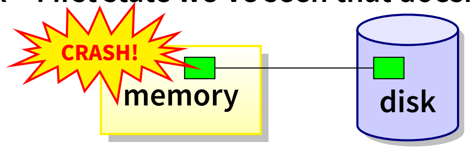
File system fun

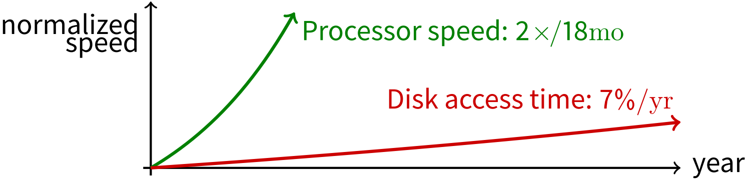
- **File systems: traditionally hardest part of OS**
  - More papers on FSes than any other single topic
- **Main tasks of file system:**
  - Don't go away (ever)
  - Associate bytes with name (files)
  - Associate names with each other (directories)
  - Can implement file systems on disk, over network, in memory, in non-volatile ram (NVRAM), on tape, w/ paper.
  - We'll focus on disk and generalize later
- **Today: files, directories, and a bit of performance**

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

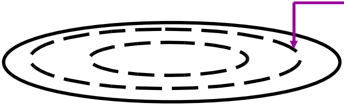
Disk vs. Memory

	Disk	MLC 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	100 MB/s	550–2500 MB/s	$>$ 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	$>$ 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile	Non-volatile	Volatile

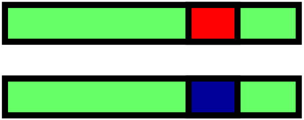
*Flash write performance degrades over time

Disk review

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



- **How to write a single byte? "Read-modify-write"**
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk
  - Key: if cached, don't need to read in



- **Sector = unit of atomicity.**

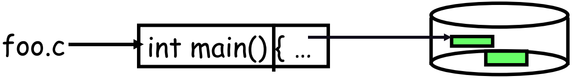
  - Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)
- **Larger atomic units have to be synthesized by OS**

Some useful trends

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

Files: named bytes on disk

- **File abstraction:**
  - User's view: named sequence of bytes



  - FS's view: collection of disk blocks
  - File system's job: translate name & offset to disk blocks:

- **File operations:**
  - Create a file, delete a file
  - Read from file, write to file
- **Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)**

What's hard about grouping blocks?

- **Like page tables, file system metadata are simply data structures used to construct mappings**
  - Page table: map virtual page # to physical page #

  - File metadata: map byte offset to disk block address

  - Directory: map name to disk address or file #

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 $(0 \ldots \text{filesize} - 1)$, $\sim$sequentially accessed
- **In some ways FS's problem is harder:**
  - Each layer of translation = potential disk access
  - Space a huge premium! (But disk is huge?!?!) Reason?
    Cache space never enough; amount of data you can get in one fetch never enough
  - Range very extreme: Many files $<$10 KB, some files many GB

Some working intuitions

- **FS performance dominated by # of disk accesses**
  - Say each access costs ~10 milliseconds
  - Touch the disk 100 extra times = 1 *second*
  - Can do a *billion* ALU ops in same time!
- **Access cost dominated by movement, not transfer:**

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

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

Common addressing patterns

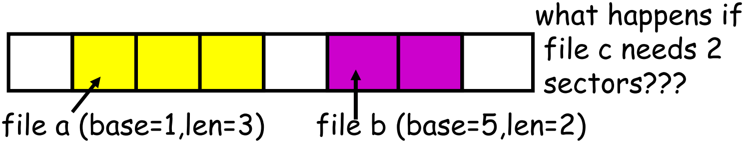
- **Sequential:**
  - File data processed in sequential order
  - By far the most common mode
  - Example: editor writes out new file, compiler reads in file, etc
- **Random access:**
  - Address any block in file directly without passing through predecessors
  - Examples: data set for demand paging,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

Problem: how to track file's data

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

Straw man: contiguous allocation

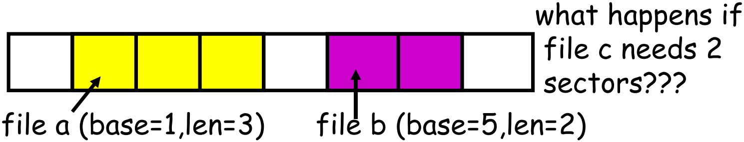
- **"Extent-based": allocate files like segmented memory**
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size



- **Example: IBM OS/360**
- **Pros?**
- **Cons? (Think of corresponding VM scheme)**

Straw man: contiguous allocation

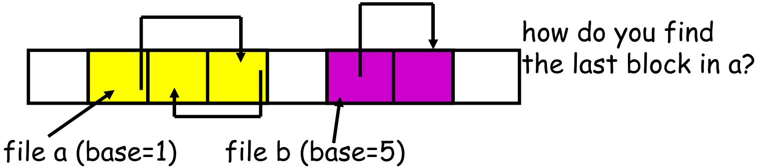
- **"Extent-based": allocate files like segmented memory**
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size



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

Straw man #2: Linked files

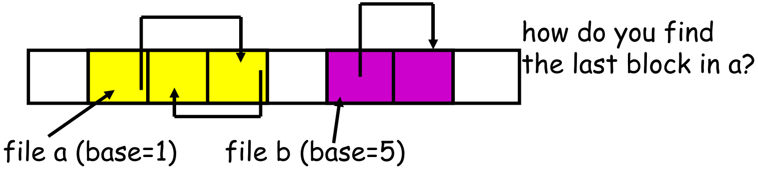
- **Basically a linked list on disk.**
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **Examples (sort-of): Alto, TOPS-10, DOS FAT**
- **Pros?**
- **Cons?**

Straw man #2: Linked files

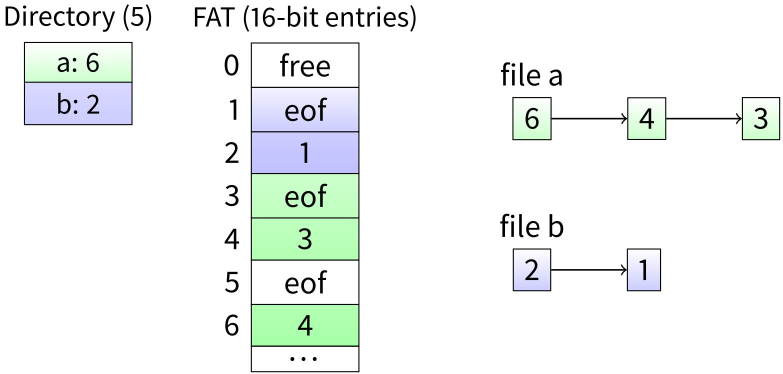
- **Basically a linked list on disk.**
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **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

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**

FAT discussion

- **Entry size = 16 bits**
  - What's the maximum size of the FAT?
  - Given a 512 byte block, what's the maximum size of FS?
  - One solution: go to bigger blocks. Pros? Cons?
- **Space overhead of FAT is trivial:**
  - 2 bytes / 512 byte block = $\sim$ 0.4% (Compare to Unix)
- **Reliability: how to protect against errors?**
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- **Bootstrapping: where is root directory?**
  - Fixed location on disk:

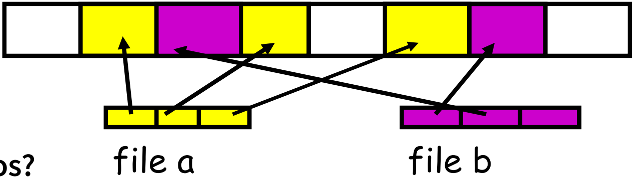
FAT discussion

- **Entry size = 16 bits**
  - What's the maximum size of the FAT? 65,536 entries
  - Given a 512 byte block, what's the maximum size of FS? 32 MiB
  - One solution: go to bigger blocks. Pros? Cons?
- **Space overhead of FAT is trivial:**
  - 2 bytes / 512 byte block = $\sim$ 0.4% (Compare to Unix)
- **Reliability: how to protect against errors?**
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- **Bootstrapping: where is root directory?**
  - Fixed location on disk:

FAT	(opt) FAT	root dir	...

Another approach: Indexed files

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

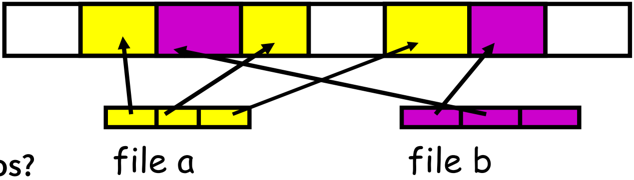


- **Pros?**

- **Cons?**

Another approach: Indexed files

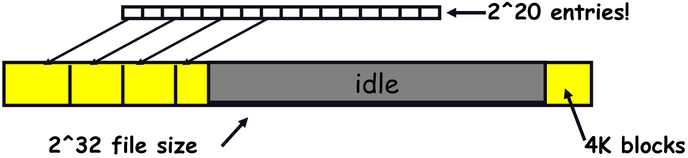
- **Each file has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



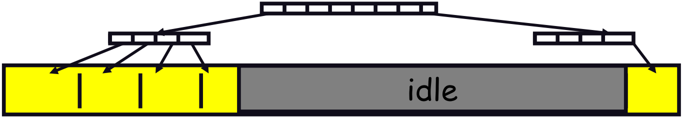
- **Pros?**
  - Both sequential and random access easy
- **Cons?**
  - Mapping table requires large chunk of contiguous space
    ...Same problem we were trying to solve initially

Indexed files

- **Issues same as in page tables**

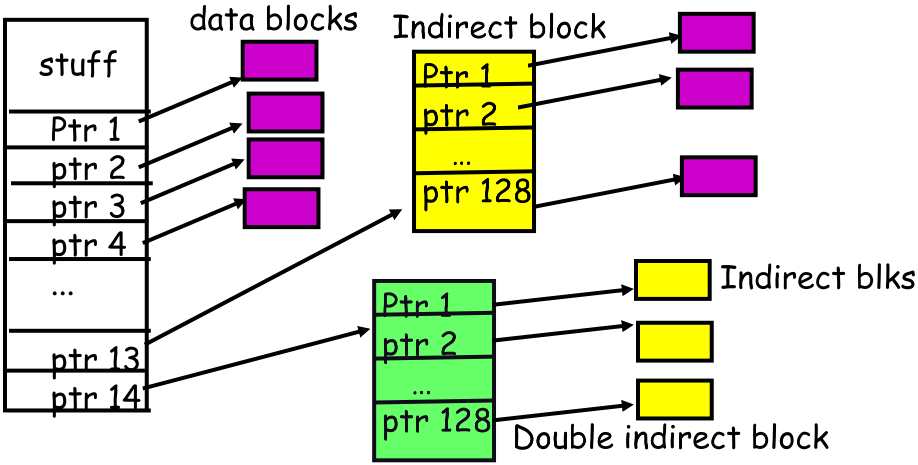


  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk
- **Solve identically: small regions with index array, this array with another array, . . . Downside?**



Multi-level indexed files (old BSD FS)

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

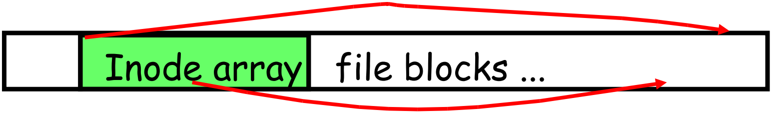


Old BSD FS discussion

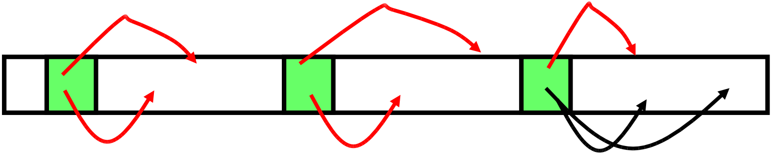
- **Pros:**
  - Simple, easy to build, fast access to small files
  - Maximum file length fixed, but large.
- **Cons:**
  - What is the worst case # of accesses?
  - What is the worst-case space overhead? (e.g., 13 block file)
- **An empirical problem:**
  - Because you allocate blocks by taking them off unordered freelist, metadata and data get strewn across disk

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

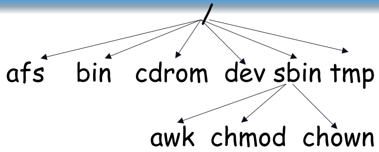
Directories

- **Problem:**
  - "Spend all day generating data, come back the next morning, want to use it." – F. Corbato, on why files/dirs invented
- **Approach 0: Users remember where on disk their files are**
  - E.g., like remembering your social security or bank account #
- **Yuck. People want human digestible names**
  - We use directories to map names to file blocks
- **Next: What is in a directory and why?**

A short history of directories

- **Approach 1: Single directory for entire system**
  - Put directory at known location on disk
  - Directory contains $\langle \text{name}, \text{inumber} \rangle$ 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.)

Hierarchical Unix



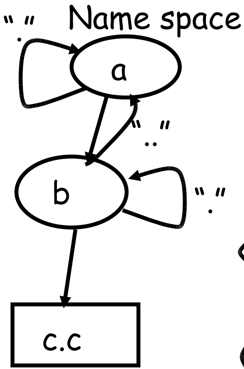
- **Used since CTSS (1960s)**
  - Unix picked up and used really nicely
- **Directories stored on disk just like regular files**
  - Special inode type byte set to directory
  - User's can read just like any other file
  - Only special syscalls can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree (what needed to make a DAG?)
- **Simple, plus speeding up file ops speeds up dir ops!**

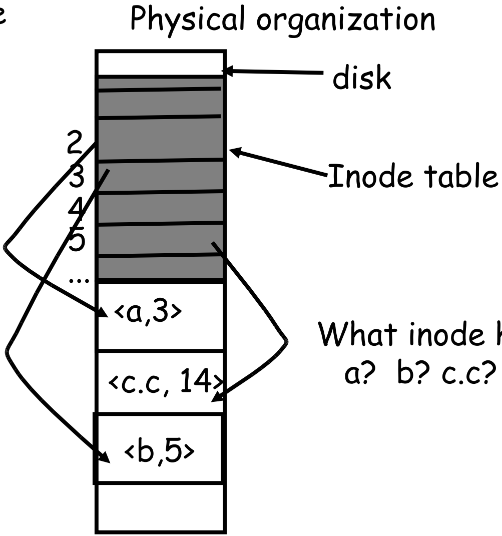
<name,inode#>
<afs,1021>
<tmp,1020>
<bin,1022>
<cdrom,4123>
<dev,1001>
<sbin,1011>
⋮

Naming magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)
- **Special names:**
  - Root directory: "`/`"
  - Current directory: "`.`"
  - Parent directory: "`..`"
- **Some special names are provided by shell, not FS:**
  - User's home directory: "$\sim$"
  - Globbing: "`foo.*`" expands to all files starting "`foo.`"
- **Using the given names, only need two operations to navigate the entire name space:**
  - `cd` *name*: move into (change context to) directory *name*
  - `ls`: enumerate all names in current directory (context)

Unix example: /a/b/c.c





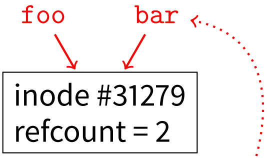
What inode holds file for
a? b? c.c?

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**

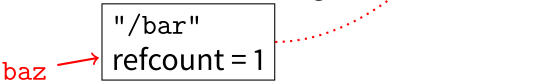
Hard and soft links (synonyms)

- **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

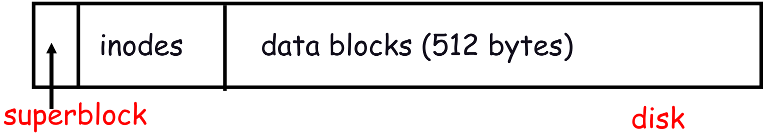
```
ln -s /bar baz
```



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

Case study: speeding up FS

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



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

A plethora of performance costs

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

Problem: Internal fragmentation

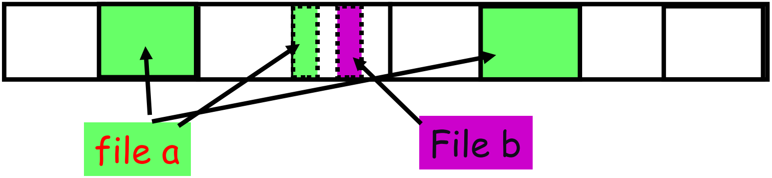
- **Block size was too small in Unix FS**
- **Why not just make block size bigger?**

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

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

Solution: fragments

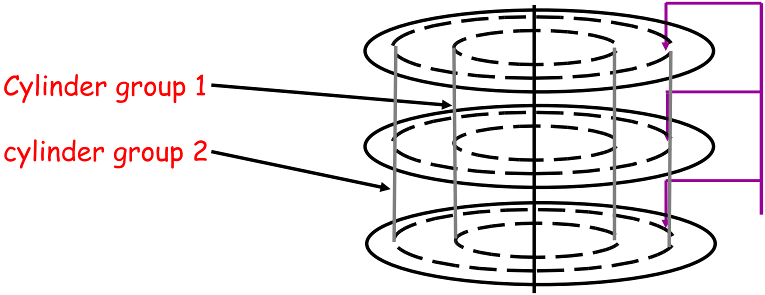
- **BSD FFS:**
  - Has large block size (4096 or 8192)
  - Allow large blocks to be chopped into small ones (“fragments”)
  - Used for little files and pieces at the ends of files



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

Clustering related objects in FFS

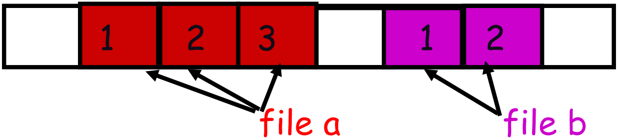
- **Group sets of consecutive cylinders into "*cylinder groups*"**



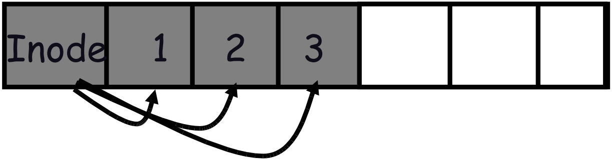
  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group

Clustering in FFS

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



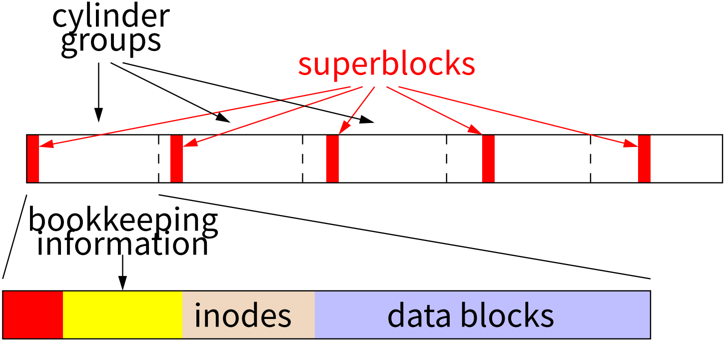
- **Tries to keep inode in same cylinder as file data:**
  - (If you look at inode, most likely will look at data too)



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

What does disk layout look like?

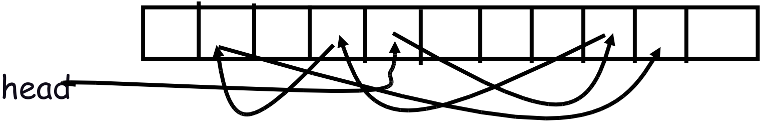
- **Each cylinder group basically a mini-Unix file system:**



- **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.

Finding space for related objs

- **Old Unix (& DOS): Linked list of free blocks**
  - Just take a block off of the head. Easy.



  - Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow
- **FFS: switch to bit-map of free blocks**
  - 1010101111111000001111111000101100
  - Easier to find contiguous blocks.
  - Small, so usually keep entire thing in memory
  - Time to find free block increases if fewer free blocks

Using a bitmap

- **Usually keep entire bitmap in memory:**
  - 4G disk / 4K byte blocks. How big is map?
- **Allocate block close to block *x*?**
  - Check for blocks near `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

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 ext2fs, XFS)
- **Writes of metadata done synchronously**
  - Really hurts small file performance
  - Make asynchronous with write-ordering ("soft updates") or logging/journaling… more next lecture
  - Play with semantics (/tmp file systems)

Other hacks

- **Obvious:**
  - Big file cache
- **Fact: no rotation delay if get whole track.**
  - How to use?
- **Fact: transfer cost negligible.**
  - Recall: Can get 50x the data for only $\sim$3% more overhead
  - 1 sector: 5ms + 4ms + 5μs ($\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
- **Next lecture:**
  - FFS in more detail
  - More advanced, modern file systems