in user looks up home directory inode, etc.

F. Lookups are expensive (requires disk IO)

1. The file system often implements dirent/dentry cache, so that common paths can be satisfied from memory
2. Also, makes relative pathing more efficient
   a) i.e. a lookup of ./foo can be resolved using cache only
3. In Linux, the virtual file system maintains this cache

X. Mounting & The VFS



A. The virtual file system allows multiple file systems to be mounted at different places with the logical namespace
B. A file system's root directory can be mounted anywhere in the name space: call the mount point
C. As such, the VFS handles the interface to lookup
    1. When a file system is mounted, it registers itself with the VFS
        a) In Linux: function pointers associated with common IO operations: e.g. read, write, lookup
        b) VFS supports generic operations, if the disk file system doesn't register a call
D. VFS has in-memory objects for each of the on-disk structures
    1. superblock
    2. vnode for each open inode
    3. dentry object, for cached directory entries
E. These abstractions allow many file system implementations to coexist in the same namespace
    1. e.g. able to mount different devices or network file system protocols
    2. You could have a lookup operation that calls your friend on the phone

XI. File System Implementations

A. Remember the objects that need implementing
    1. Superblock (prob. w/ redundancy)
    2. Inodes
    3. Directories
    4. Data blocks
B. The state of those objects
    1. Clean/dirty
    2. Consistent/Inconsistent
    3. Allocated/Unallocated
C. Device limitations & requirements

D. Directory structures

E. Block indexing techniques

1. Contiguous
   a) All file blocks must be contiguous on disk
   b) Excellent sequential performance
   c) Remind you of anything? (Dynamic storage-allocation problem and external fragmentation)
2. Linked
   a) File blocks are linked together with pointers
   b) May be spread anywhere on disk
   c) Potential for terrible performance (non-sequential access = disk seeks)
   d) Solutions: increase the block size (and internal fragmentation), or improve allocation locality
   e) Potential for unreliability (A break in the chain, leads to a loss of data)
3. Indexed & Multi-level indices
   a) A metadata block used to point to the data blocks of the file
   b) Direct blocks, indirect blocks, doubly indirect blocks, and triply indirect blocks
   c) Small files have very good performance (and most files are small)
   d) Large files require additional I/O, but which may be aggregated for better performance

F. Free-Space Management

1. Bit-vector/map
   a) Each data block represented by a bit: 0 is free, 1 is allocated
   b) Simple to implement and manage
   c) Interesting algorithms used for finding co-located blocks
   d) Large disks mean large bit maps: inefficient to store an entire bitmap in memory
2. Linked List
   a) Blocks themselves contain pointers

b) Free blocks point to other free blocks
c) Blocks are allocated from the head of the list
d) Very inefficient and can lead to file fragmentation
e) Variants that group blocks together in the list exist, but aren't widely seen IRL

3. Space Maps
   a) Sun's ZFS was designed for large-scale computer: large file sizes, lots of files: metadata IO can be a significant cost

G. Consistency

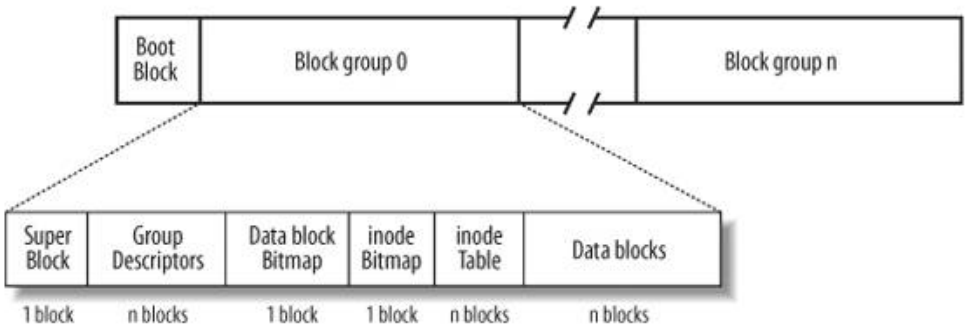
1. What happens if metadata is written but data is not (due to crash)?
2. Consider: appending a single block to a file
   a) Requires at least 3 IOs: data block, inodes, group bitmap
   b) just the data block written to disk; data exists, but no pointers to it, no way know if it's allocated; file system consistent, but data loss
   c) just the inode is written to disk; inode points to a data block, but no data, leading to garbage data; inconsistent state (disagreement between inode and bitmap)
   d) just the bitmap; again, inconsistent. left unresolved, leaking data
3. Solutions
4. FSCK: 6-pass scan of the entire file system
   a) Pass 1: scan superblock and all inodes
      (1) for each inode, ensure metadata makes sense (valid permissions, sensible/valid block numbers)
      (2) check if two inodes point to the same data blocks
   b) Pass 2 & 3: check directories
      (1) Do . and .. create a cycle-less namespace
      (2) Orphaned directories are placed in lost+found
   c) Pass 4: Check ref counts
      (1) Make sure all that ref counts are correct: do the number of names, inode pairs match to the inode count
   d) Pass 5: check bitmaps
      (1) Do the in-use bitmaps match the current inode, block state
   e) Pass 6: check sectors (optional)

(1) scrub sectors looking for bad ones

5. Journalling
   a) Idea: before overwriting data structures, create a note that describes the operation
   b) Journal mode: Both metadata and data are written to the journal; performance: everything is written twice
   c) Ordered mode: Only metadata is journaled; data are written before metadata are marked as committed; much faster than journal mode
6. File system design

XII. Example File Systems

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



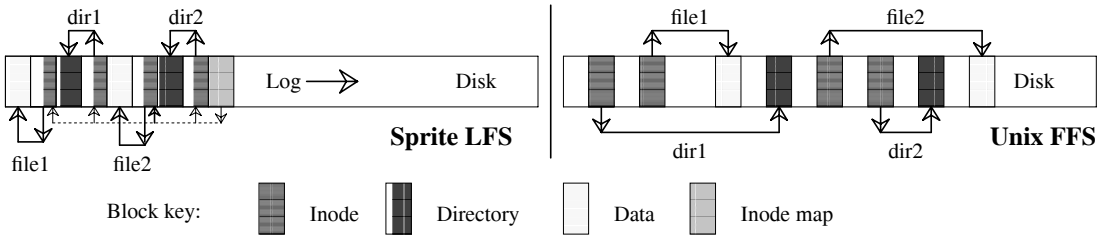
A. ext3 file system (ext2 w/ journalling)

1. Organized into **cylinder/block groups** (a la FFS)
   a) size of block group is constrained by block size (and thus, the number of bits in a bitmap)
2. Groups sized equal to one or more disk tracks
3. Each group contains
   a) a copy of the superblock (although not all are current)
   b) a group description (block group metadata)
      (1) block number of block allocation bitmap
      (2) block number of inode allocation bitmap
      (3) number of free blocks, inodes and directories in group
   c) space for inodes (128 bytes each)
   d) space for data blocks
4. ext3 inodes use a multi-level index
   a) 12 direct, 1 single indirect, 1 double indirect and 1 triple

indirect

5. ext3 attempts to allocated related data blocks near each other, near their inode (i.e. within a group): within a cylinder group, in near cylinder groups
   a) All blocks of a file can be accessed in one track access
6. Hard links/soft links
   a) Hard links are directory entries
   b) Soft links attempt to store path in direct blocks
7. Directories contain
   a) array of name-inode pairs
   b) inode number (4 bytes), record length (2 bytes), name length (1 byte), file type (1 bytes) and name (aligned to 4 bytes)
   c) when a file is removed, its inode is set to 0, and the record length of the previous entry's rec length is updated to extend over deleted entry
8. Requires **mkfs** to initialize device
9. Fixed number of inodes per group

B. Log-Structured file system



1. Data are written to a circular log; never overwriting data
2. metadata are not a fixed position; in-memory structures are created at mount time
3. file system state is occasionally checkpointed; fast recovery from crashes; always consistent
4. implicit versioning;
5. Requires garbage collection: how do we know when data are current? Scans the disks, looking for data that expired, also can move data up to make contiguous
6. On HDD: Excellent write performance, poor read performance (data are non-sequential)

7. SSD might resurrect this idea

XIII.Other Designs & Features

A. Versioning & COW: ZFS & WAFL & ext3cow

1. As opposed to continuous versioning in LFS, file system can take snapshots
2. Snapshots provide for point-in-time logical views of the entire (or portions) of the file system
3. Often leverage copy-on-write

B. Blocks vs. extents

1. Blocks align with virtual memory systems
2. Extents

C. Content-based addressing

1. Data are addressed by their content, rather than logical address
2. Provides for **de-duplication**: only one copy of a data object are stored; objects may be shared between users
3. Implementation decision:
   a) Size of object to be hashed (blocks or entire objects)
   b) Hierarchical hashing (e.g. merkle tree)

D. Read-ahead: Given a disk access, read the entire track or near tracks into buffers in anticipation of future requests

E. File Prediction/Clustering: Read-ahead, but for file system objects

1. Simple models tend to work best

XIV.How IO in Linux Works

A. A user modifies a single block of an existing file

1. User makes a system call to write()
2. The VFS receives the call, a kernel buffer is allocated, and user data is copied in. The VFS identifies which partition/disk file system the file belongs
3. The disk file system is called, passing the “file” object and the kernel buffer. Identifies the inode associate with the file, grabs the inode semaphore, and updates inode data. Finds the

memory pages needing updating as well as the identity of the backing device, then calls the VFS

4. Which contacts the memory manager (which manages a unified buffer cache)
5. Pages are marked direct
6. User is informed write has completed
7. At some point in the future, dirty pages are flushed to disk (the disk file system may again be called, but many just defer to the memory manager), which passes the page to the device driver
8. Device driver may re-order requests, and eventually pass them to device
9. Device may cache, re-order, and write to media; reporting success back to the device driver, where the pages are marked clean