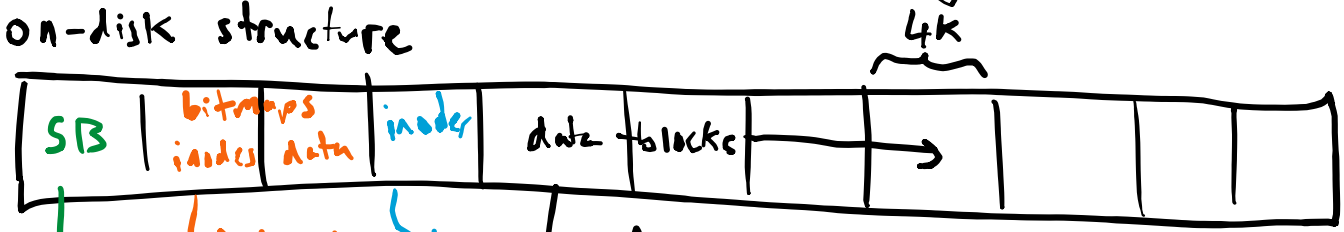
Last time: VSFS (very simple file system)

today: efficiency Fast File System

on-disk structure



metadata
type
size
inodes
blocks

allocation "free list"

metadata
type,
size,
perms
[pointers to data blocks

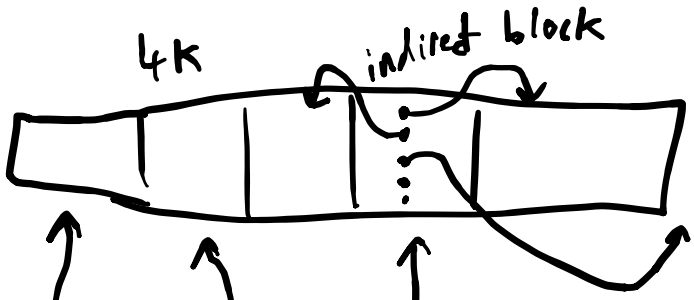
file content
directory contents
mapping between human readable → inode #

/x/y/z /a/b/c
path traversal

Q's: large files? inode design

what about efficiency? caching... FFS

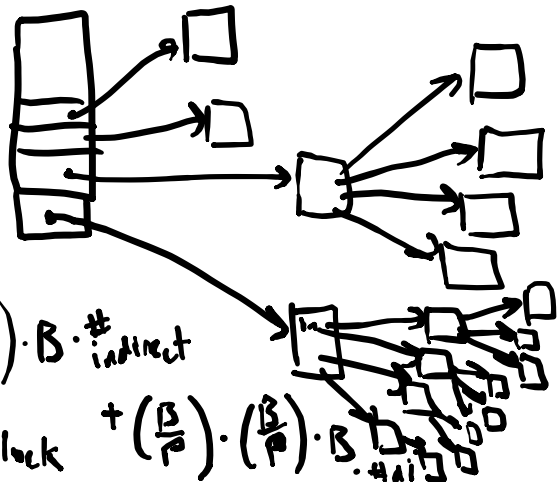
inode:
type
size
perms
pointers to data blocks



direct pointer

indirect pointer

double indirect pointer

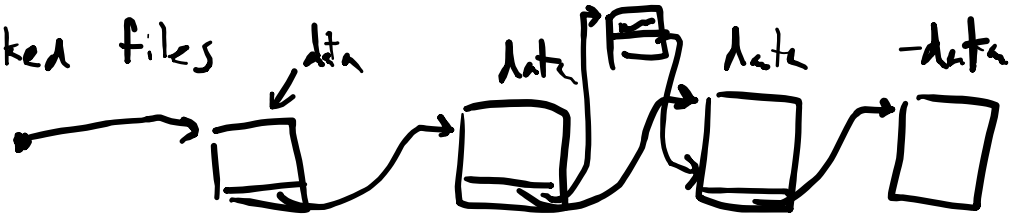


4k size B

size file we can handle $\left[\#\text{direct} \cdot B + \left(\frac{B}{P}\right) \cdot B \cdot \#\text{indirect} + \left(\frac{B}{P}\right) \cdot \left(\frac{B}{P}\right) \cdot B \right]$

pointers per block

FAT linked files



Extents: (ptr + length) contiguous blocks

Efficiency: /a/b/c/d/e/f.txt

root inode
/ data
a inode
data
b inode
⋮

works well for reads

Keep in memory
buffer-cache: just FS
page cache: unified virtual memory

what about writes?
in background, write to disk
[batching, scheduling]
how frequently should you write

Old UNIX FS
1. API everyone loved
2. performance was terrible
   2% of peak & getting worse

problems: seeking too much

→ inodes & data not "close"
→ related files? not "close"
→ fragmentation (internal)

overall: FS assumed disk was like RAM
→ treat disk like a disk ⇒ FFS

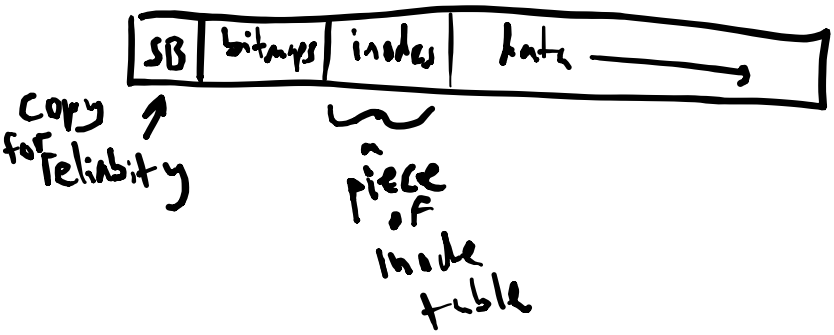
FFS: treat disk like a disk

1) chop disk up into groups
2) put related things in same group

"cylinder group" same cylinder = no seek

today: device doesn't expose geometry
but locality does have perf differences
"block groups"

Each group:



Allocation policies

related?

- inode + its data
- files within same directory (namespace-based)

ex

mkdir: pick group
lots of free inodes, data blocks
low # other dir.

Create file: put data in same group as inode

large files: fill up entire group w/ 1 file

FFS rule: once > size
file
place next block in another group.

splitting a large file between groups is NOT bad

groups { 0 /a aaaaaaaaaa
1 /b
2

Amortization
seek cost is amortized over the transfer

0 /a/b aaaa bb
1 aaaa
2 aaaa

xfer: { 40 MB/s
10 ms (avg seek)

How much data should be transferred to amortize seek time

target efficiency → 90% efficiency

$\frac{40\ MB}{sec} \cdot \frac{1\ sec}{1000\ ms} \cdot 90\ ms \simeq 3.6\ MB \quad \frac{90\ ms}{10\ ms} = 90\%$

Other FFS stuff
Changed impl. without changing API
→ symbolic links
→ long file names > 8 char
- parameterization (e.g. rotation-aware)
- sub-blocks reduce internal fragmentation (buffer writes)

Understanding structure

mmap

(struct SB *) mmap

sb -> size

Crash consistency