Chapter 13-14-15: File-System Interface & implementation

- File Concept
- Access Methods
- Disk and Directory Structure
- Protection
- FS implementation
- Allocation
- Crash Recovery
- Other issues and systems
  - NFS,
  - WAFL
  - etc.

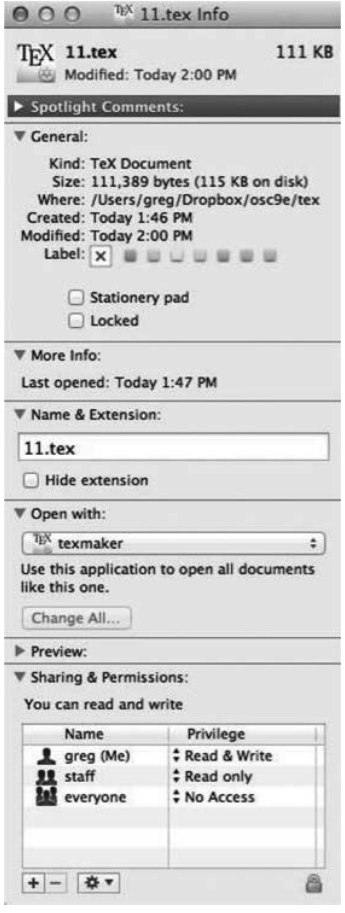
skipped slides are not included in the exam(atlanan slidelar sinava dahil degil)

File Concept

- Contiguous logical address space
- Types:
  - Data
    - Numeric
    - Character
    - Binary
  - Program
- Contents defined by file's creator
  - Many types
    - **text file,**
    - **source file,**
    - **executable file**

File Attributes

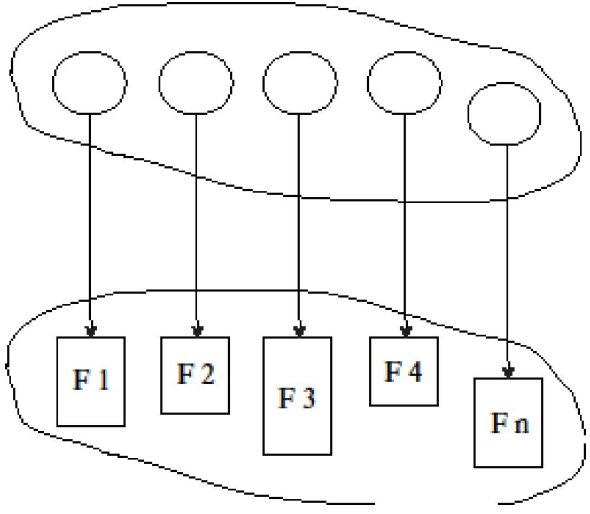
- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk
- Many variations, including extended file attributes such as file checksum
- Information kept in the directory structure



File info Window on Mac OS X

Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

File Operations

- **Create**
- **Write –** at **write pointer** location
- **Read –** at **read pointer** location
- **Reposition within file - seek**
- **Delete**
- **Truncate**
- ***Open*** **(*F_i*)** – search the directory structure on disk for entry ***F_i***, and move the content of entry to memory
- ***Close*** **(*F_i*)** – move the content of entry ***F_i*** in memory to directory structure on disk

Open Files

- Several pieces of data are needed to manage open files:
  - **Open-file table**: tracks open files
  - File pointer: pointer to last read/write location, per process that has the file open
  - **File-open count**: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

File Locking

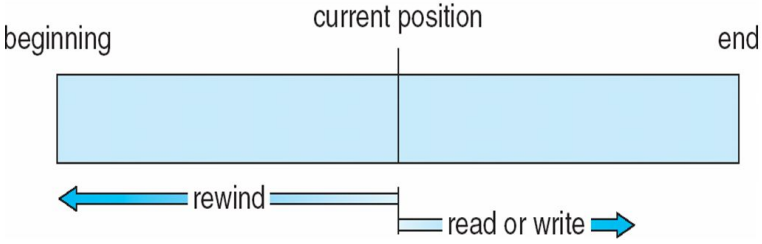
- Provided by some operating systems and file systems
  - Similar to reader-writer locks
  - **Shared lock** similar to reader lock – several processes can acquire concurrently
  - **Exclusive lock** similar to writer lock
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

Access Methods

- A file is fixed length **logical records**
- **Sequential Access**
- **Direct Access**
- **Other Access Methods**

Sequential Access

- Operations
  - **`read next`**
  - **`write next`**
  - **`Reset`**
  - no read after last write (rewrite)



Direct Access

- Operations
  - **`read n`**
  - **`write n`**
  - **`position to n`**
    - **`read next`**
    - **`write next`**
    - **`rewrite n`**

  n = **relative block number**

- Relative block numbers allow OS to decide where file should be placed

Simulation of Sequential Access on Direct-access File

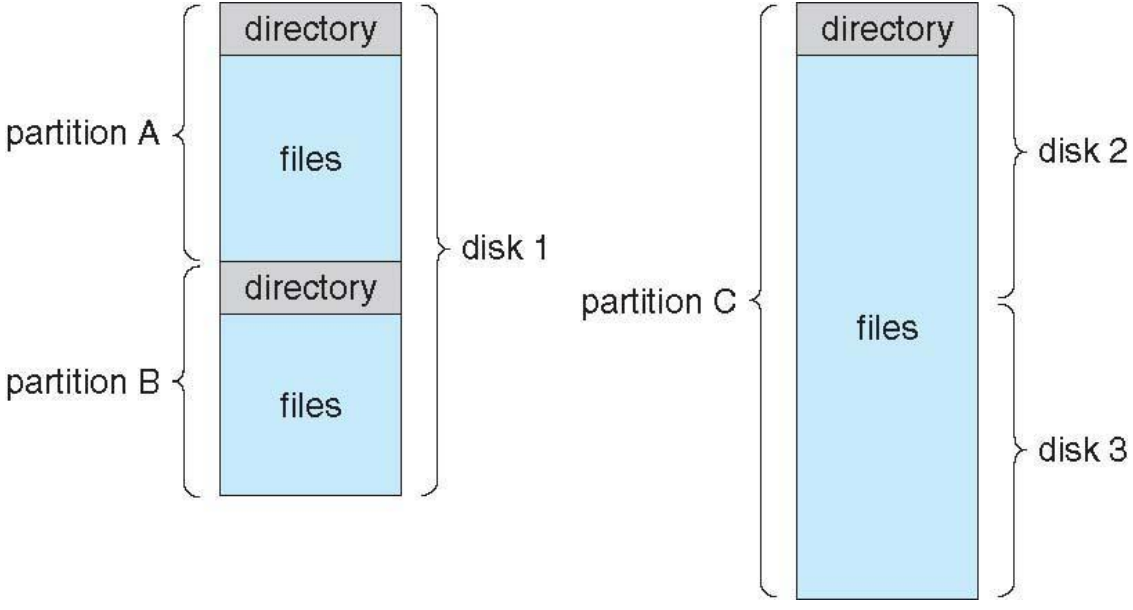
sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Disk Structure

- Disk can be subdivided into **partitions**
- Disks or partitions can be **RAID** protected against failure
- Disk or partition can be used **raw** – without a file system, or **formatted** with a file system
- Partitions also known as minidisks, slices

- Entity containing file system is known as a **volume**
- Each volume containing a file system also tracks that file system's info in **device directory** or **volume table of contents**
- In addition to **general-purpose file systems** there are many **special-purpose file systems**, frequently all within the same operating system or computer

A Typical File-system Organization

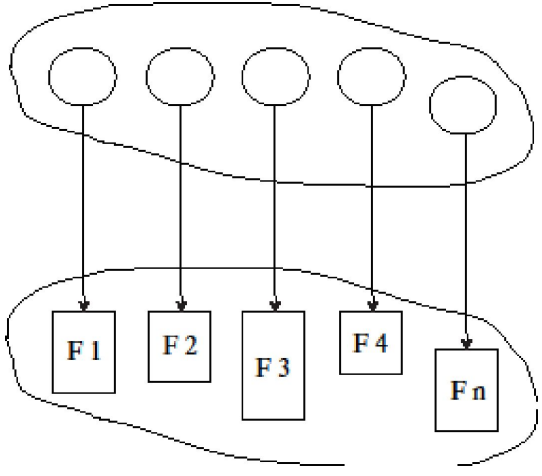


Types of File Systems

- We mostly talk of general-purpose file systems
- But systems frequently have may file systems, some general- and some special- purpose
- Consider Solaris has
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems

Directory Structure

- A collection of nodes containing information about all files



- Both the directory structure and the files reside on disk

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

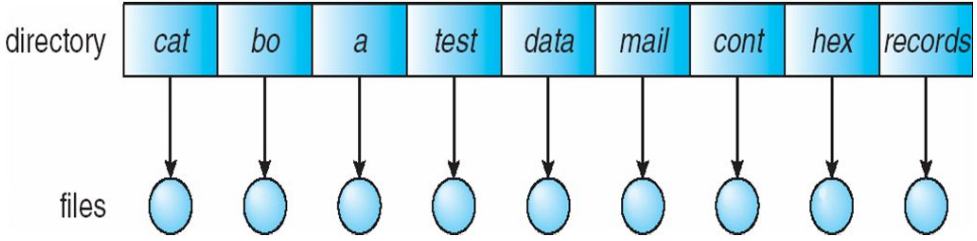
Directory Organization

The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)

Single-Level Directory

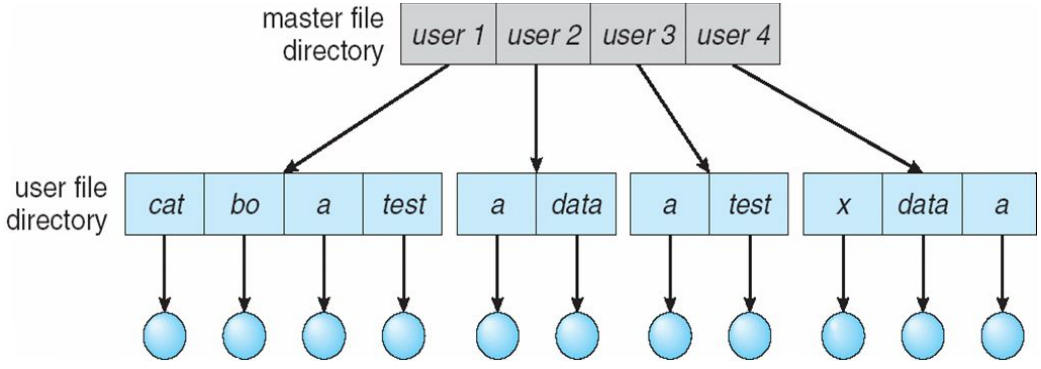
- A single directory for all users



- Naming problem
- Grouping problem

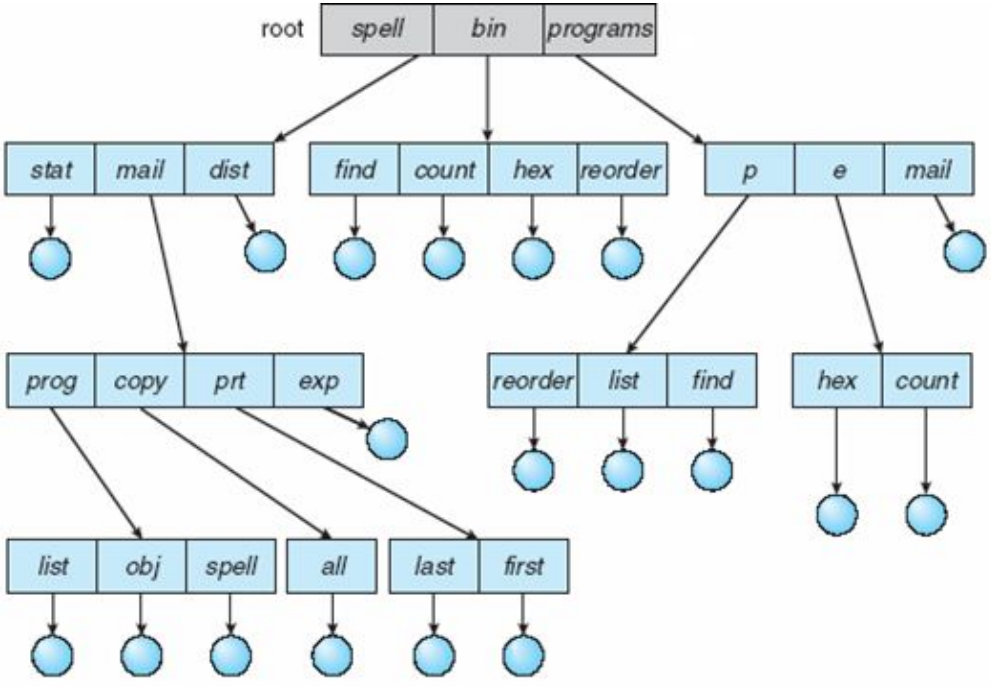
Two-Level Directory

- Separate directory for each user



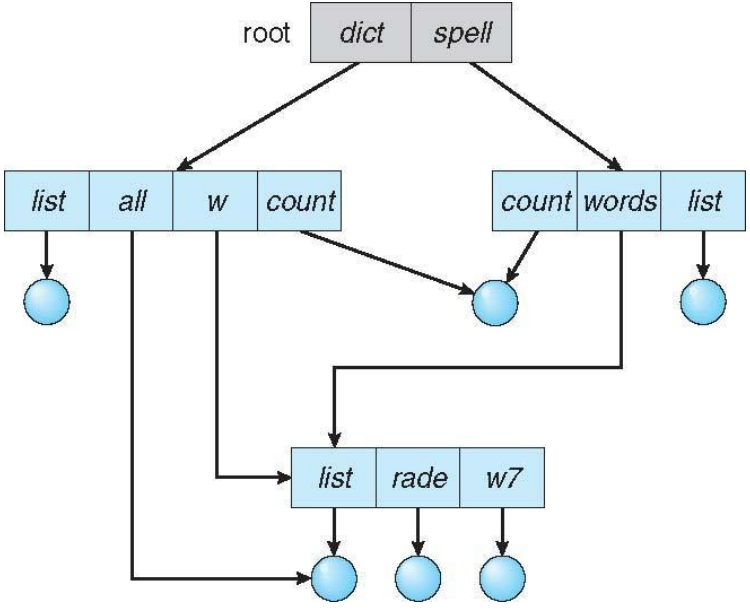
- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories



Acyclic-Graph Directories

- Have shared subdirectories and files
- Example

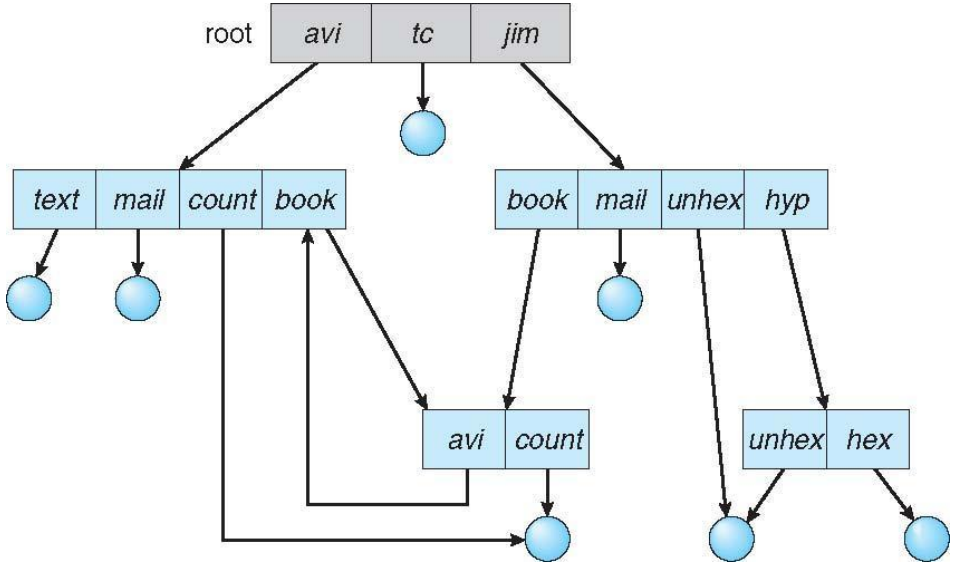


Acyclic-Graph Directories (Cont.)

- Two different names (aliasing)
- If ***dict*** deletes **w/*list*** ⇒ dangling pointer

  Solutions:
  - Backpointers, so we can delete all pointers.
    - Variable size records a problem
  - Backpointers using a daisy chain organization
  - Entry-hold-count solution
- New directory entry type
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

General Graph Directory

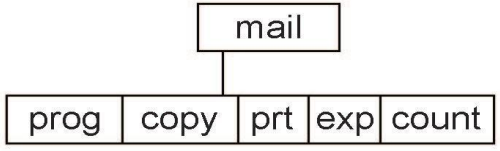


General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to files not subdirectories
  - **Garbage collection**
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK

Current Directory

- Can designate one of the directories as the current (working) directory
  - **`cd /spell/mail/prog`**
  - **`type list`**
- Creating and deleting a file is done in current directory
- Example of creating a new file
  - If in current directory is **`/mail`**
  - The command
    mkdir <dir-name>
  - Results in:



  - Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”

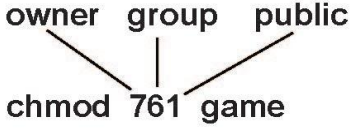
Protection

- File owner/creator should be able to control:
  - What can be done
  - By whom
- Types of access
  - **Read**
  - **Write**
  - **Execute**
  - **Append**
  - **Delete**
  - **List**

Access Lists and Groups in Unix

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

	RWX		
a) **owner access**	7	⇒	1 1 1
	RWX		
b) **group access**	6	⇒	1 1 0
	RWX		
c) **public access**	1	⇒	0 0 1



  - Ask manager to create a group (unique name), say G, and add some users to the group.
  - For a file (say *game*) or subdirectory, define an appropriate access.

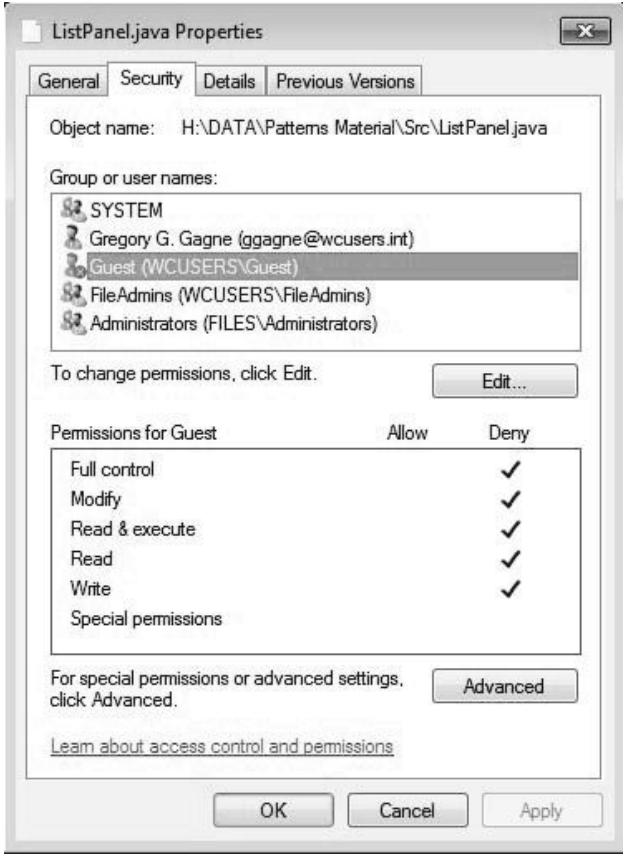
```
chgrp      G      game
```

  - Attach a group to a file

A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Windows 7 Access-Control List Management

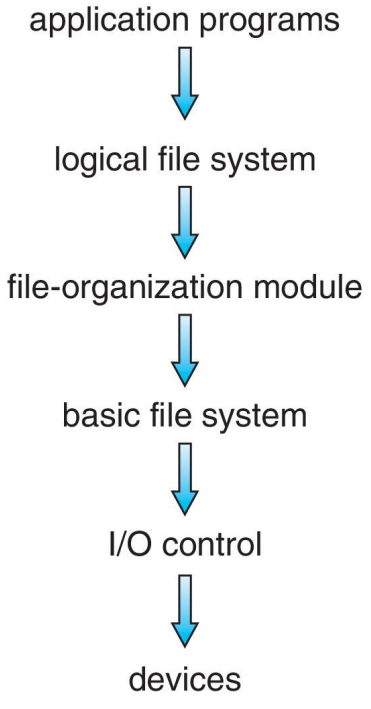


Chapter 14: File System Implementation

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System

File-System Structure

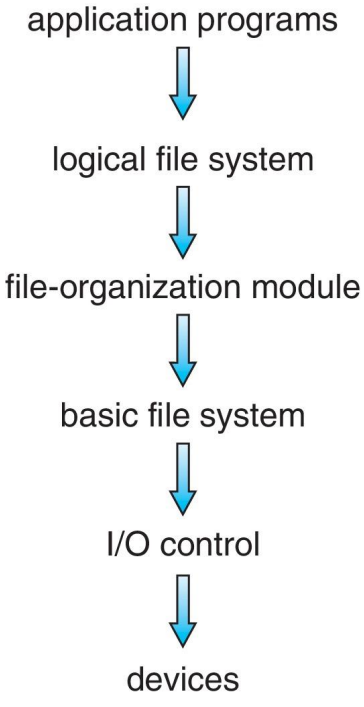
- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** (**FCB**) – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers



Layered File System

File System Layers

- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like
    - read drive1, cylinder 72, track 2, sector 10, into memory location 1060
  - Outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like “retrieve block 123” translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation



Layered File System

File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

File System Layers (Cont.)

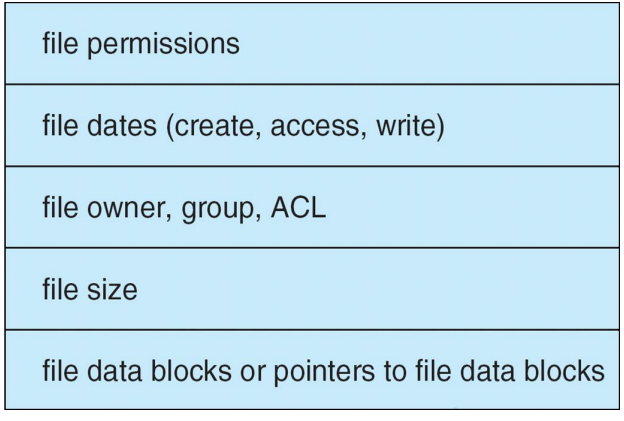
- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Operations

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table

File Control Block (FCB)

- OS maintains **FCB** per file, which contains many details about the file
  - Typically, inode number, permissions, size, dates
  - Example

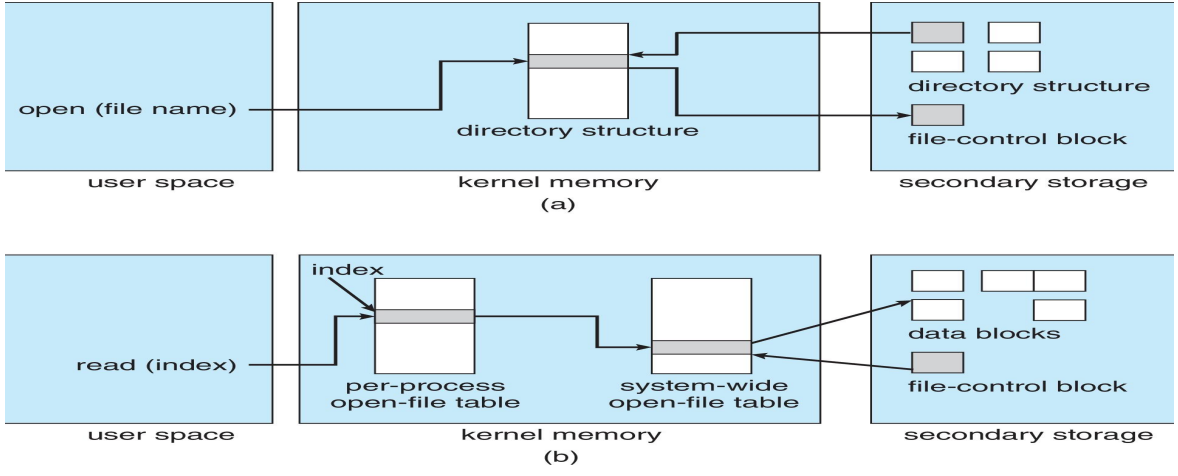


In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info

In-Memory File System Structures (Cont.)

- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file



Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via
      - linked list
      - or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

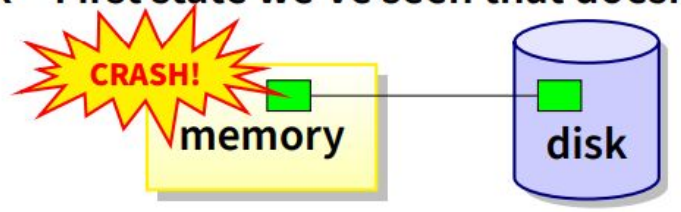
Main tasks of file system

- ➔ Associate bytes with name (files)
- ➔ Associate names with each other (directories)
- ➔ Don’t go away (ever)
- ➔ 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

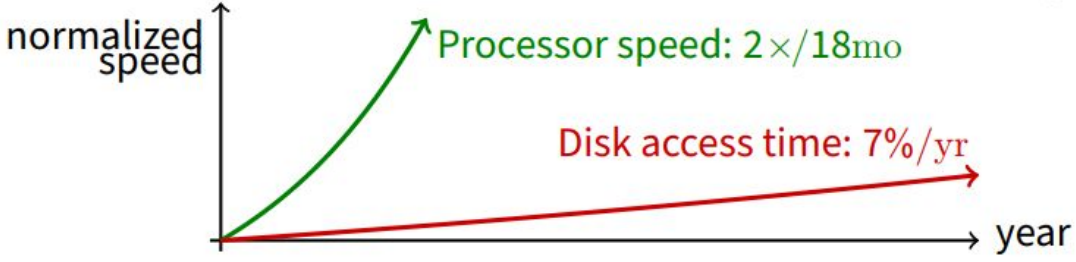
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Why disks are different

- **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)**



- **Huge (64–1,000x bigger than memory)**
  - How to organize large collection of ad hoc information?
  - File System: Hierarchical directories, Metadata, Search

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Disk vs Memory

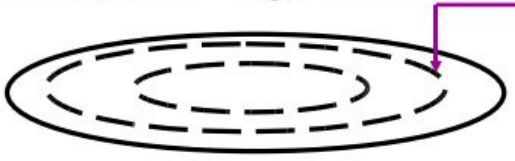
	Disk	TLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	200 MB/s	550–2500 MB/s	$>$ 10 GB/s
Sequential write	200 MB/s	520–1500 MB/s*	$>$ 10 GB/s
Cost	$0.01–0.02/GB	$0.06–0.10/GB	$2.50–5/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*Flash write performance degrades over time

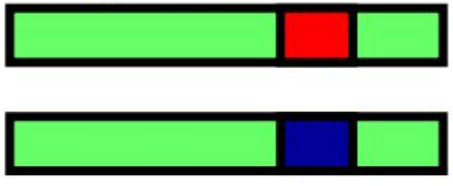
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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**

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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

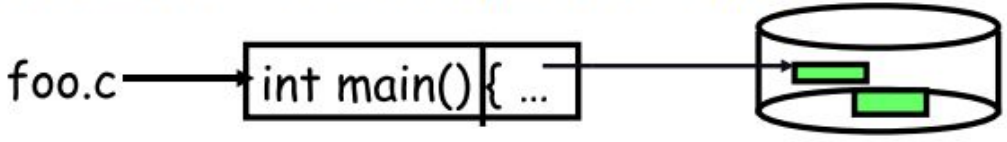
- Complex prefetching to support more IO patterns

- Delay data placement decisions to reduce random IO

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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
- **Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)**

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 #

  - File metadata: map byte offset to disk block address

  - Directory: map name to disk address or file #

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 $(0 \ldots \text{filesize} - 1)$, ~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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 *billions* of ALU ops in same time!
- **Access cost dominated by movement, not transfer:**

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

  - 1 sector: 5ms + 4ms + 5μs ($\approx 512\,\mathrm{B}/(100\,\mathrm{MB/s})$) $\approx$ 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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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?)

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

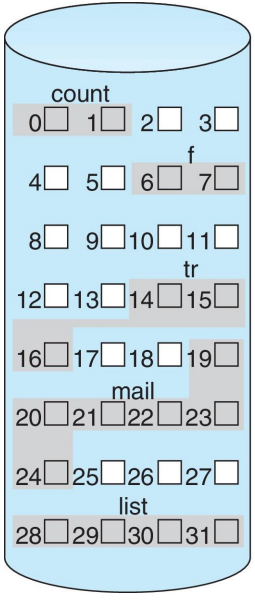
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - File Allocation Table (FAT)

Contiguous Allocation Method

- An allocation method refers to how disk blocks are allocated for files:
- Each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include:
    - Finding space on the disk for a file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

Contiguous Allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

APFS (Apple), HFS plus(Apple), NTFS(windows), ext4 (Linux), etc.

In indirect/direct block addressing, logical and physical blocks are mapped one-to-one,

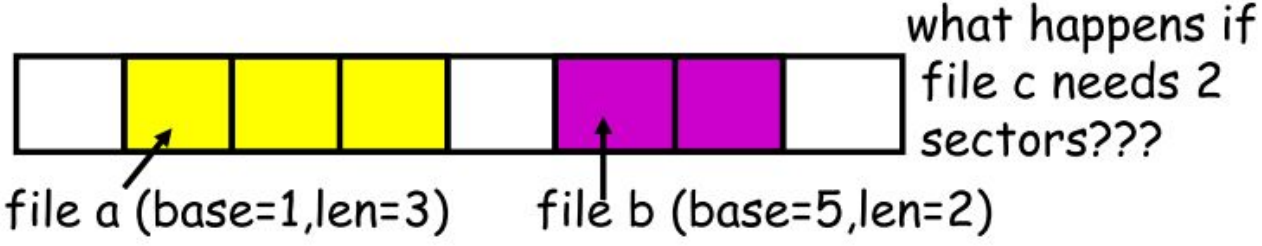
in extent-based mapping, a range of logical blocks are mapped to a range of physical blocks using a single extent structure.

```
Example File:

Extent    Logical Blocks    Physical Blocks    Length

  0            0 - 3           104 - 107          4

  1              4                112              1


struct ext4_extent {
    __le32  ee_block = 0;
    __le16  ee_len = 4;
    __le16  ee_start_hi = 0;
    __le32  ee_start_lo = 104;
};
```

https://blogs.oracle.com/linux/post/extents-and-extent-allocation-in-ext4

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

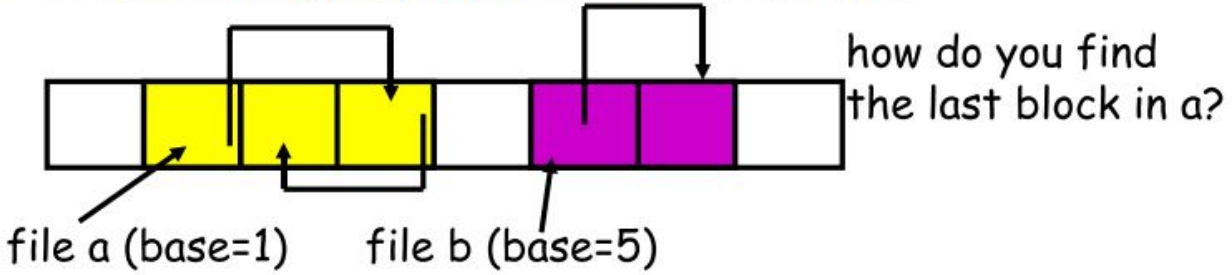


- **Example: IBM OS/360**
- **Pros?**
  - Simple, fast access, both sequential and random
- **Cons? (Think of corresponding VM scheme)**
  - External fragmentation

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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

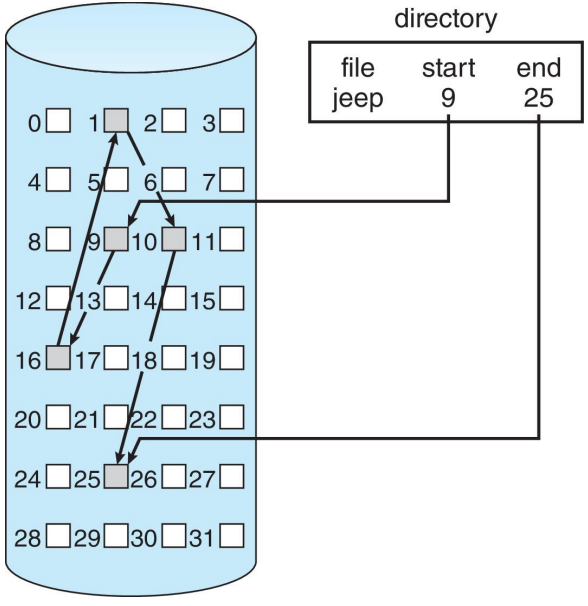


- **Examples (sort-of): Alto, TOPS-10, DOS FAT**
- **Pros?**
  - Easy dynamic growth & sequential access, no fragmentation
- **Cons?**
  - Linked lists on disk a bad idea because of access times
  - Random very slow (e.g., traverse whole file to find last block)
  - Pointers take up room in block, skewing alignment

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

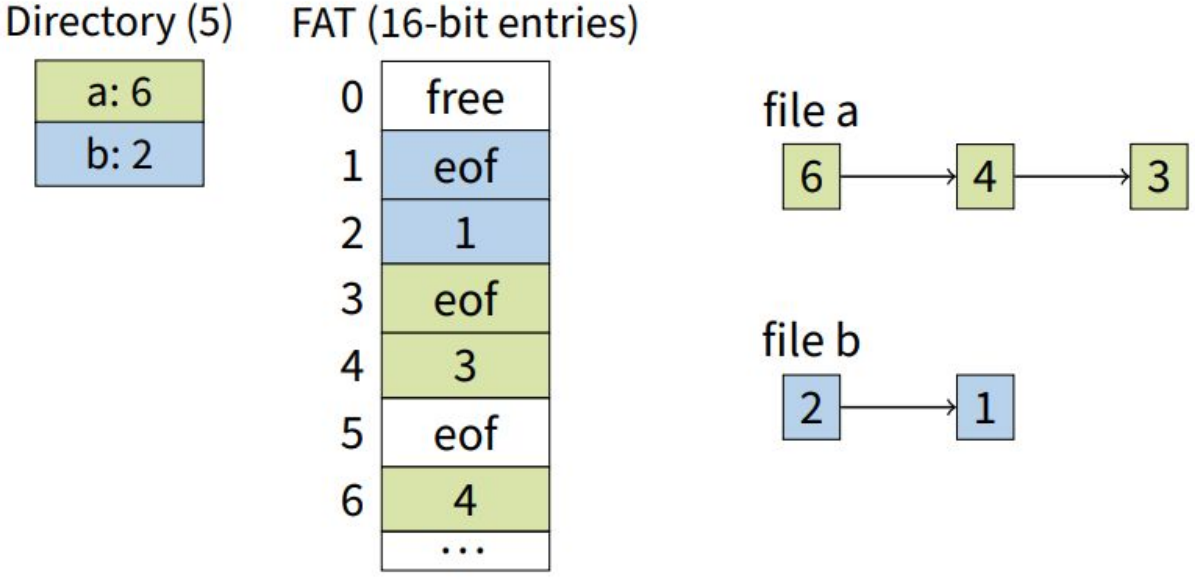
Linked Allocation Example

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme



Example: DOS FS (simplified)

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

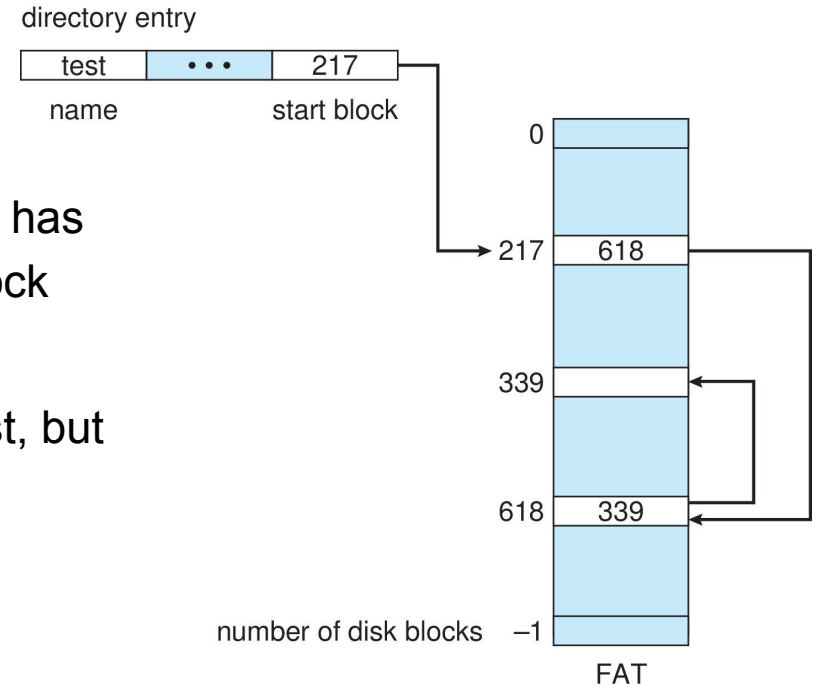


- **Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access**

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

File-Allocation Table

- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple



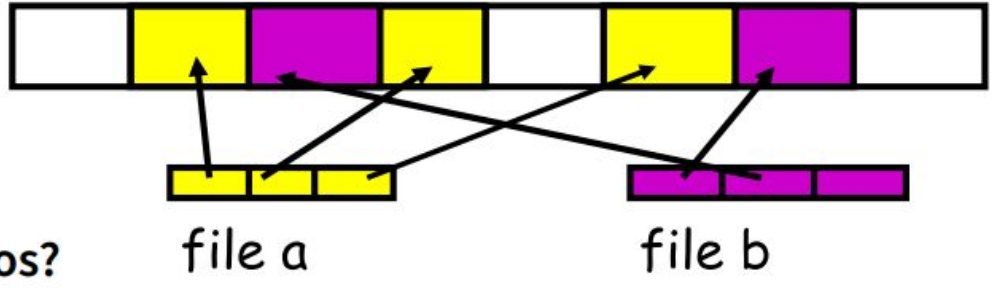
FAT discussion

- **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:

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Approach #3: Indexed Allocation Method

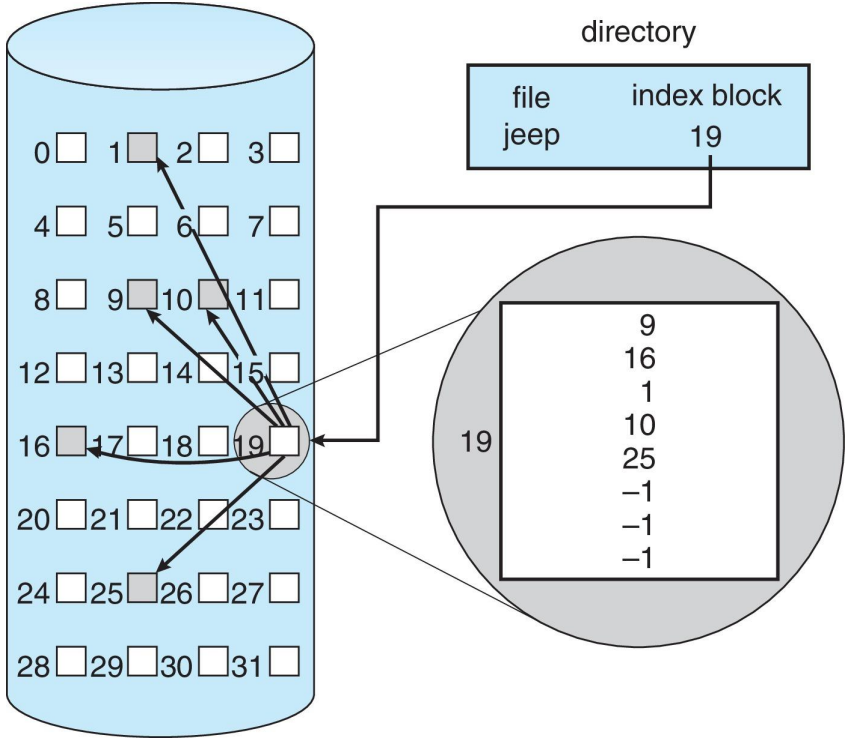
- **Each file has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



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

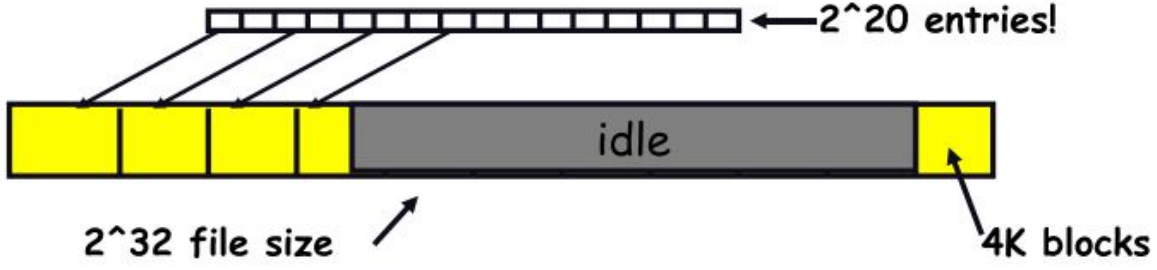
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Example of Indexed Allocation



Indexed files

- **Issues same as in page tables**



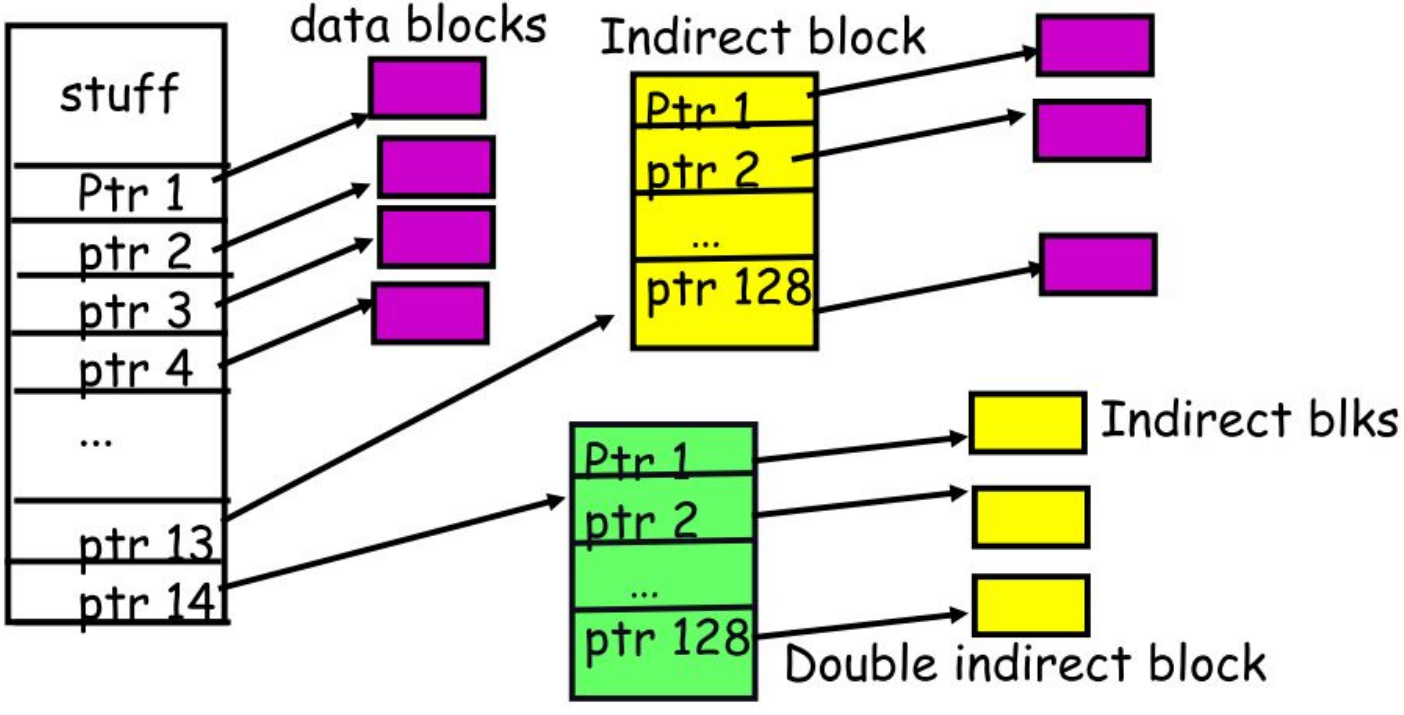
  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk

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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Multi-level indexed files (old BSD FS,UNIX UFS)

- **Solve problem of first block access slow**
- **inode = 14 block pointers + "stuff"**

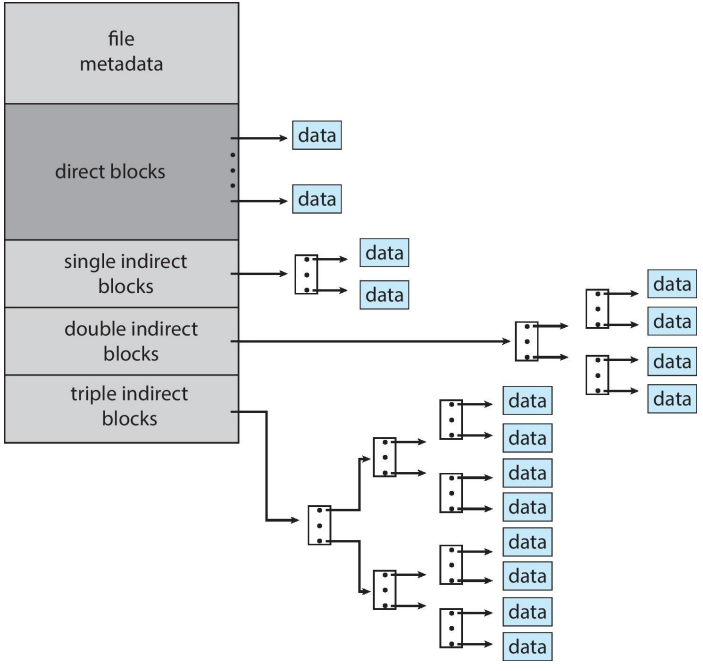


https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

UNIX UFS

- 4K bytes per block, 32-bit addresses

- More index blocks than can be addressed with 32-bit file pointer



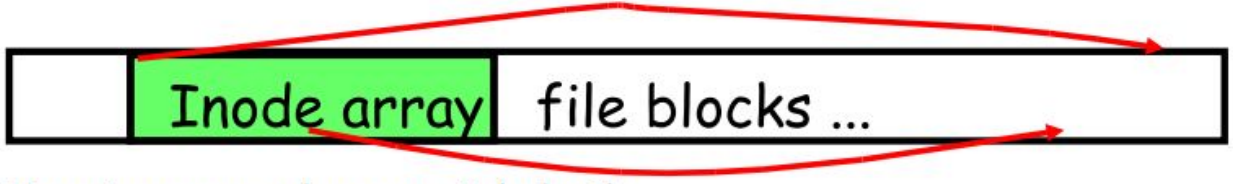
Old BSD FS discussion

- **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 freelist, metadata and data get strewn across disk

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Directories

- **Problem:**
  - "Spend all day generating data, come back the next morning, want to use it." – F. Corbató, 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?**

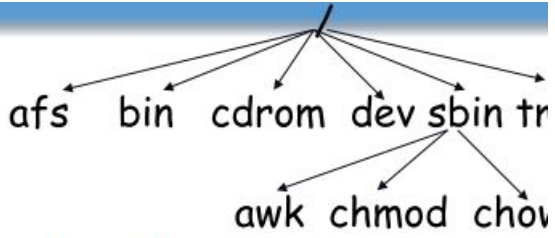
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 `ls` 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.)

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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
  - Users can read just like any other file (historically)
  - 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?)

<name, inode#>
<afs, 1021>
<tmp, 1020>
<bin, 1022>
<cdrom, 4123>
<dev, 1001>
<sbin, 1011>
⋮

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

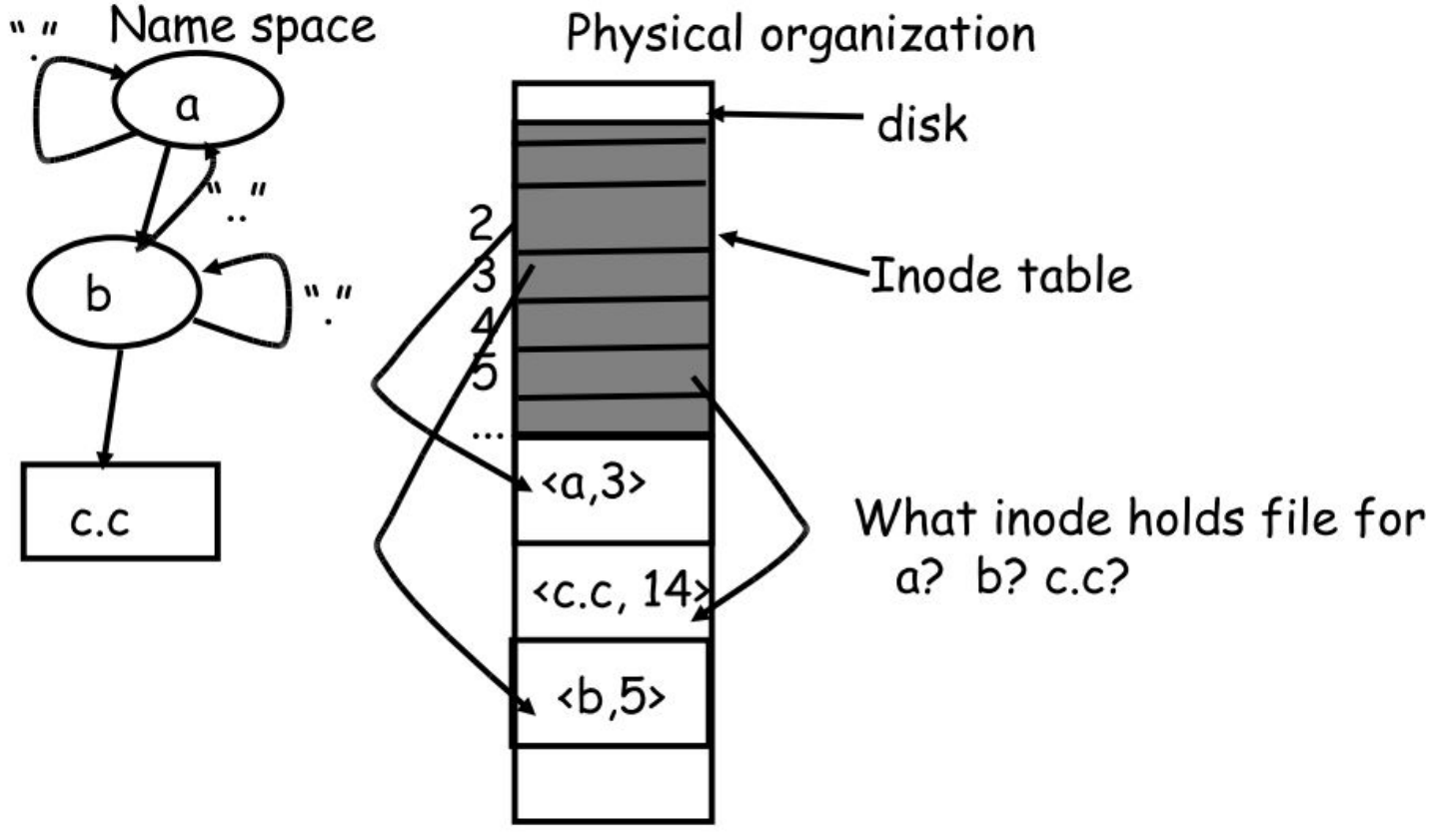
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Naming magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)
- **Special names:**
  - Root directory: "`/`" (fixed by kernel—e.g., inode 2)
  - Current directory: "`.`" (actual directory entry on disk)
  - Parent directory: "`..`" (actual directory entry on disk)
- **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)

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Unix example: /a/b/c.c



https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

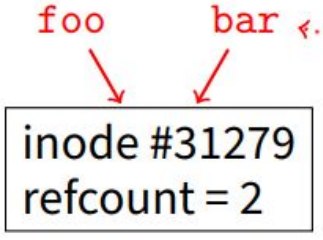
Default context: working directory

- **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**

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

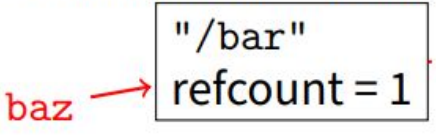
Hard and soft links (synonyms)

- **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

```
ln -s /bar baz
```

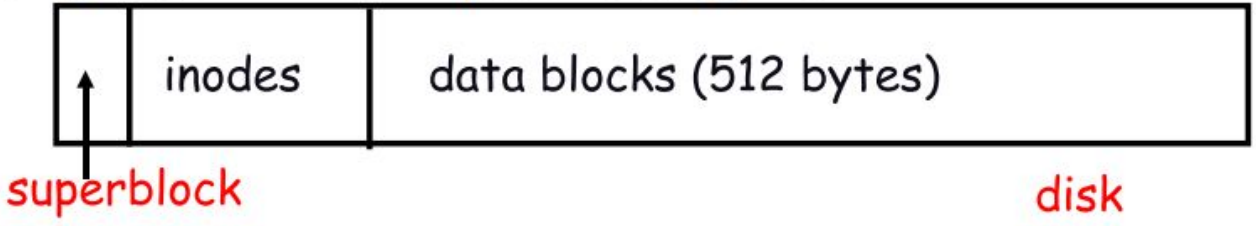


  - When the file system encounters a symbolic link it automatically translates it (if possible).

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Case study: speeding up FS

- **Original Unix FS: Simple and elegant:**

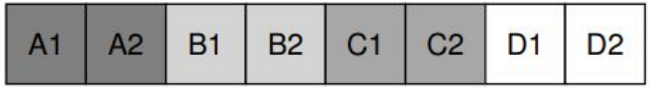


- **Components:**
  - Data blocks
  - Inodes (directories represented as files)
  - Hard links
  - Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
- **Problem: slow**
  - Only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!

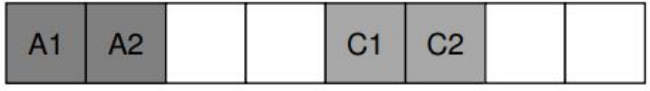
https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Example

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

side note: this problem is exactly what disk defragmentation tools help with..

- ★ Smaller blocks were good because they minimized internal fragmentation (waste within the block),
- ★ but bad for transfer as each block might require a positioning overhead to reach it

How do we make the file system "disk aware"?

https://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf

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 files in same directory not close together
  - Poor enumeration performance: e.g., "`ls -l`", "`grep foo *.c`"
- **Usability problems**
  - 14-character file names a pain
  - Can't atomically update file in crash-proof way

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Problem: Internal fragmentation

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

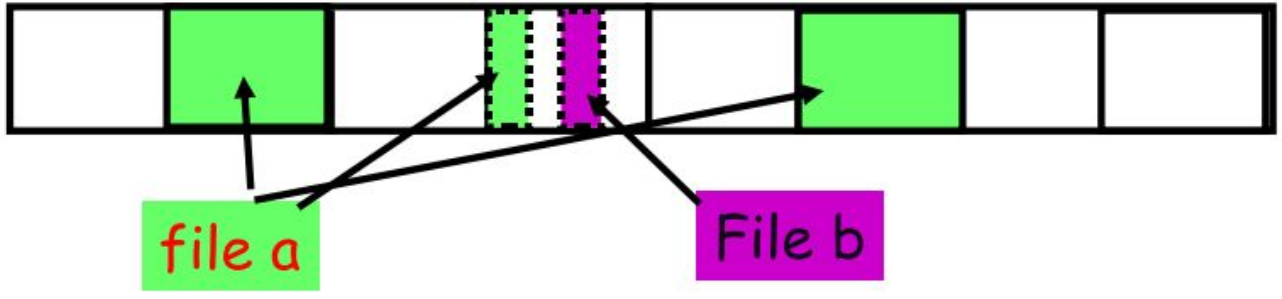
Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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

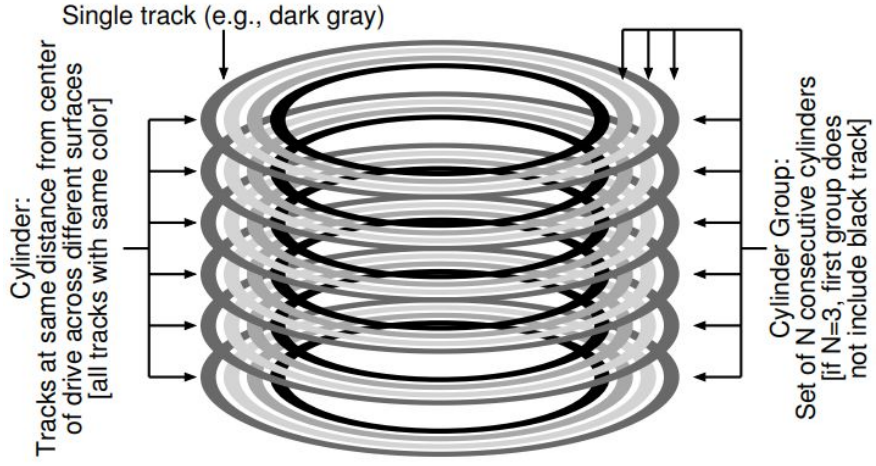


- **Best way to eliminate internal fragmentation?**
  - Variable sized splits of course
  - Why does FFS use fixed-sized fragments (1024, 2048)?

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

BSD FFS (Fast File System)

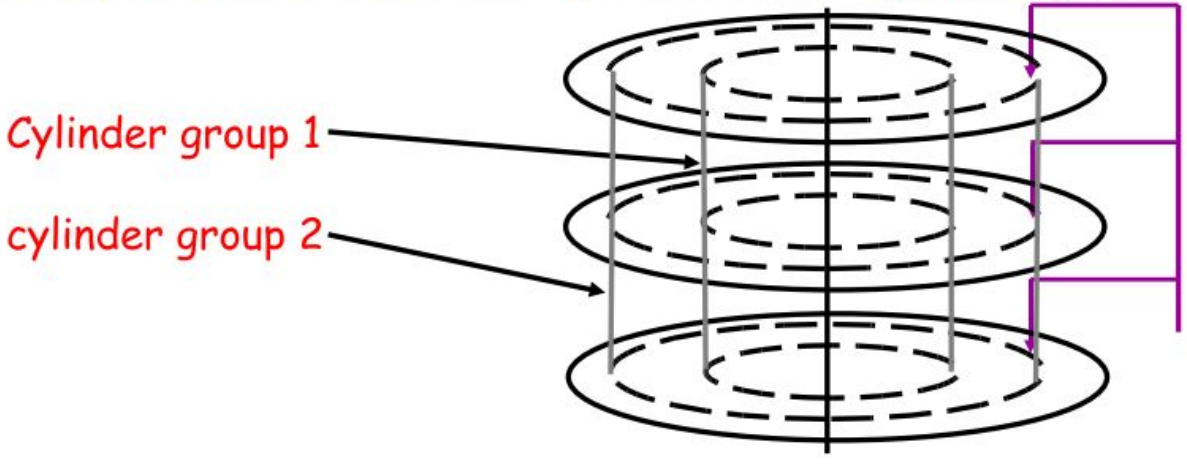
The idea is to design the file system structures and allocation policies to be “disk aware”



https://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf

Clustering related objects in FFS

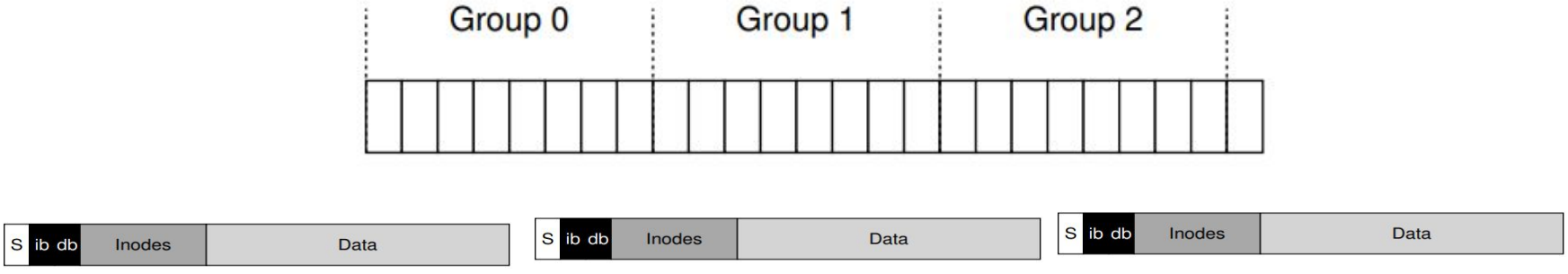
- **Group sets of consecutive cylinders into "*cylinder groups*"**



  - 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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

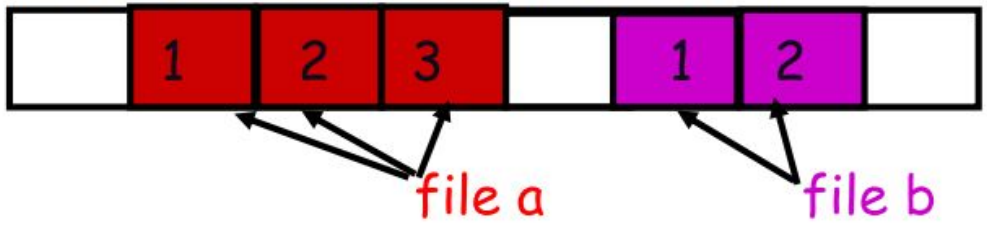
A per-group inode bitmap (ib) and data bitmap (db) serve this role for inodes and data blocks in each group



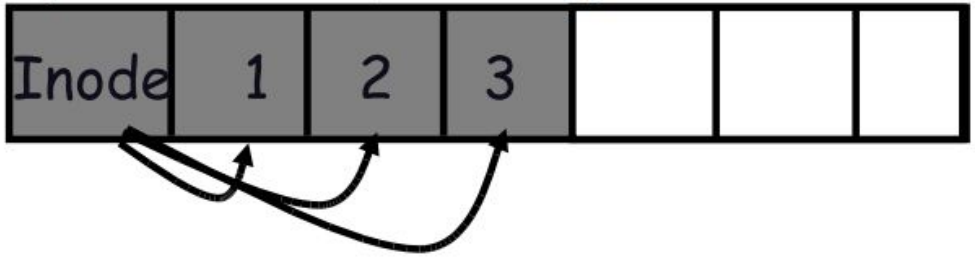
https://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf

Clustering in FFS

- **Tries to put sequential blocks in adjacent sectors**
  - (Access one block, probably access next)



- **Tries to keep inode in same cylinder group as file data:**
  - (If you look at inode, most likely will look at data too)

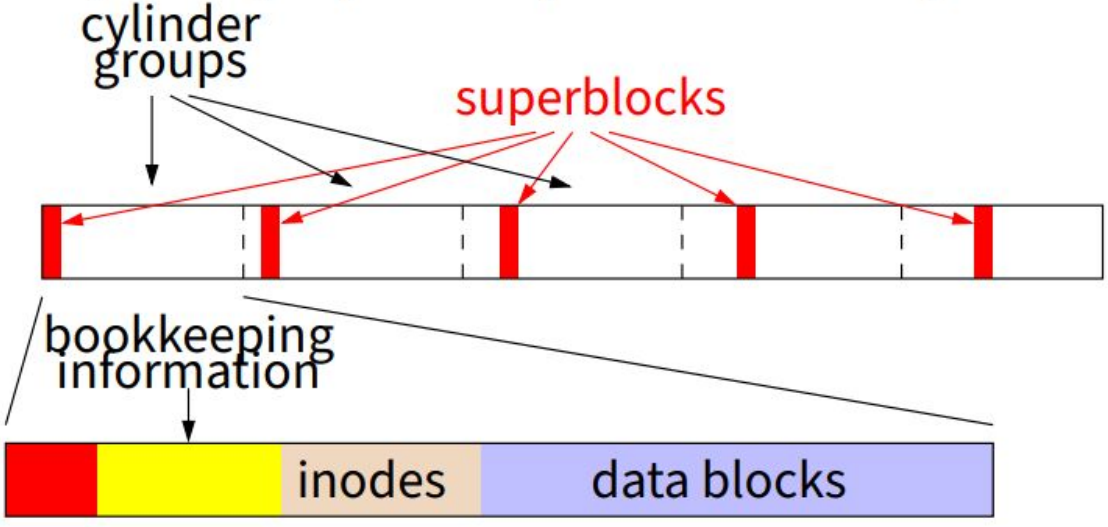


- **Tries to keep all inodes in a dir in same cylinder group**
  - Access one name, frequently access many, e.g., "`ls -l`"

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

What does disk layout look like?

- **Each cylinder group basically a mini-Unix file system:**

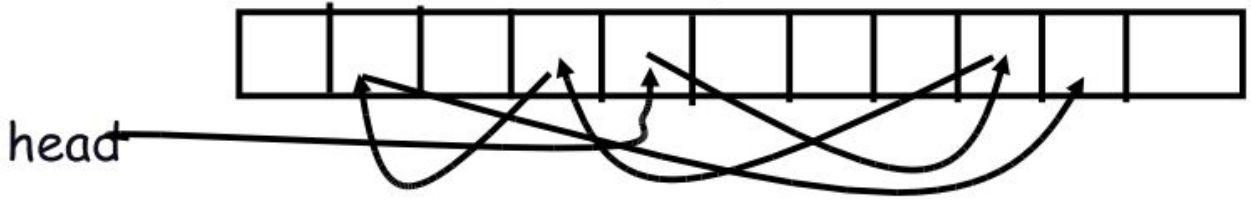


- **How how to ensure there's space for related stuff?**
  - Place different directories in different cylinder groups
  - Keep a "free space reserve" so can allocate near existing things
  - When file grows too big (1MB) send its remainder to different cylinder group.

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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**
  - 10101011111110000011111111000101100
  - Easier to find contiguous blocks.
  - Small, so usually keep entire thing in memory
  - Time to find free block increases if fewer free blocks

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 `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 (`df` 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

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 ext4fs, 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)

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

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 ($\approx 512\,\mathrm{B}/(100\,\mathrm{MB/s})) \approx$ 9ms
  - 50 sectors: 5ms + 4ms + .25ms = 9.25ms
  - How to use?
- **Fact: if transfer huge, seek + rotation negligible**
  - LFS: Hoard data, write out MB at a time

https://www.scs.stanford.edu/24wi-cs212/notes/file_systems.pdf

Crash recovery

How To Update The Disk Despite Crashes?

Early file systems took a simple approach to crash consistency

- Basically, they decided to let inconsistencies happen and then fix them later (when rebooting).
- A classic example of this lazy approach is found in a tool that does this: **fsck**

https://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf

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 containing 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, file names must be unique
  - All directories must be reachable

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

A basic summary of what fsck does:

Superblock: fsck first checks if the superblock looks reasonable

- mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated.

Free blocks: Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system

- It uses this knowledge to produce a correct version of the allocation bitmaps
- if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.

https://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf

A basic summary of what fsck does:

Inode state

Inode links

Duplicates

Bad blocks

Directory checks

building a working fsck requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging

https://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Strawman: just write all data asynchronously**
  - Any subset of data structures may be updated before a crash
- **Delete/truncate a file, append to other file, crash?**

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Crash recovery permeates FS code

- **Have to ensure fsck can recover file system**
- **Strawman: just write all data asynchronously**
  - Any subset of data structures may be updated before a crash
- **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!

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Sidenote: kernel-internal disk write routines

- **BSD has three ways of writing a block to disk**

1. `bdwrite` – **delayed write**
   - Marks cached copy of block as dirty, does not write it
   - Will get written back in background within 30 seconds
   - Used if block likely to be modified again soon
2. `bawrite` – **asynchronous write**
   - Start write but return immediately before it completes
   - E.g., use when appending to file and block is full
3. `bwrite` – **synchronous write**
   - Start write, sleep and do not return until safely on disk

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

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 (`bwrite`)**
  - 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
  - Makes `bawrite` as slow as `bwrite` for many updates to same block

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

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 slowing normal case does not make recovery fast**
  - If fsck takes one minute, then disks get 10× bigger, then 100× …
- **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**
  - Next: two advanced techniques

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

- Soft updates
- Journaling
- log structured file system (LFS)
- copy on write file systems

Soft updates an approach to maintaining file system **metadata integrity** in the event of a crash or power outage.

Journaling uses **transactions** to achieve **consistency**

- Neither journaling nor soft updates guarantees that no data will be lost,
- but they do make sure that the file system remains consistent.

First attempt: Ordered updates

- **Want to avoid crashing after "bad" subset of writes**
- **Must follow 3 rules in ordering updates [Ganger]:**
  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

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Example ordered updates

- **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*
  - By rule #1, must write *X* before writing *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

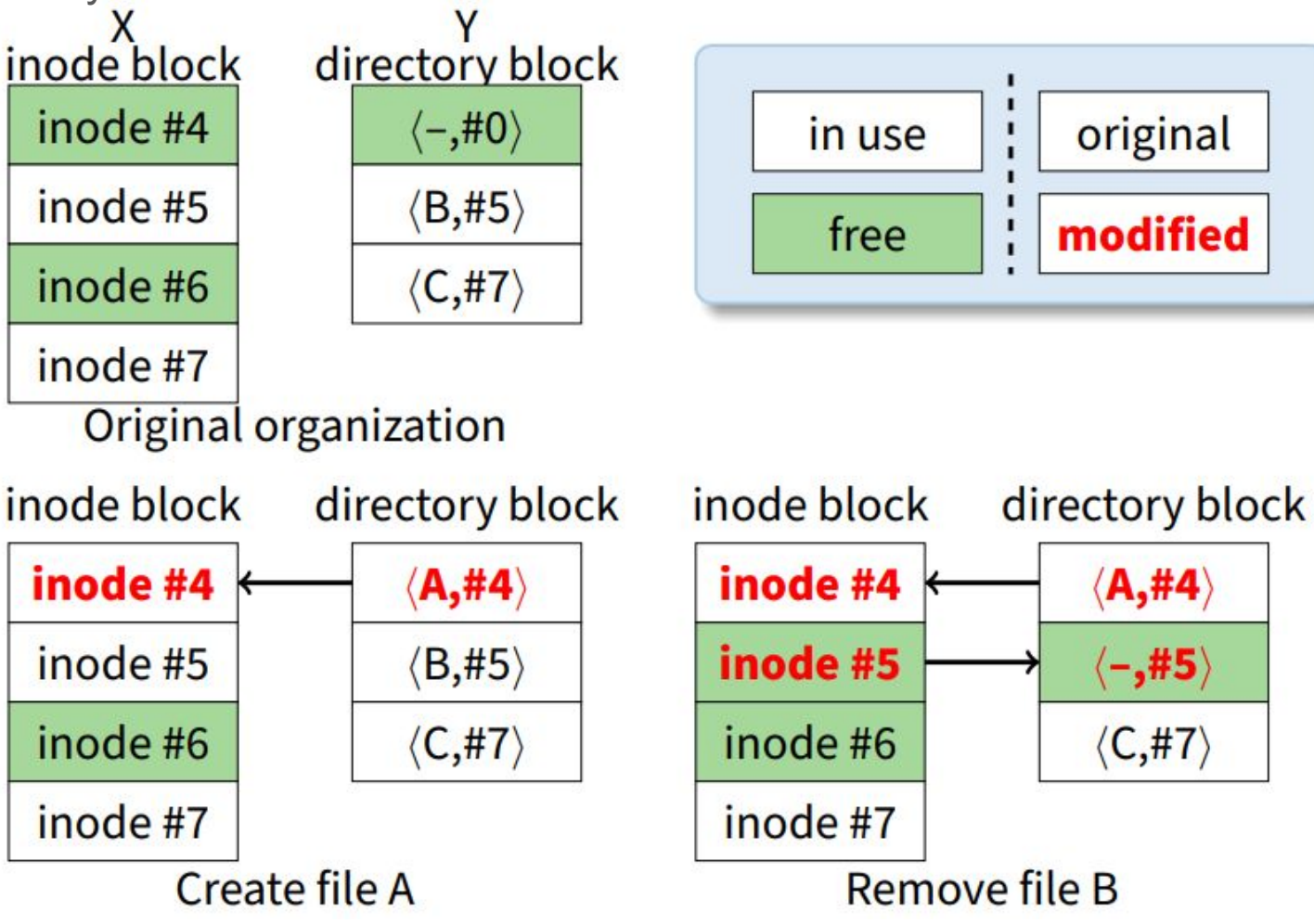
https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Problem: Cyclic dependencies

- **Suppose you create file *A*, unlink file *B*, but delay writes**
  - Both files in same directory block *Y* & inode block *X*
- **Rule #1: Must write *A*'s inode before dir. entry ($Y \rightarrow X$)**
  - 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
- **Rule #2: Must clear *B*'s dir. entry before writing inode ($X \rightarrow Y$)**
  - 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 slow**
  - Check every directory entry and every inode link count

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Cycle dependency



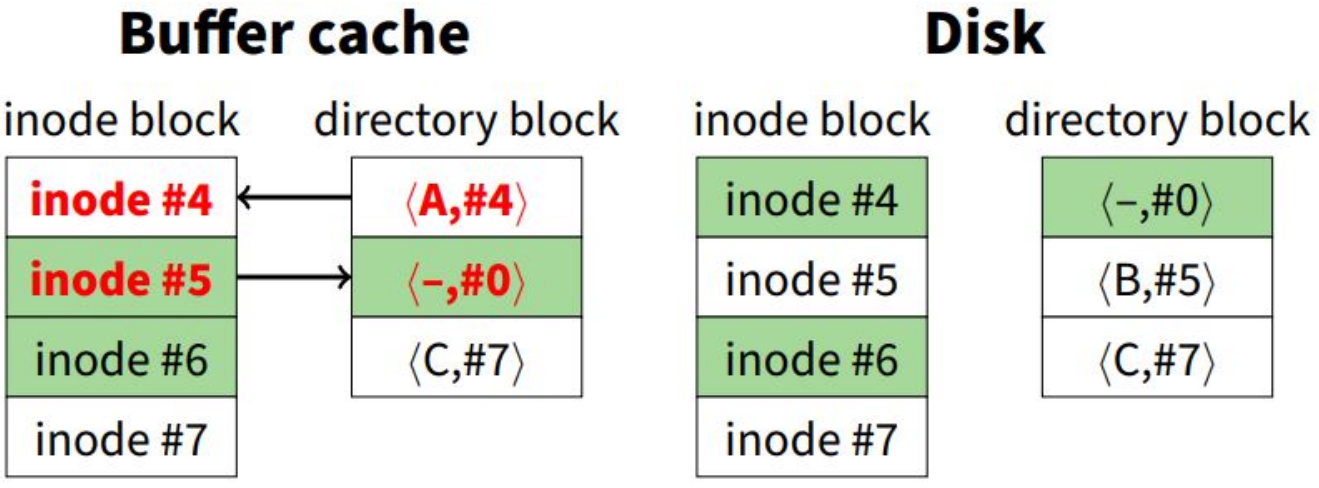
https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

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
  - I.e., can't write block with any arrows pointing to dependees …but can temporarily undo whatever change requires the arrow

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

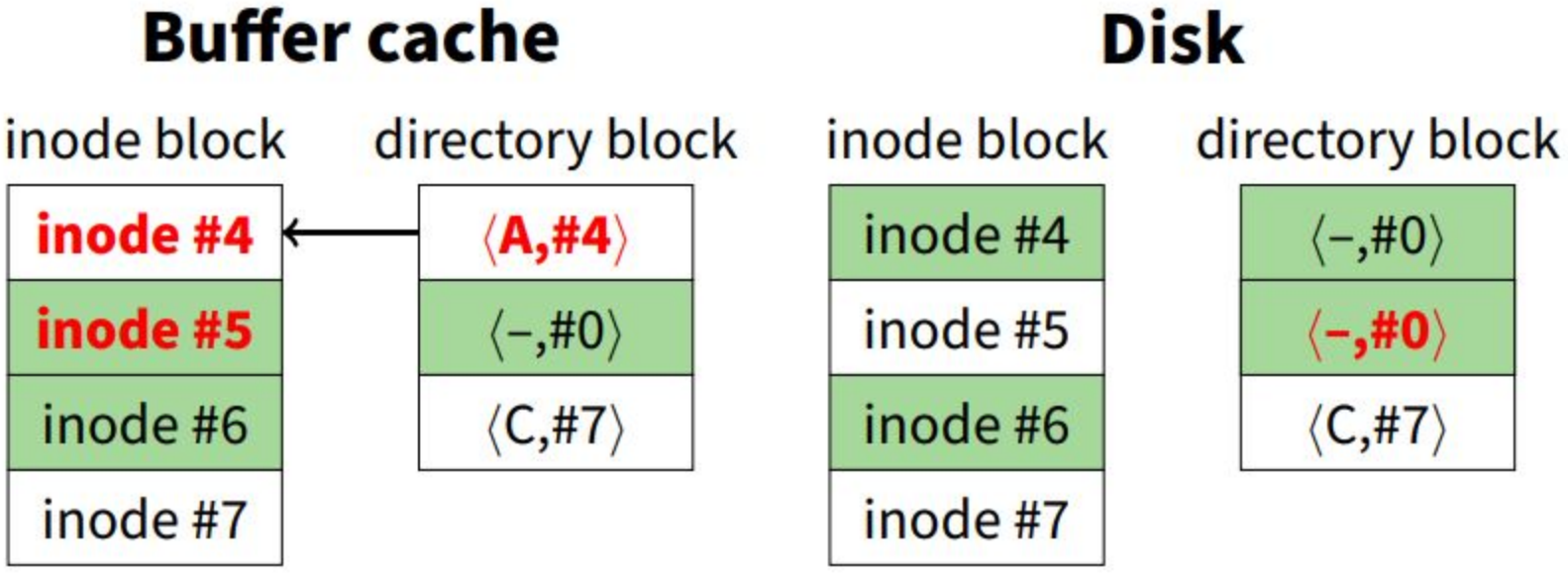
Breaking dependencies with rollback



- **Created file A and deleted file B**
- **Now say we decide to write directory block…**
- **Can't write file name *A* to disk—has dependee**

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

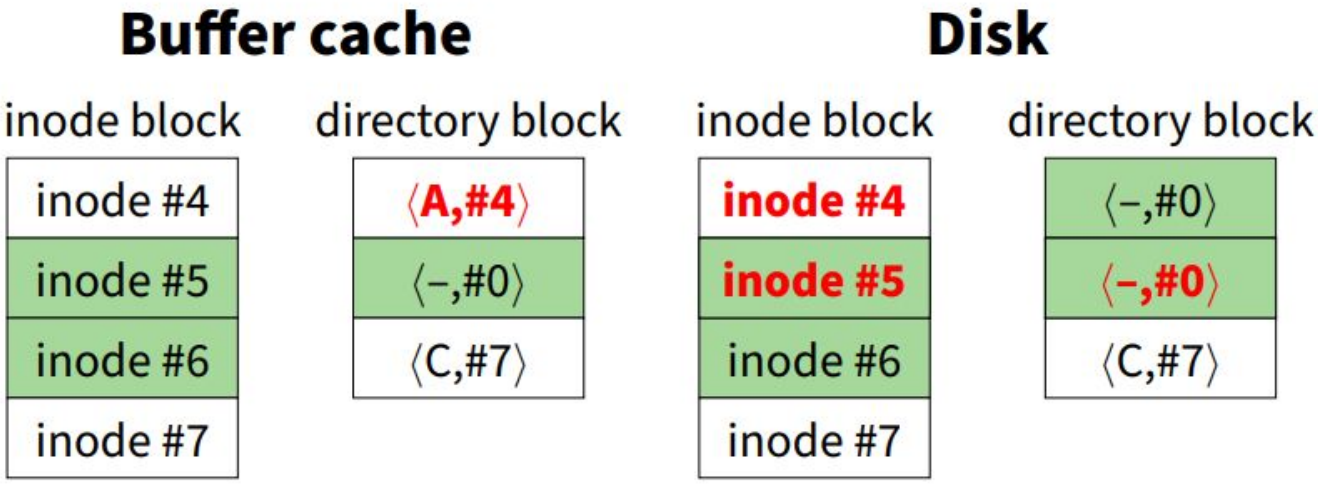
Breaking dependencies with rollback



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

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

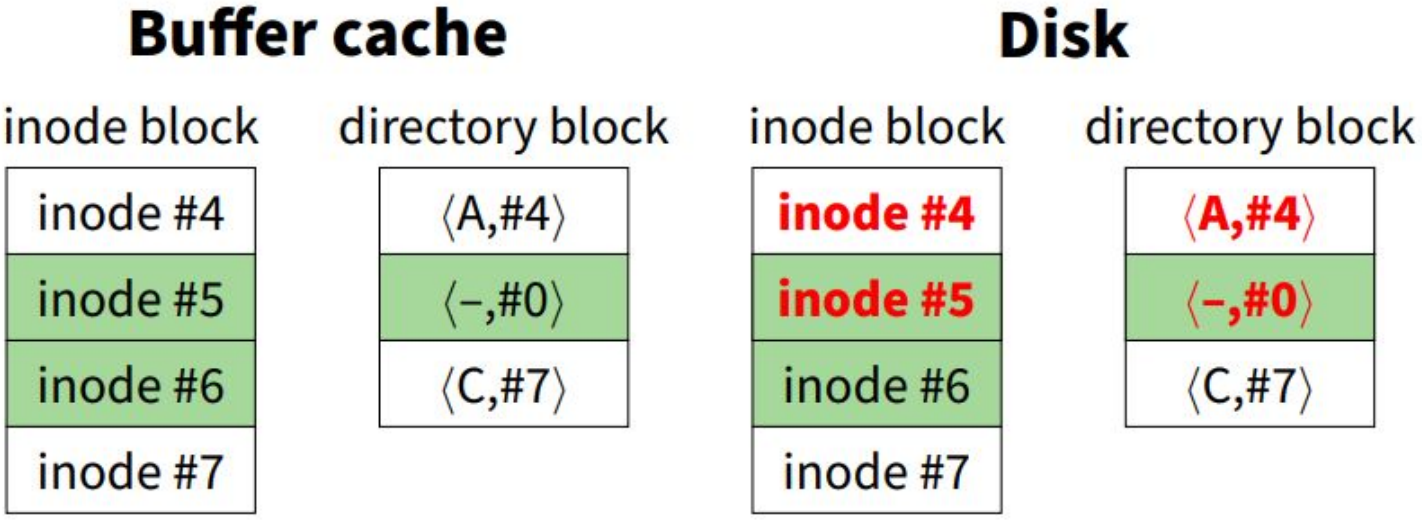
Breaking dependencies with rollback



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

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Breaking dependencies with rollback



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

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

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

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

An alternative: Journaling

- **Biggest crash-recovery challenge is inconsistency**
  - Have one logical operation (e.g., create or delete file)
  - Requires multiple separate disk writes
  - If only some of them happen, end up with big problems
- **Most of these problematic writes are to metadata**
- **Idea: Use a *write-ahead* log to *journal* metadata**
  - Reserve a portion of disk for a 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

Many uses this idea:

Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

- **Group multiple operations into one log entry**
  - E.g., clear directory entry, clear inode, update free map—either all three will happen after recovery, or none
- **Performance advantage:**
  - Log is consecutive portion of disk
  - Multiple operations can be logged at disk b/w
  - Safe to 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)

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Journaling details

- **Must find oldest relevant log entry**
  - Otherwise, redundant and slow to replay whole log
  - Worse, old directory/indirect blocks reallocated as data could get corrupted by old replay (because only metadata logged)
- **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

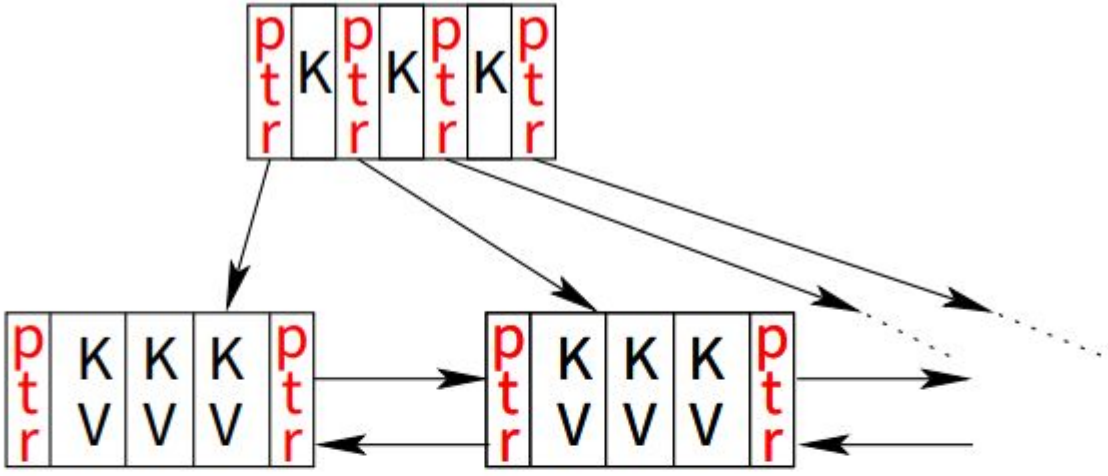
https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Case study: XFS

- **Main idea: Think big**
  - Big disks, files, large # of files, 64-bit everything
  - Yet maintain very good performance
- **Break disk up into *Allocation Groups* (AGs)**
  - 0.5 – 4 GiB regions of disk
  - New directories go in new AGs
  - Within directory, inodes of files go in same AG
  - Unlike cylinder groups, AGs too large to minimize seek times
  - Unlike cylinder groups, no fixed # of inodes per AG
- **Advantages of AGs:**
  - Parallelize allocation of blocks/inodes on multiprocessor (independent locking of different free space structures)
  - Can use 32-bit block pointers within AGs (keeps data structures smaller)

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

B+trees



- **XFS makes extensive use of B+-trees**
  - Indexed data structure stores ordered Keys & Values
  - Keys must have an ordering defined on them
  - Stored data in blocks for efficient disk access
- **For B+-tree with n items, all operations $O(\log n)$:**
  - Retrieve closest ⟨key, value⟩ to target key k
  - Insert a new ⟨key, value⟩ pair
  - Delete ⟨key, value⟩ pair

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Other approaches: Copy-on-write (yes, COW)

used e.g. ZFS

it places new updates to previously unused locations on disk.

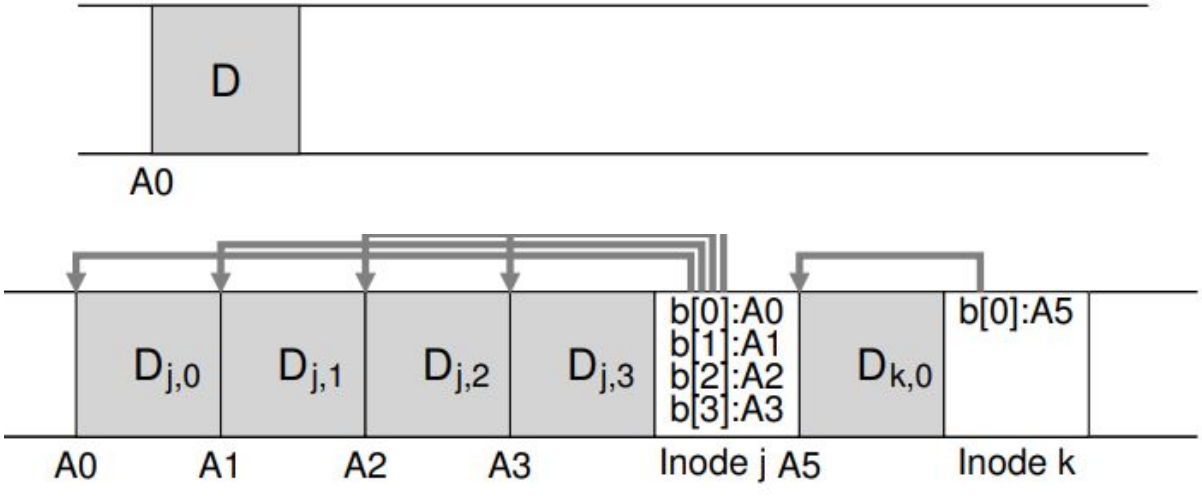
After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures.

https://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf

Other approaches: Log Structured File System (LFS)

- **Log-structured file system** is a file system in which data and metadata are written sequentially to a circular buffer, called a log.
- A log-structured file system thus treats its storage as a circular log and writes sequentially to the head of the log.

buffer writes, then commit all at once



https://pages.cs.wisc.edu/~remzi/OSTEP/file-lfs.pdf

Flash-Friendly File System (F2FS)

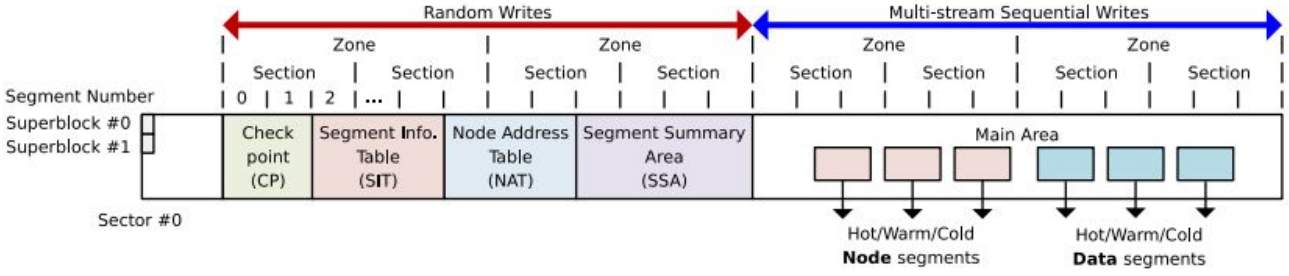
see
https://www.kernel.org/doc/Documentation/filesystems/f2fs.txt

Flash-Friendly File System (F2FS)

- **File system targeted at flash devices with FTL (e.g., SSDs)**
  - Try to do mostly large sequential writes
  - *Don't* attempt to do wear leveling (since have FTL anyway)
  - See also [Brown]
- **Break disk up into:**
  - Blocks – 4 KiB
  - Segments – 512 blocks, chosen so one block fits segment summary
  - Sections – 2^i segments (default $i = 0$), unit of log cleaning
  - Zones – n sections (default $n = 1$), if device internally comprises "subdevices," send parallel IO to different zones
- **Split device in two parts:**
  - Main area, in which to perform large sequential writes
  - Smaller metadata area has random writes, relies on FTL

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

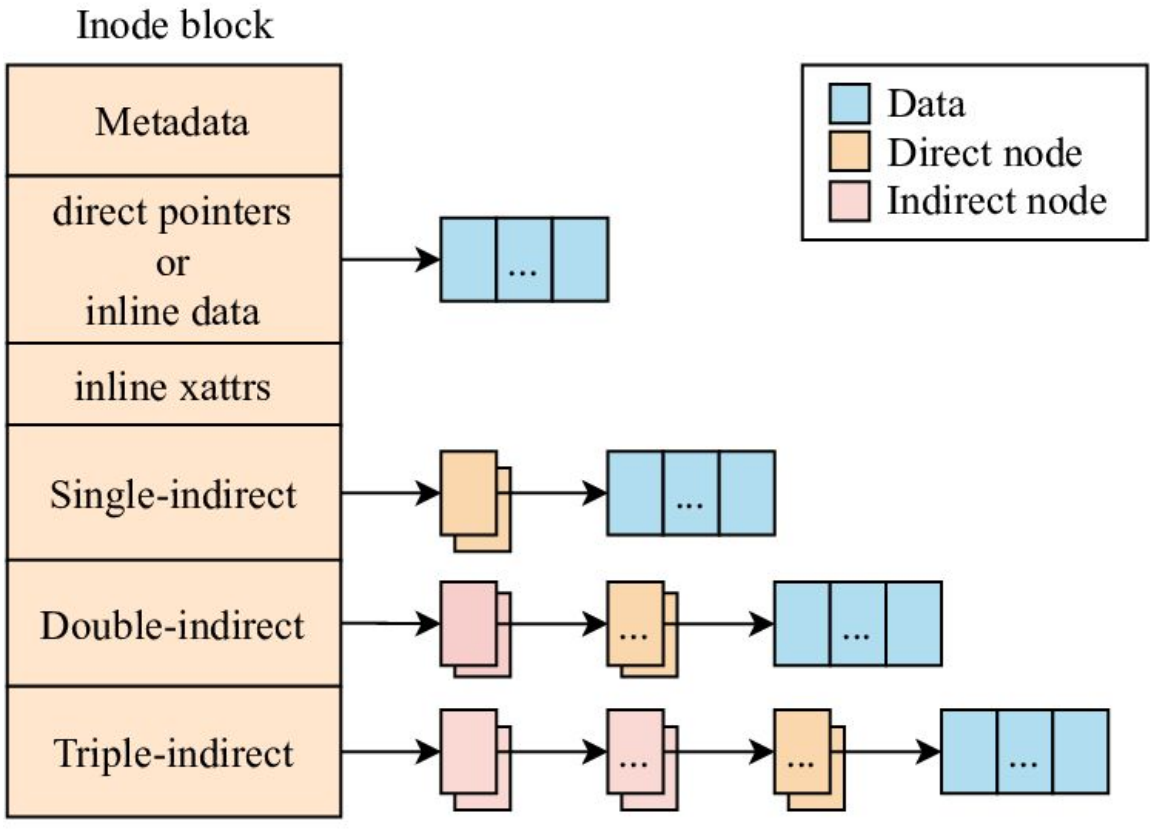
F2FS layout



- **CP – Valid SIT/NAT sets, list of orphan (open+deleted) inodes**
  - Place version # in header+footer, use consistent CP with highest #
- **SIT – Per-segment block validity bitmap and count**
  - Two SIT areas and a small journal avoids updating in place
  - CP says which SIT area is active
- **NAT – Translates node numbers to actual block storing node**
  - Updated like SIT
- **SSA – Parent info for each block (e.g., inode+offset)**
  - Just updated in place, CP records active ones to recover

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

F2FS inode



- **Small files (<3,692 bytes) stored "inline" inside inode**
- **Node pointers use NAT table for level of indirection**
  - Lets F2FS move a node without updating parent pointers

https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Multi-head logging

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

- **Two kinds of cleaning foreground and background**
  - Foreground (only if needed) greedily cleans most free section
  - Background just loads blocks into buffer cache and marks dirty
- **With no disk head, can efficiently maintain multiple logs**
  - Group data by similar expected lifetime (see above)
  - Means can clean empty or mostly empty sections

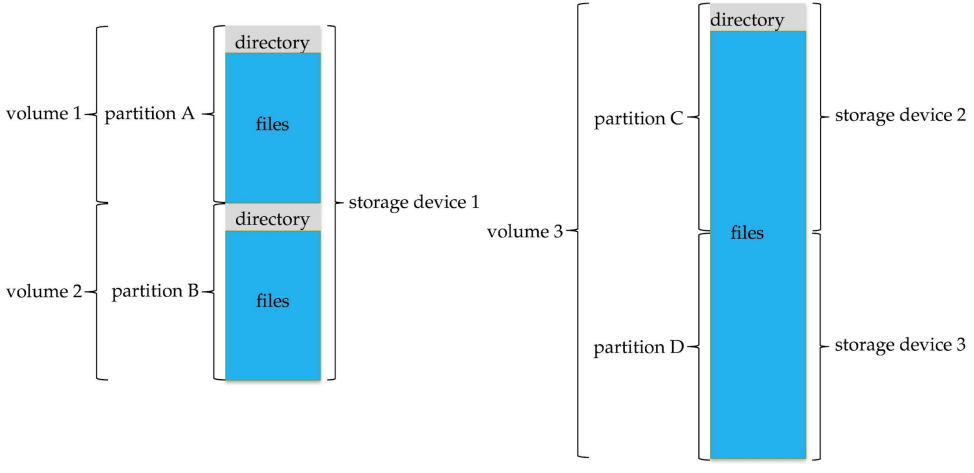
https://www.scs.stanford.edu/24wi-cs212/notes/advanced_fs.pdf

Network file systems, WAFL, other issues

not included in the exam(sinava dahil degil)

File System

- General-purpose computers can have multiple storage devices
- Devices can be sliced into partitions, which hold volumes
- Volumes can span multiple partitions
- Each volume usually formatted into a file system
- # of file systems varies, typically dozens available to choose from
- Typical storage device organization:



Solaris File Systems

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

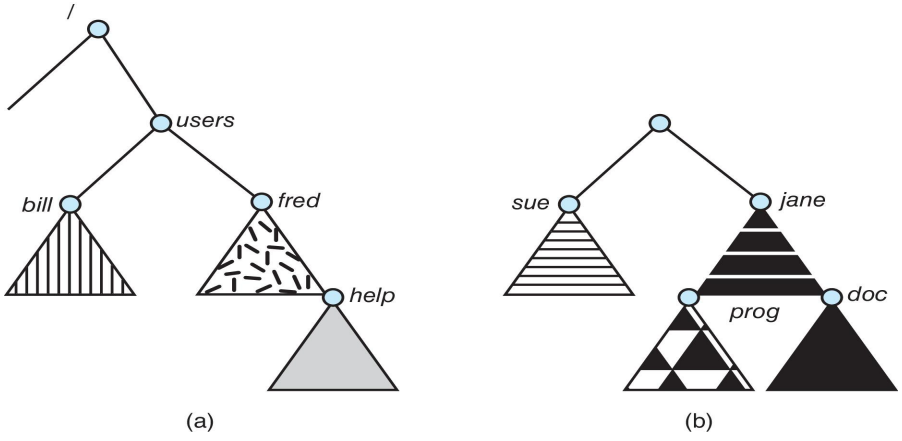
Partitions and Mounting

- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually on **mount points** – location at which they can be accessed
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

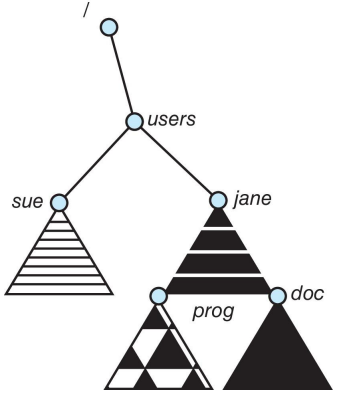
File Systems and Mounting

(a) Unix-like file system directory tree

(a) Unmounted file system



After mounting (b) into the existing directory tree



File Sharing

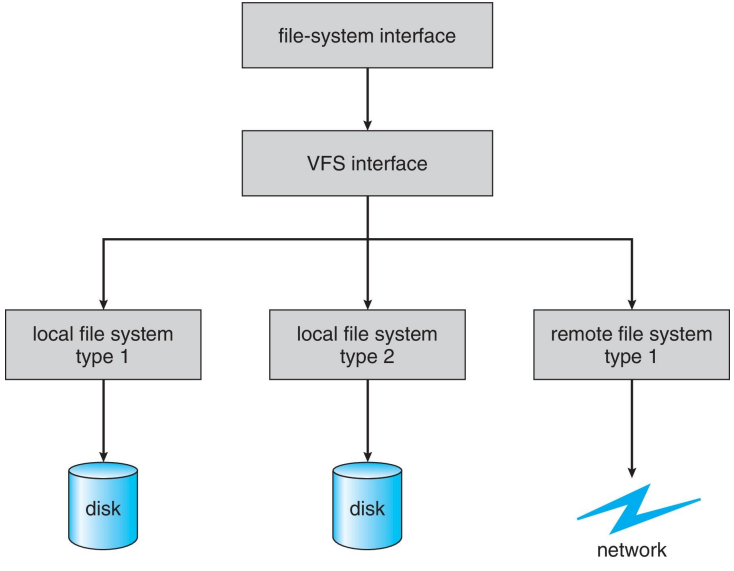
- Allows multiple users / systems access to the same files
- Permissions / protection must be implemented and accurate
  - Most systems provide concepts of owner, group member
  - Must have a way to apply these between systems

Virtual File Systems

- **Virtual File Systems** (**VFS**) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system
- Example



Virtual File System Implementation

- For example, Linux has four object types:
  - **inode, file, superblock, dentry**
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
    - • `int open(. . .)`—Open a file
    - • `int close(. . .)`—Close an already-open file
    - • `ssize t read(. . .)`—Read from a file
    - • `ssize t write(. . .)`—Write to a file
    - • `int mmap(. . .)`—Memory-map a file

Remote File Systems

- Sharing of files across a network
- First method involved manually sharing each file – programs like **ftp**
- Second method uses a **distributed file system** (**DFS**)
  - Remote directories visible from local machine
- Third method – **World Wide Web**
  - A bit of a revision to first method
  - Use browser to locate file/files and download /upload
  - **Anonymous** access doesn't require authentication

Client-Server Model

- Sharing between a server (providing access to a file system via a network protocol) and a client (using the protocol to access the remote file system)
- Identifying each other via network ID can be spoofed, encryption can be performance expensive
- NFS an example
  - User auth info on clients and servers must match (UserIDs for example)
  - Remote file system mounted, file operations sent on behalf of user across network to server
  - Server checks permissions, file handle returned
  - Handle used for reads and writes until file closed

Distributed Information Systems

- Aka **distributed naming services**, provide unified access to info needed for remote computing
- **Domain name system** (**DNS**) provides host-name-to-network-address translations for the Internet
- Others like **network information service** (**NIS**) provide user-name, password, userID, group information
- Microsoft's **common Internet file system** (**CIFS**) network info used with user auth to create network logins that server uses to allow to deny access
  - **Active directory** distributed naming service
  - **Kerberos-derived** network authentication protocol
- Industry moving toward **lightweight directory-access protocol** (**LDAP**) as secure distributed naming mechanism

Consistency Semantics

- Important criteria for evaluating file sharing-file systems
- Specify how multiple users are to access shared file simultaneously
  - When modifications of data will be observed by other users
  - Directly related to process synchronization algorithms, but atomicity across a network has high overhead (see Andrew File System)
- The series of accesses between file open and closed called **file session**
- UNIX semantics
  - Writes to open file immediately visible to others with file open
  - One mode of sharing allows users to share pointer to current I/O location in file
  - Single physical image, accessed exclusively, contention causes process delays
- Session semantics (Andrew file system (OpenAFS))
  - Writes to open file not visible during session, only at close
  - Can be several copies, each changed independently

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)
- The implementation originally part of SunOS operating system, now industry standard / very common
- Can use unreliable datagram protocol (UDP/IP) or TCP/IP, over Ethernet or other networks

NFS (Cont.)

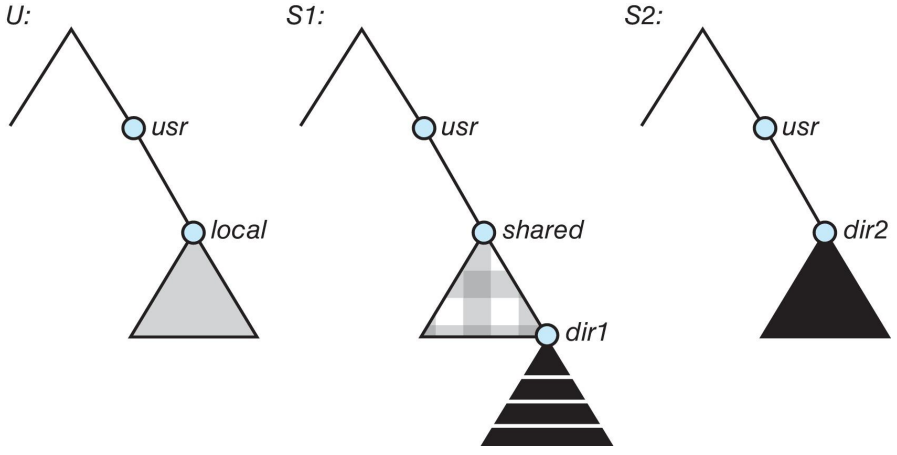
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

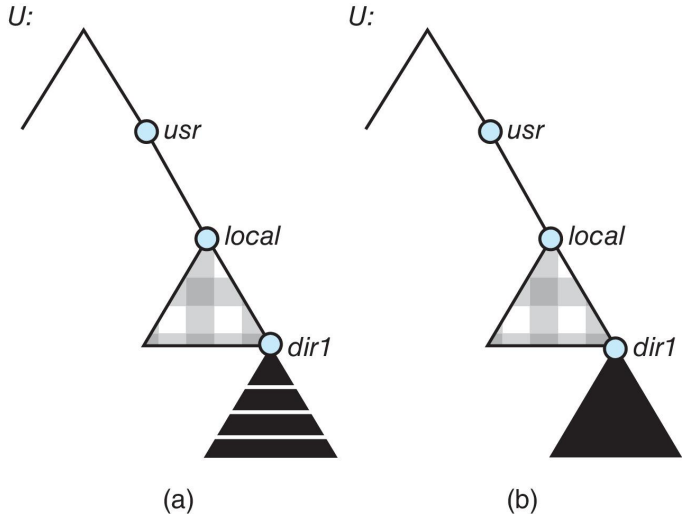
NFS Mounting Example

- Three independent file systems



NFS Mounting Example (Cont.)

- Mounts and cascading mounts



Mount s

Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user’s view and does not affect the server side

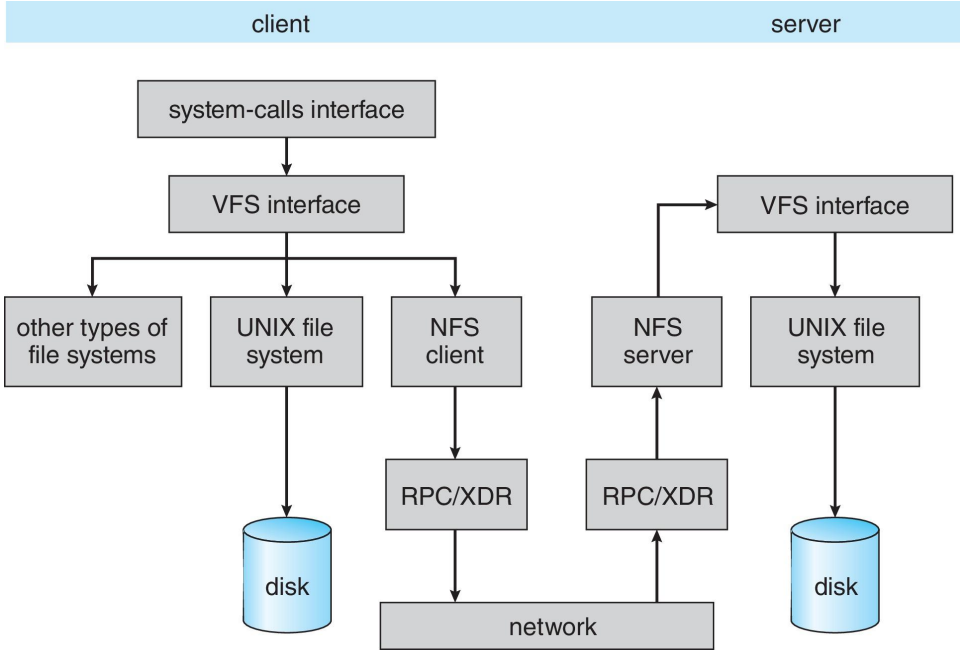
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is newer, less used – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)
- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests
- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names

NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
    - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file’s directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

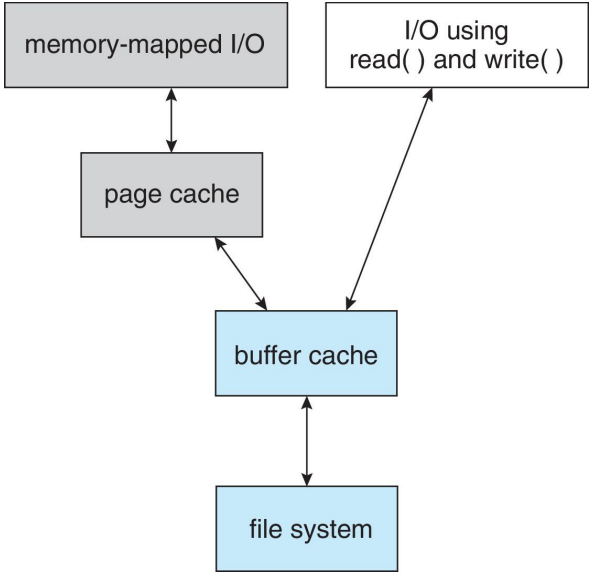
Efficiency and Performance (Cont.)

- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

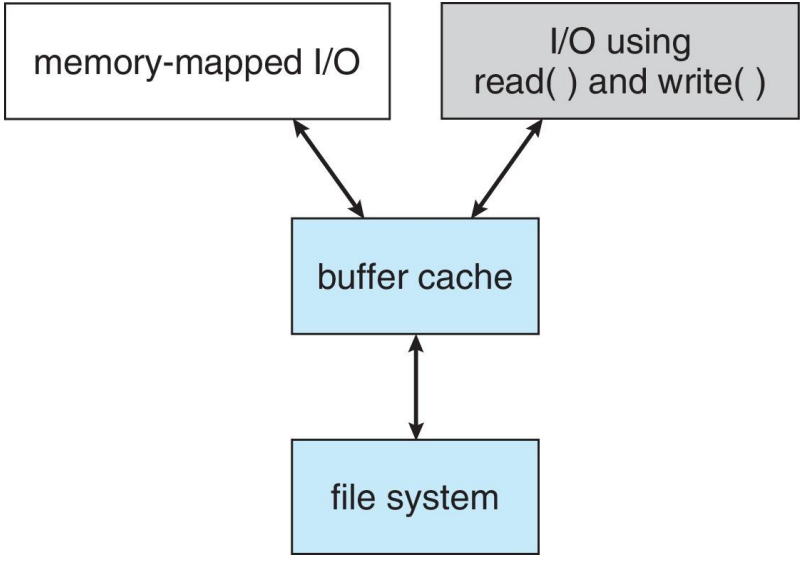
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
  - But which caches get priority, and what replacement algorithms to use?

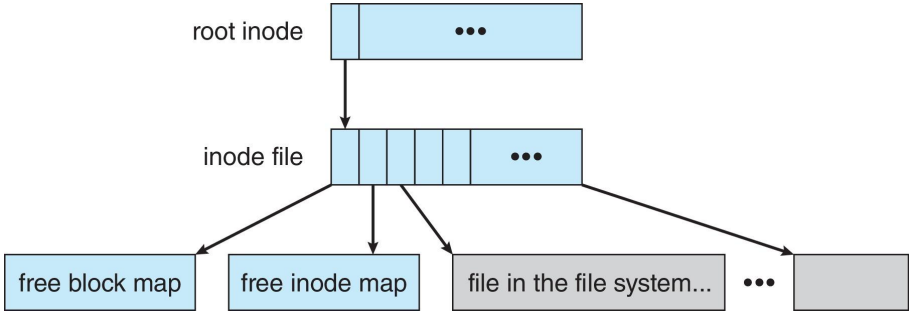
I/O Using a Unified Buffer Cache



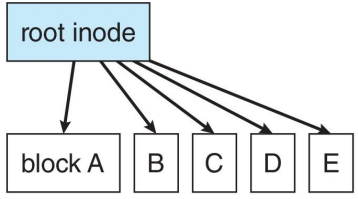
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

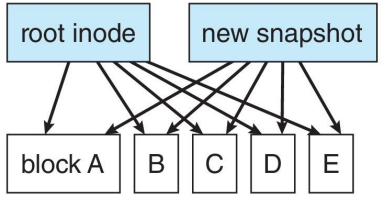
The WAFL File Layout



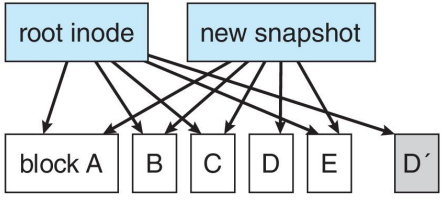
Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

Apple File System (**APFS**) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.