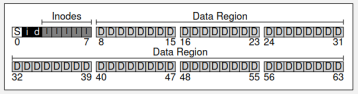
A Simple File System – Organization

- the disk partition is viewed as a series of blocks
  - numbered 0 to N-1



- inodes contain pointers to the data blocks
  - classic Unix implementation is a multilevel index
    - inode has a fixed number of direct pointers and one indirect pointer to a block containing pointers to data blocks
    - to support larger files, inode also includes double and triple indirect pointers
  - imbalanced tree design reflects that most files are small, and having more levels than needed wastes space
- *inode bitmap* (i) tracks usage of inodes
- *data bitmap* (d) tracks usage of data blocks
  - 1 bit per inode/block
- *superblock* (S) stores file system information

Directory Organization

- at its simplest, a directory is a list of entry names and inode numbers

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

  - often treated as a special type of file
    - directory entries are stored in data blocks
    - inode contains a type marker so system knows how to interpret the data
- fancier data structures are possible

Free Space Management

- need to track both available inodes and available data blocks
- simple solution is a bitmap for each
  - must search to find a free inode or data block
- *pre-allocation* allocates a small sequence of data blocks when a new file is created
  - allows small files and the first part of larger files to occupy a contiguous chunk of space

Access Paths – Reading

to find the inode number for the file, it is necessary to traverse the pathname starting from the root
– root inode number is well-known (typically 2 in UNIX systems)
– read root inode, check permissions, find the data block with the directory contents
– read the root directory data block to find the inode number for foo
– read foo's inode, check permissions, find the data block with the directory contents
– read the foo directory data block to find the inode number for bar

finally read bar's inode, check permissions, allocate and return a file descriptor

repeat for each block of the file –
– read bar's inode to find the next data block of the file
– read the data block
– update last accessed time in the inode
– update the file offset in the open file table

to close, deallocate the file descriptor

- open `/foo/bar`
- read the file (3 blocks)
- close

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			read	read	read	read	read	read		
read()					read write				read	
read()					read write					read
read()					read write					read

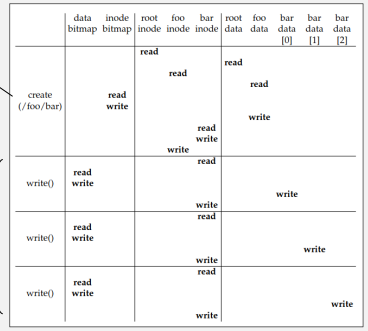
Access Paths – Writing

- create `/foo/bar`
- write to the file (3 blocks)
- close

to create a new file –
– traverse directory structure (inodes and data blocks) to find the inode and data block for the containing directory (foo)
– read inode bitmap, locate a free inode, update and write inode bitmap
– initialize and write new file inode (for bar)
– update and write directory data block (for foo)
– update and write directory inode (for foo)
+ additional work if it is necessary to allocate a new directory data block

to write a new file requires allocating data blocks –
– read inode
– read data bitmap, locate a free data block, update and write data bitmap
– update inode to include a pointer to the new data block, write inode

to close, deallocate the file descriptor



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
			read			read				
				read			read			
create (/foo/bar)		read write								
					read write		write			
					write					
write()	read write				read					
					write			write		
write()	read write				read					
					write				write	
write()	read write				read					
					write					write

Caching

- disk I/O is slow, and many such requests are needed when reading and writing files
  - it is especially bad to locate files deep within the directory hierarchy
- solution: use memory to cache
- modern OSes use *dynamic partitioning*, combining virtual memory pages and file system pages into a *unified page cache*
  - subsequent opens of the same file or files in the same directory benefit from cache hits for the directory inode and data blocks

Buffering

- disk I/O on writes can't be entirely avoided through caching but *write buffering* helps
  - delay writes to batch updates
  - allows for coalescing, scheduling, and sometimes avoidance (e.g. file is deleted before writes are written to disk)
- writes are commonly buffered for 5-30s
  - tradeoff is if there is a crash during this time, pending updates are lost

The Story So Far

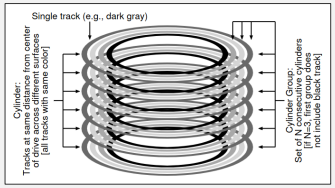
- the original Unix file system looked much like the simple file system just discussed
  - maintained a free list rather than inode and disk block bitmaps
  - file abstraction and directory hierarchy were important advances
  - performance was terrible
    - data blocks not close to inodes
    - lack of clever free space management resulted in high levels of fragmentation
    - block size (512 bytes) was too small

THE CRUX:
HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

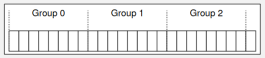
solution: FFS (Fast File System)
strategy: same abstractions and interface, different implementation

FFS: Disk Organization

- organize disk into groups of blocks with minimal seek time within each group
- cylinder groups
  - a *cylinder* consists of the tracks the same distance from the center on all platters
  - a *cylinder group* consists of *N* consecutive cylinders



- block groups
  - a *block group* is a consecutive group of blocks
  - approximates cylinder groups for modern disks which provide only a logical address space



FFS: Disk Organization

- each group has the same structure as the simple file system discussed

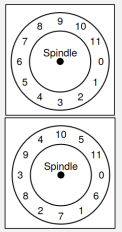
  - a copy of the super block
    - provides redundancy in case one copy becomes corrupt
  - per-group inode bitmap and data bitmap
  - per-group inodes, data blocks

FFS: Policies

- philosophy: *keep related stuff together*
  - *together* = in the same block group
- heuristics – directories
  - find a block group with a low number of allocated directories and a high number of free inodes
    - helps balance directories across groups and have room for lots of files
- heuristics – files
  - allocate data blocks for a file in the same group as its inode
  - put files in the same block as the directory that contains them
    - exploits namespace locality – files in the same directory are commonly accessed together
  - large files are split up across multiple block groups
    - can choose a chunk size large enough to mitigate performance loss due to seeking between blocks
    - FFS matches chunk size to the structure of the inode
      - first 12 direct blocks (48KB) are in the same group as the inode
      - each indirect block and its data blocks (4MB) are in a different group

FFS: Other Features

- *sub-blocks*
  - many files were ~2KB but 4KB block size meant up to 50% of the disk capacity could be wasted due to internal fragmentation
  - solution: allocate 512-byte sub-blocks until 4KB filled, then copy the 8 sub-blocks into a single 4KB block
  - to avoid the extra I/O, `libc` buffers writes into 4KB blocks
- *parameterization*
  - if a file's data blocks are arranged sequentially, by the time block *k* has been read and a request for block *k*+1 issued, block *k*+1 will have rotated too far
  - instead, stagger the layout
    - can be tuned according to the specific performance parameters of the disk
  - modern drives read the entire track into a *track buffer*
    - OS no longer has to manage low-level drive details



FFS: Usability

- support for long file names
  - not limited to 8 characters (or 8+3)
- support for symbolic links
- atomic `rename()` for files