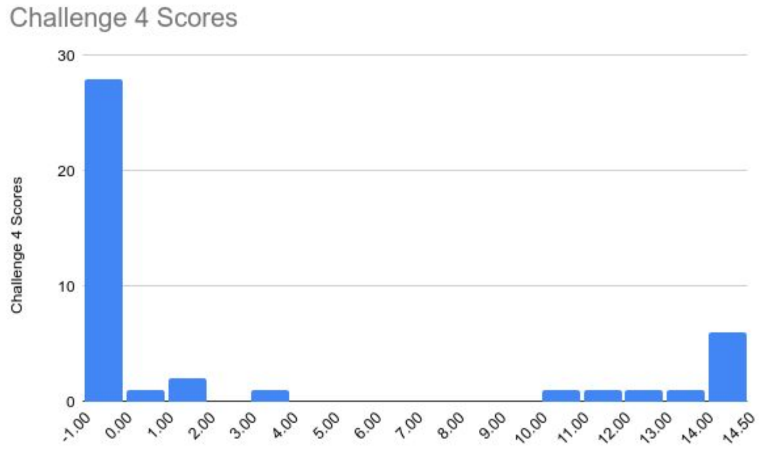
EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Logistics

- HW2 & HW3 grades are on Gradescope
  - Includes autograder and oral exam scores
  - Not included:
    - HW2 late penalty: -1 mark in the gradebook
    - HW2/HW3 extra credit will be accumulated for the end of the semester
- 1 Week left on HW4



Review on FS

What we covered so far

- Ugly interface - disk block read/write
- File Systems build abstracts of files, directories & directories on top of that...

File abstraction

- Operations: create, delete, read, write, seek, get attributes...
- What does a file system store:
  - some systems store records, tree..
  - unix like systems: a stream of bytes
    - name (e.g., .mp3), used by application to understand contents
- Attributes:
  - owner, permissions, length...
  - fstat to get user visible attributes

Directories

- Way to find/get to files...
- Original systems flat, one directory:
  - files in a cabinet
- Typically organize files into a tree:
  - different directories owned by different users
  - refer to by absolute or relative path name
- Operations: readdir, opendir, create

File Implementation

- Implementations:
  - contiguous layout
  - linked list
  - linked list in memory (e.g., DOS)
  - inodes
- What are the tradeoffs?

Tradeoffs

- contiguous layout

  + dense MD, few seeks

  - fragmentation, hard to extend files
- linked list

  + dense MD, no fragmentation

  - need to read sequentially
- linked list in memory (e.g., DOS)

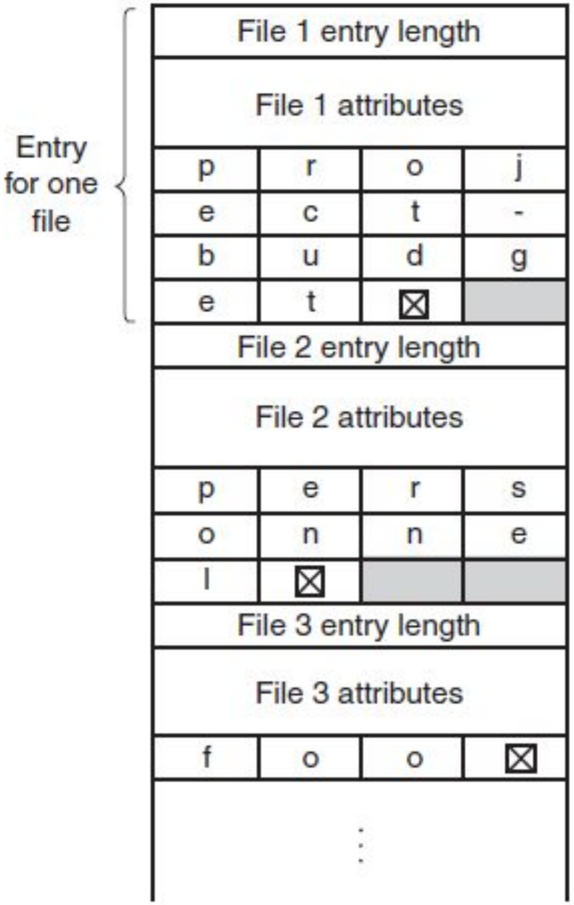
  + dense MD, no fragmentation, random access

  - massive memory requirement
- inodes

  + reasonable compromise, random access, sequential, density...

  - not optimal for anything

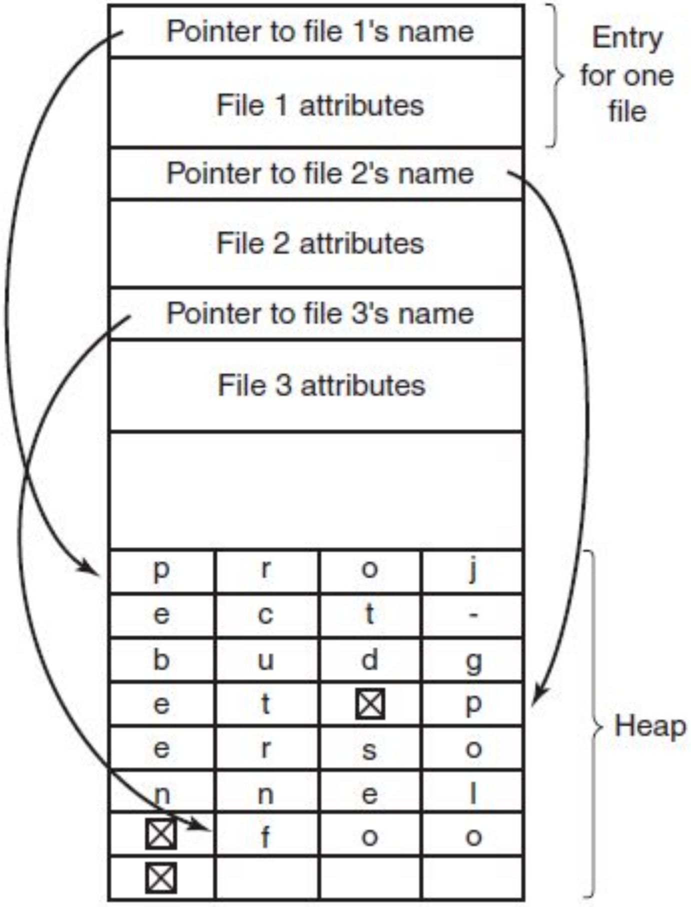
Directory implementation



(a)

- Could have a fixed sized entry, with max file name length
  - internal fragmentation
- Often organized as fixed sized information, followed by name, and filler to align struct
- Other options possible:
  - e.g., string names at end of block: avoiding filler
  - Hash table based on string name

Implementing Directories (2)



p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	p
e	r	s	o
n	n	e	l
☒	f	o	o
☒			

(b)

- Lots of other alternatives:
  - file names at the end
  - hash table in secondary data structure
- IMHO, probably not worth it:
  - more efficient on cache to pack
  - most file names small
  - most directories small
  - can re-organize in memory
  - ….
  - but sometimes for massive file names and massive directories performance is bad
  - OSes are compromises; but there are specialized FS, e.g., HPC

Referring to file

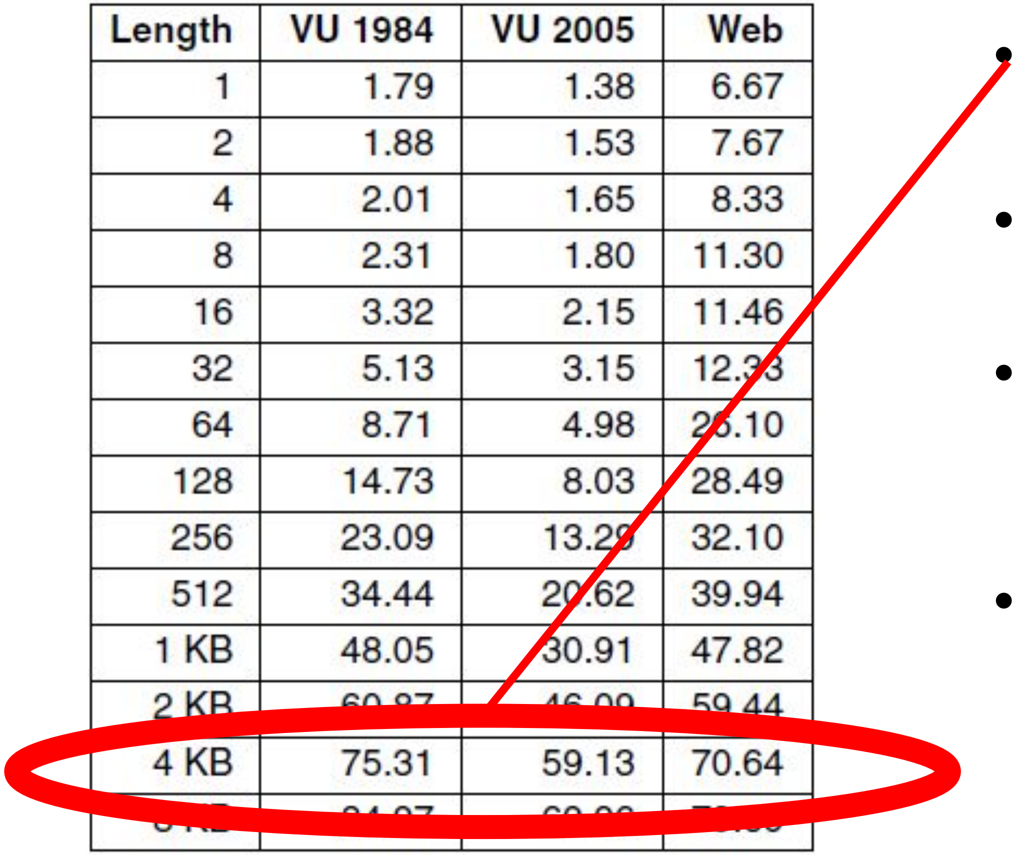
- Rather than putting attributes in the directory, just keep an "inode number" in it
- Now you can have multiple directories refer to same file: hard links
- Inode keeps track of number of links:
  - no delete, just an unlink
- Soft links - just put in the directory path of other location

Disk space management

How big should block be?

- Size of sector?
- Size of page?
- Size of track?

Need some data...

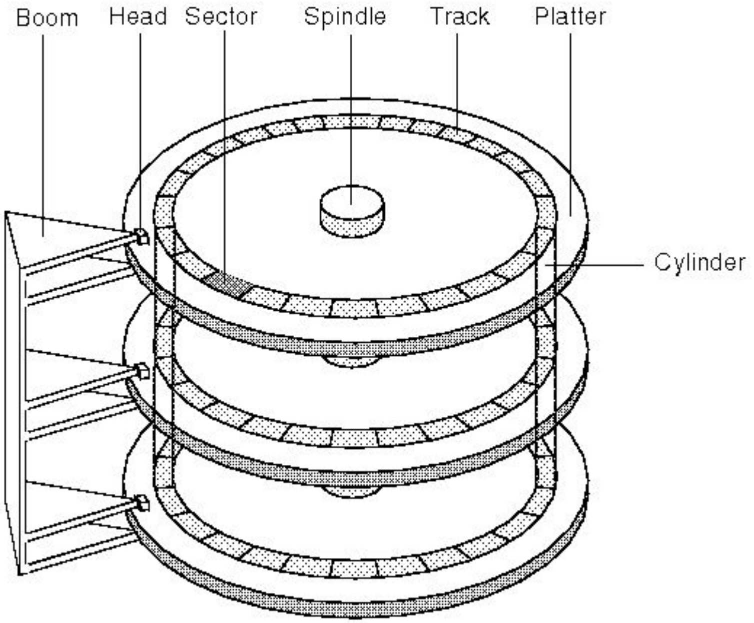


Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

- With 4KB 60-70% in a single block: so fast...
- But how much space is wasted?
- If all the data is in huge files, only waste end of last block...
- Turns out, in same study:
  - most space in huge files
  - with 4KB blocks, 93% of of disk space used by 10% biggest files..
- What about disk BW?

Disk Bandwidth

- Hard disk
  - several platters – disks (heads)
  - each platter has multiple tracks (start with 0)
  - each track has multiple sectors (start with 1)



Assume

- avg seek 5ms
- rotation 8ms
- avg Rotational latency = 8ms/2
- 1M track

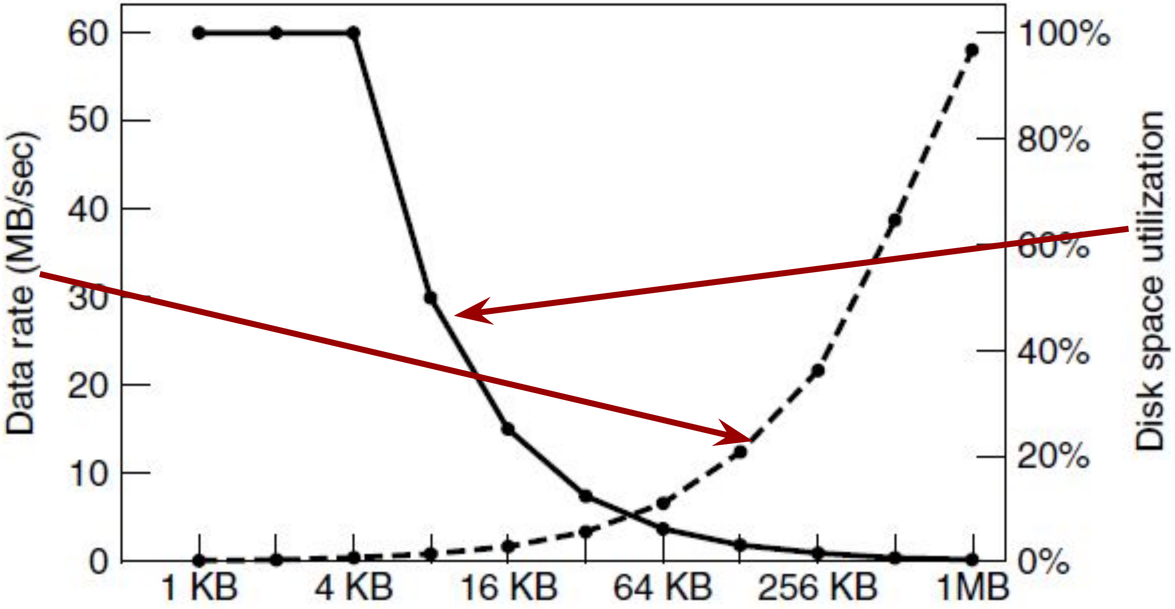
Transfer rate:

1M/8ms = 125MB/s

With 1 seek:

1M/(8+5+4) = 59MB/s

Tradeoff disk space vs performance



- Assume all files 4KB (pessimistic) utilization goes down with larger block
- Since read time dominated by seek and rotational latency, larger blocks result in much faster data rate

How big should block be?

- Size of sector?
- Size of page?
- Size of cylinder?

In modern systems, consider SSD and Page cache

Smaller - Less Fragmentation, Worse bandwidth

Its not that simple

- FS can put separate blocks on same track
- OS can prefetch, have many blocks outstanding... so get good transfer rate even with small blocks
- Device can prefetch into track buffer
- Some file system use variable length blocks, called extents:
  - logical block
  - starting physical block
  - number of physical blocks

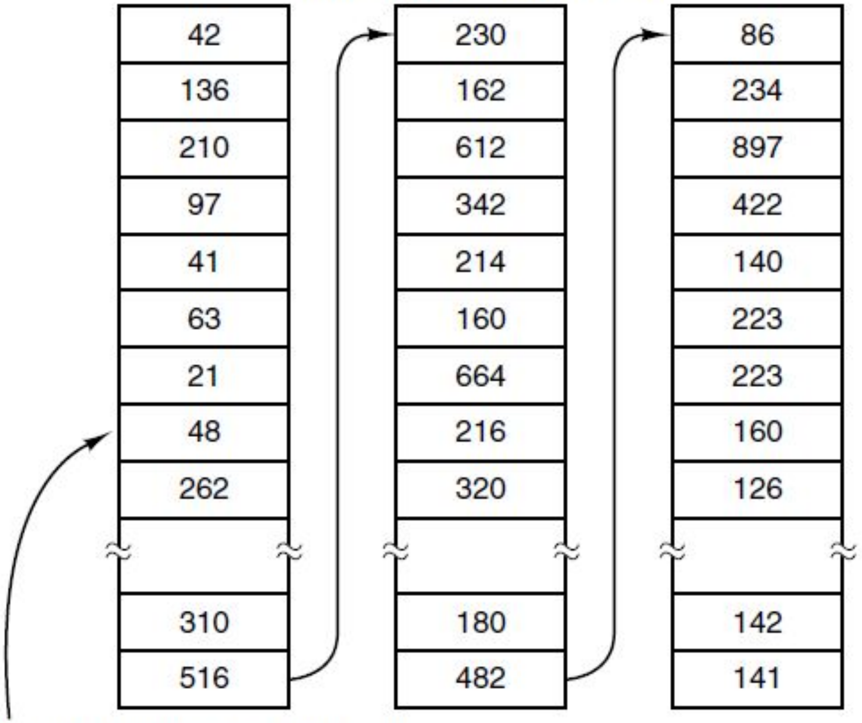
Block size

- Historically FS have chosen block sizes in the range of 1K-4K
  - 4K matches minimum page size, modest space overhead
  - Disk meta data… increase with smaller block size
- With slow modern disks, larger blocks or extents probably make sense
- With SSD, caches, smaller blocks probably make sense
- Depends on your workload, how big files are, required transfer rate, device

How do we keep track of free blocks?

Linked list

Free disk blocks: 16, 17, 18

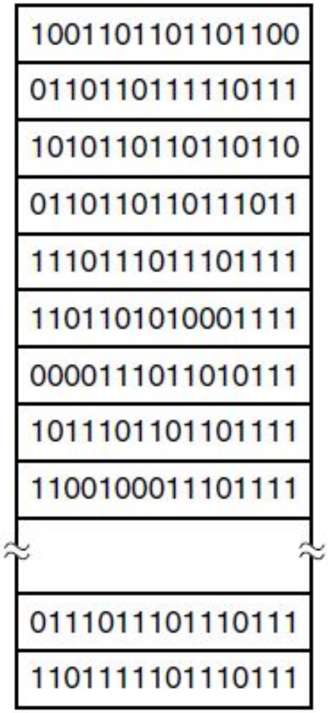


A 1-KB disk block can hold 256
32-bit disk block numbers

(a)

- For 1TB disk, this means 1M blocks to hold free list
- Compresses to small number of blocks when disks full
- Huge initialization cost
- Can maintain ranges, rather than individual blocks

Bit vector



```
1001101101101100
0110110111110111
1010110110110110
0110110110111011
1110111011101111
1101101010001111
0000111011010111
1011101101101111
1100100011101111

0111011101110111
1101111101110111
```

A bitmap

(b)

- 1 bit per block, for 1TB disk, this means 130K blocks to hold free inf
- Still large initialization cost, but easier to batch (e.g., write all zeros)
- Easier to find contiguous regions of disk
- Can allocate from one block at a time, keeping disk head in one region disk
- When disk fragmented, each block may have just few bits available - expensive to search

Tradeoff

- Linked list nice when few blocks free
- Bit vector nice when lots of blocks free
- Bit vector nice for extents...
- In the Hurricane File System:
  - linked list for fast lookup on single block requests
  - bit vector for extents
  - combined to find if blocks are free...

Review of FS basics…

Review of FS basics...

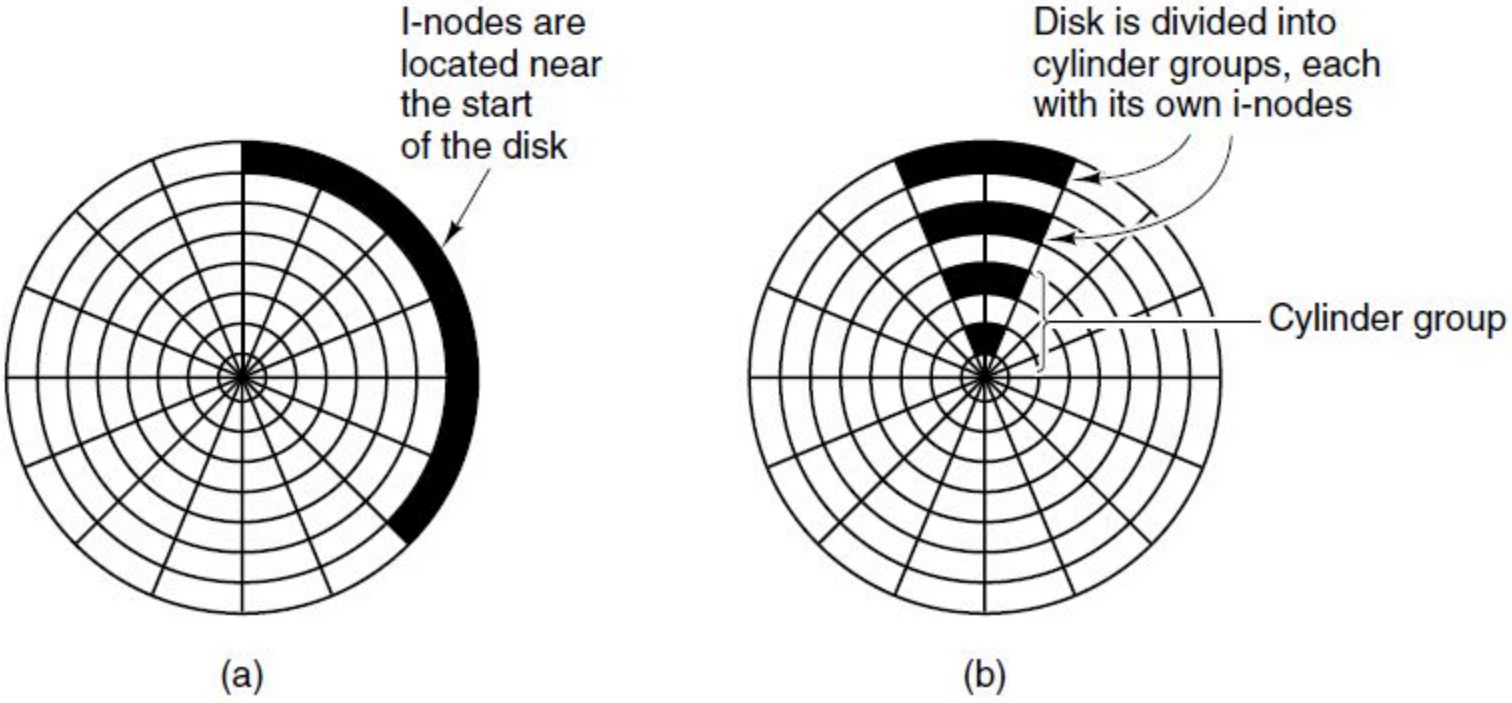
- Abstractions: files/directories
- Implementation/tradeoffs of these abstractions
- Tradeoffs block size
- Organization of free information

Discussion

- There is no right answer, OS are a compromise between:
  - needs different applications
  - HW characteristics
- Specialization makes a huge difference
- Different file systems can/are optimized for different usage, but…
  - often time we have a wide diversity of applications using data
  - can only partition your disk so far...

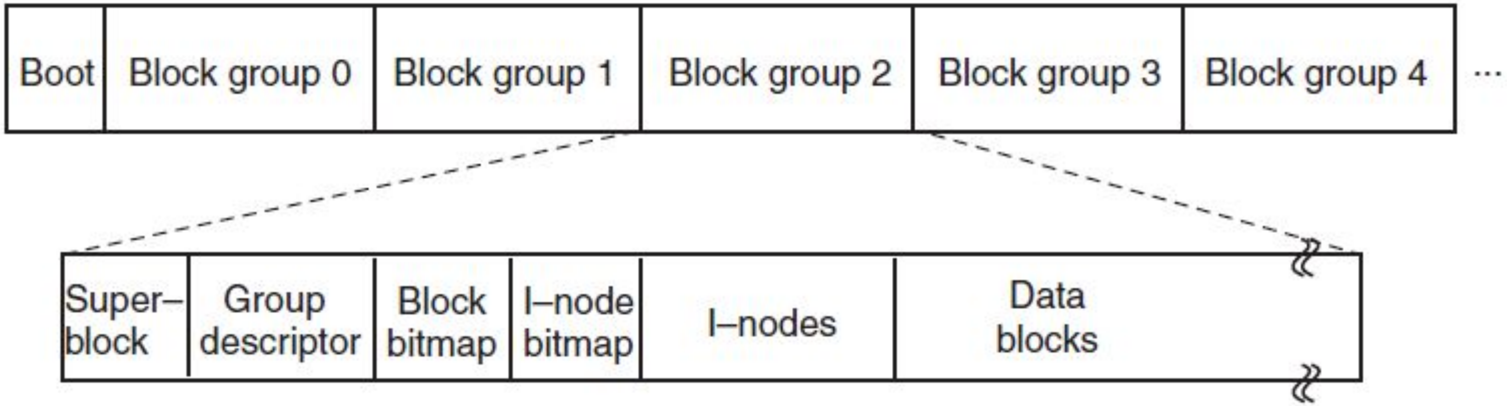
Example EXT2 file system

A bit about disks...


I-nodes are located near the start of the disk
Disk is divided into cylinder groups, each with its own i-nodes
Cylinder group
(a)
(b)

- Putting inodes in fixed place great if you want to snap up a bunch, and if cache hit likely (a)
- Seeks are expensive.... If we put near the data can avoid...

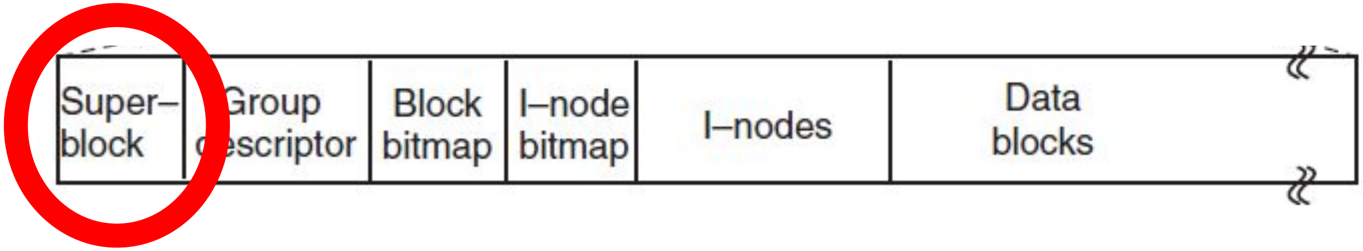
Disk Layout



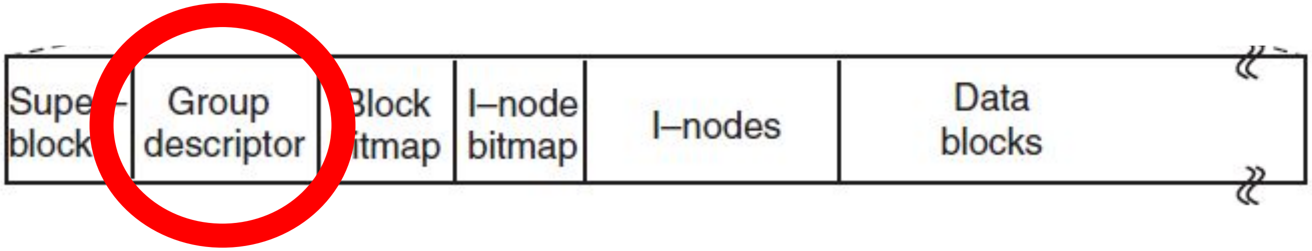
Disk divided into a number of Block Groups distributed across disk

Block Groups

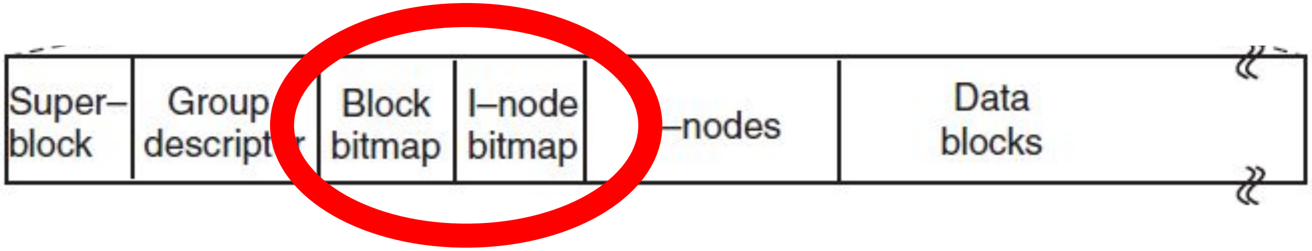
- Used to get cylinder locality.
- Policy attempts to
  - Spread directories equally across block groups
  - Allocate inodes in the same block group as directory (consider ls -l)
  - Allocate disk blocks in the same group as inode
- When disk becomes fragmented can use inodes/disks from any group



- The root of the entire file system. Contains:
  - number of inodes
  - number of disk blocks
  - magic number - type of file system
  - revision number - newer FS can mount old
  - first inode - root directory
  - block size - can be 1K - 8K
- replicated for redundancy to multiple block groups. Why?

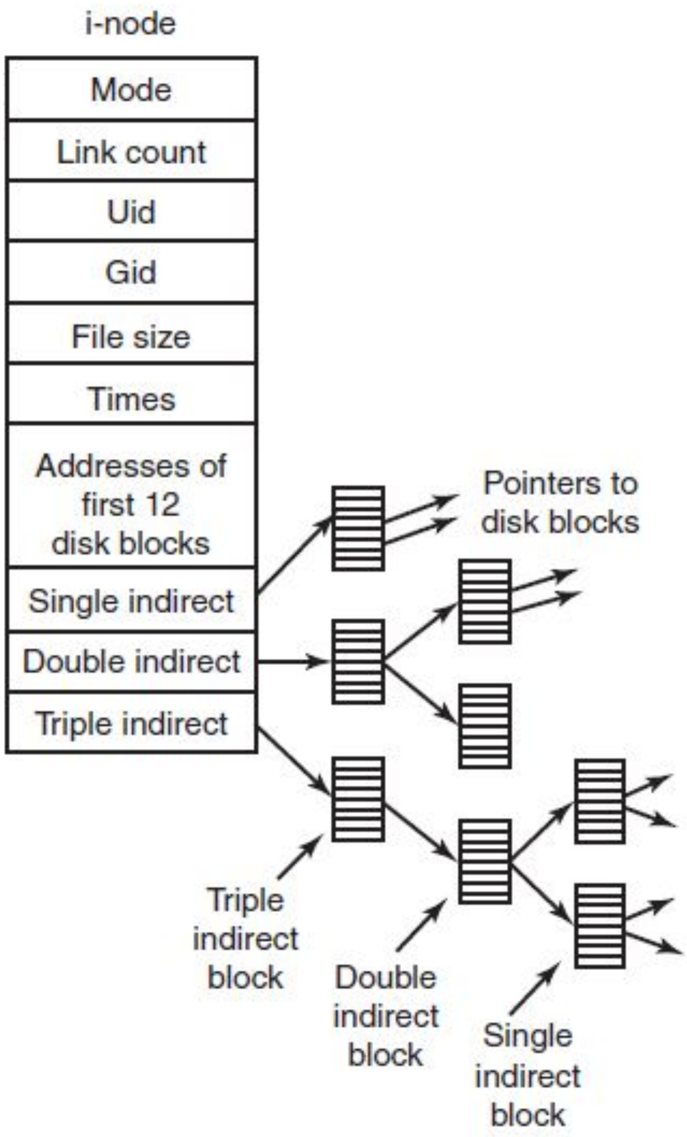


- location of the bitmaps:
  - inodes, free blocks
- number of directories in the group
  - used to balance across groups
- inode table starting block
- free block count
- free inode count



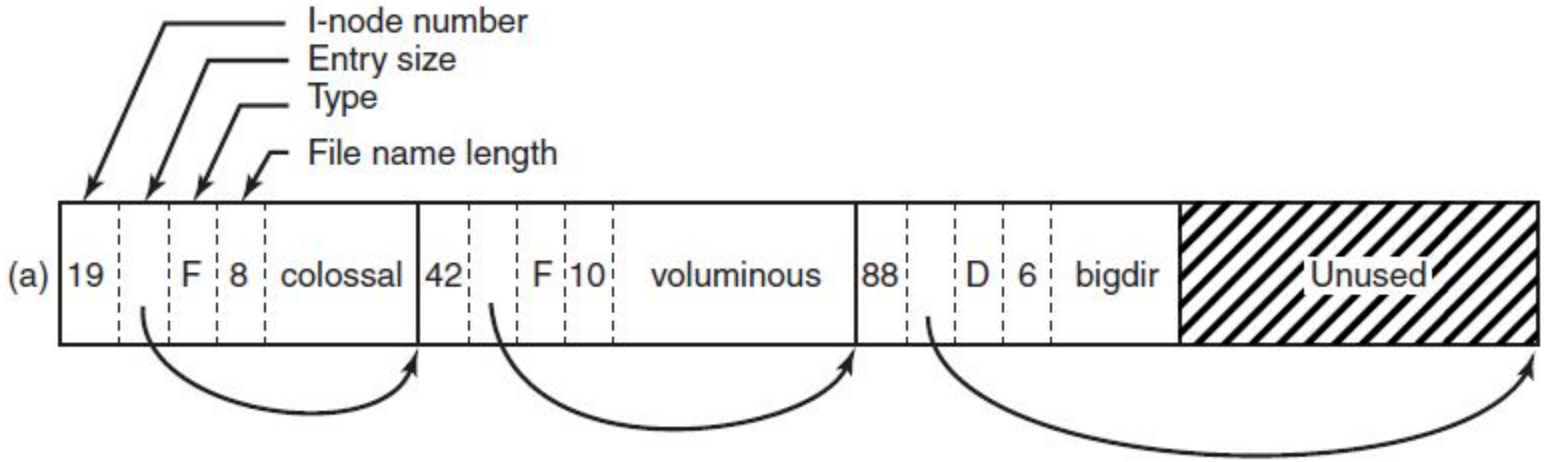
- bitmaps are one block long:
  - with 1K blocks 8K blocks or 8M
  - with 4K blocks 32K blocks or 132M
  - with 8K blocks 64K blocks or 536M

Inode



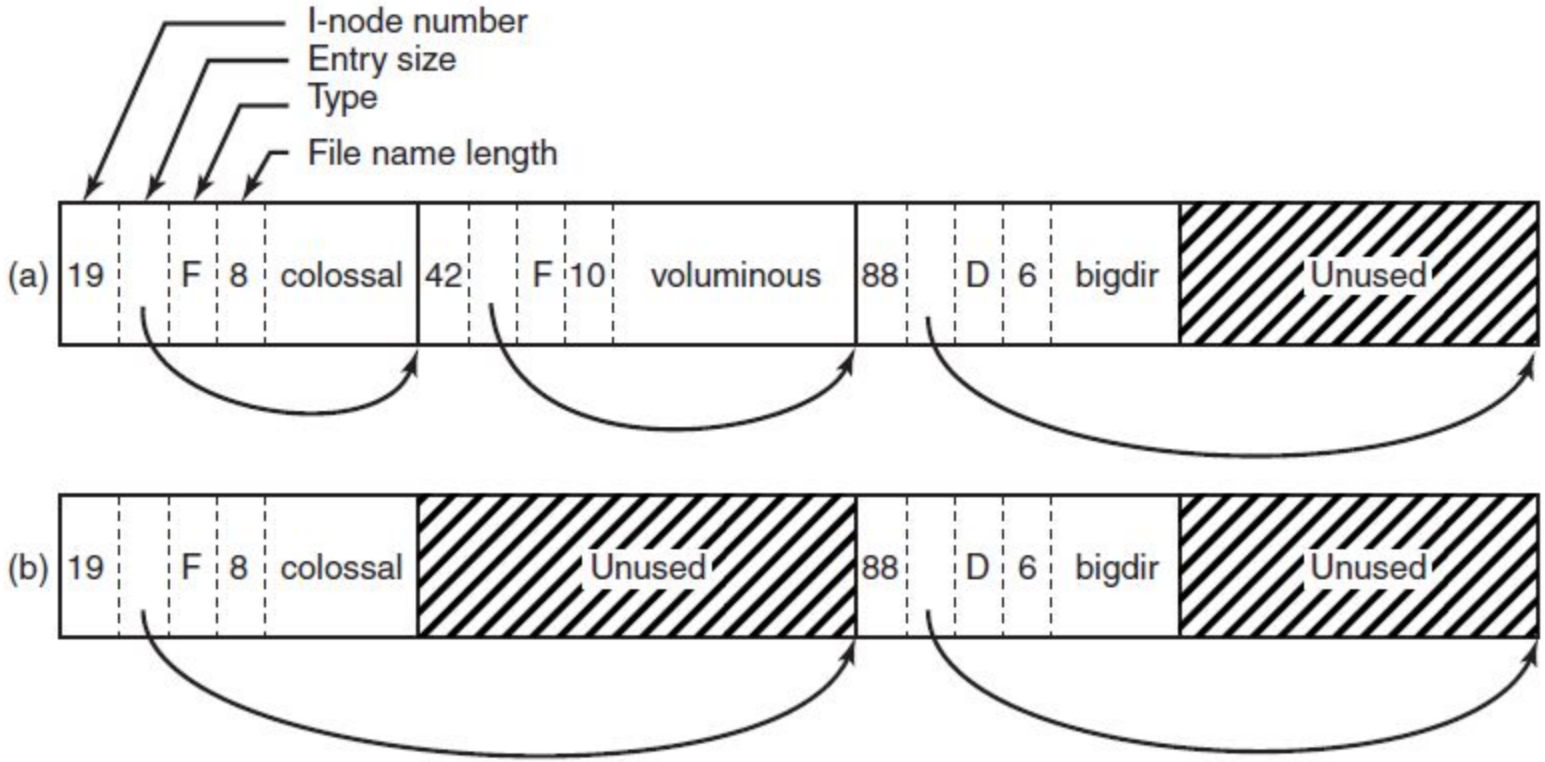
- standard linux mode/uid… times
- 12 direct, double indirect…
- max file size:
  - 1KB block - 16GB
  - 2KB block - 256GB
  - 4KB block - 2TB

Directory structure



- entries unsorted
- type: file, directory, …
- file name length for file, entry size includes padding

After free



- space converted into padding

FS Metadata consistency

Consider removing a file

1. Remove file from its directory.
2. Release i-node to the pool of free i-nodes.
3. Return all disk blocks to pool of free disk blocks

But… everything cached… what could go wrong?.

What could go wrong

Imagine if system crashes:

1. After freeing inode but before freeing disk blocks

2. After freeing disk blocks, before freeing inode

3. Similar problems on writing ….

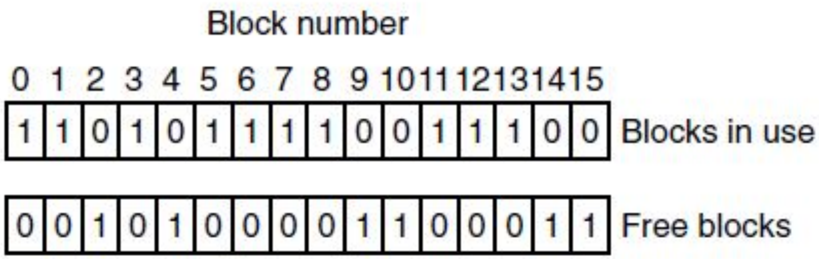
Original approach - fsck

fsck - walk through file inodes, to check that:

1. each block either free or used in some file once
2. all inodes in a directory routed from top
3. all ref counts in inodes match directory refs
4. all freed disk blocks not referred to by file

Maintains two tables

1. how many times block referenced in file
2. how many times block on free list/bitvector



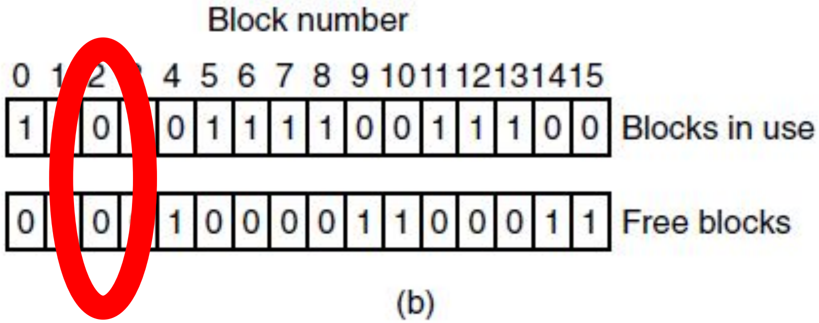
Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

One in just one list... a consistent file system

Maintains two tables

1. how many times block referenced in file
2. how many times block on free list/bitvector



Block number

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	[illegible]	0	[illegible]	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	[illegible]	0	[illegible]	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

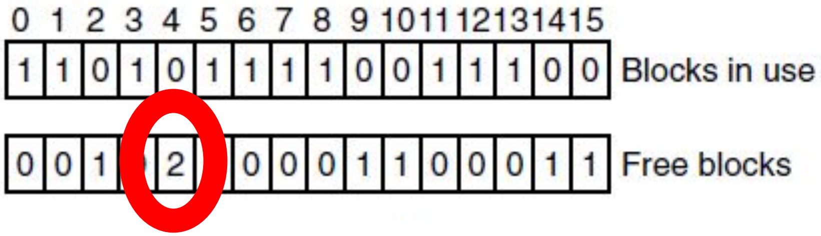
(b)

Block lost. Recovery?

fsck can just free

Maintains two tables

1. how many times block referenced in file
2. how many times block on free list/bitvector



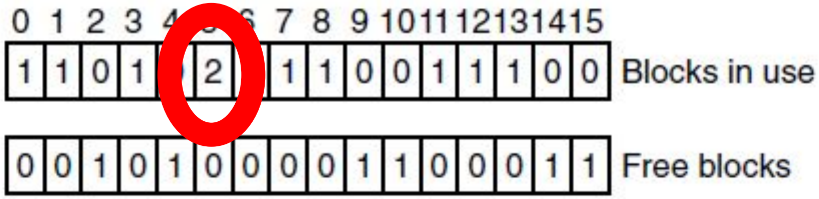
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	[illegible]	0	0	0	1	1	0	0	0	1	1	Free blocks

Block is free twice. Recover?

rebuild free list...

Maintains two tables

1. how many times block referenced in file
2. how many times block on free list/bitvector



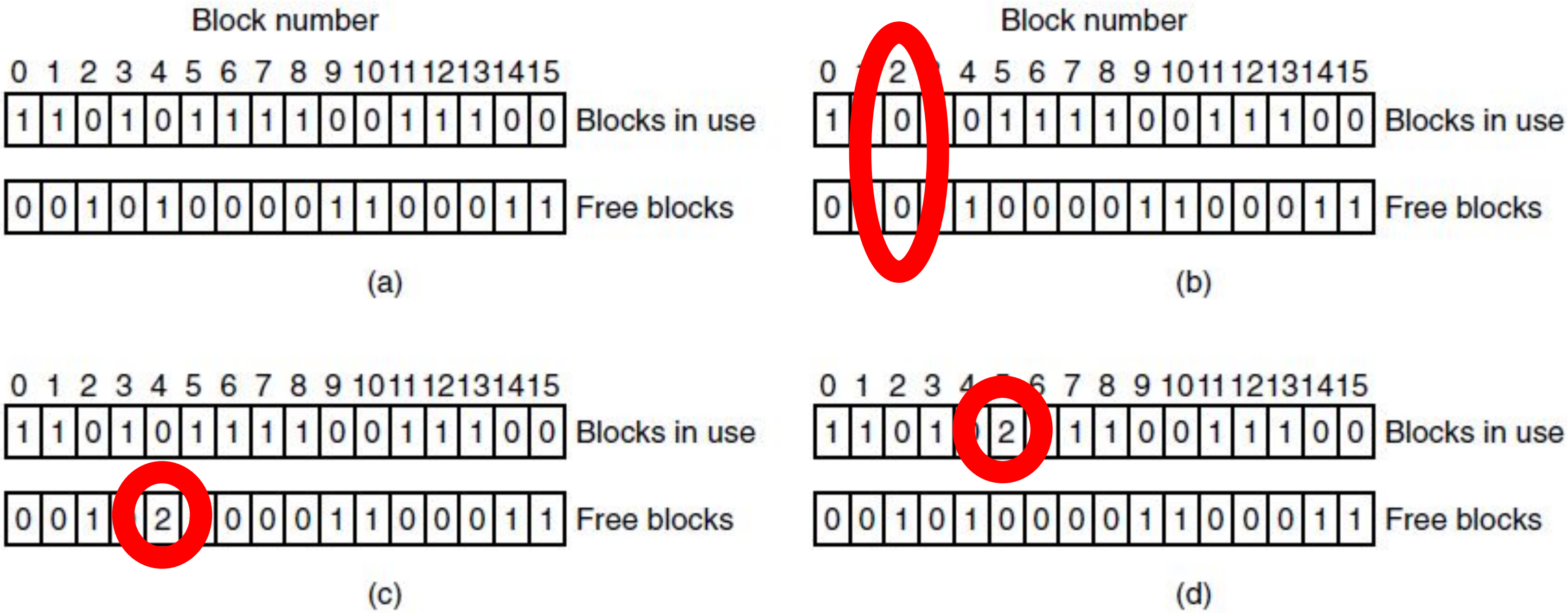
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	[illegible]	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Two files point to the same block.

Recovery?

Allocate a new block. Tell user

e.g. counter list used/free



Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a)

Block number

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	[illegible]	0	[illegible]	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	[illegible]	0	[illegible]	1	0	0	0	0	1	1	0	0	0	1	1

(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	[illegible]	2	[illegible]	0	0	0	1	1	0	0	0	1	1

(c)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	[illegible]	2	[illegible]	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(d)

This is expensive, can take many hours, system could crash again.

File system checking takes hours… is there something better we can do?

Failures happen

- We need to accept that data will be lost unless we implement a write through cache and block process.
- Sometimes we will get disk errors… so fsck will always be needed, but...
- Can we get have consistent on-disk state not have any corruption from system crashes/power failures?
  - (some orphaned state may be okay for a while)

Approach 1: synchronous ordered writes meta-data

Logically order writes to preserve dependencies & sync write to disk, e.g.:

a. don’t free blocks until released inode
b. "create" allocates inode before directory entry

This can be incredibly expensive, nobody does it anymore.

You could throw HW at the problem; e.g. NVRAM

Approach 2: soft update

1. Maintain dependency of metadata in memory
2. Data blocks can be written at any time.
3. When metadata is being written, done asynchronously but ordered:
    a. e.g., inode must reach disk before directory

Disk is in a crash consistent state, data may be lost, but meta-data is consistent (mostly)

Approach 3: Journaling/Log

Technique originally used for DB:

- log separate from FS metadata with an entry for every change
- if failure occurs, re-play log of complete operations; abort partial ones

Great comparison here; journaling won, don't really know why.

Key is that logged operations idempotent, i.e.can repeat without harm. Is free block A idempotent if:

1. you have a linked list of free blocks NO
2. you have a bit vector of free blocks YES

Logging in ext3/ext4

- backward compatible with ext2 disk format
- ext3 adds journaling: three options
  - journal - metadata and data
  - writeback - metadata only
    - equivalent to super fast fsck
  - ordered - metadata only *after* data written disk
    - FS never contains stale data
- Journal can be on same or seperate disk
- Goes through Journaling Block Device
  - can’t write journal through ext3/4 since would then be journaled

ext4 continued

- log records grouped into atomic operations, either all applied or none
- ext4 supports extents - e.g., 128 MB of contiguous disk blocks in a single extent
    - logical blk, num blocks, physical block
- ext4 increases:
    - maximum file size from 16 GB to 16 TB
    - the maximum file system size to 1 EB (Exabyte).

Log-structured FS

The best research is based on hypothesis.

Imagine if the memory huge? With large file caches, how would this change the operations that get to disk?

FS are designed to optimize for reads (e.g., extent based)

Log-structured FS idea

- Organize entire disk into a massive log
- Write out segments, with inodes, directories, blocks as a big (e.g., 1MB) write.
- Maintain map of inodes to location in last segment.
- Garbage collect segments with small amount of data

Trade offs

- The good:
  - Recovery can be very fast.
  - Can easily integrate snapshots
  - Writes are huge, very efficient on disk
- The bad:
  - Reads can be slower, data may be scattered over disk; but most files written at the same time.
- Is now used in SSDs at bottom layer to even wear over the entire disk.

A fun story

- For my PhD built Hector/Hurricane.
- Based on VFS, built OO layer, and a NFS, but people wanted very high speed disks…
- Developed I/O board and wrote HFS
- System crashed every day or so (it was the HW), so wrote a fsck for HFS, all was fine… until the file system grew and grew… took hours to run…
- The log structured FS came out, and eureka

My solution

- All data/meta data written to new locations on disk
- Two superblocks, alternate between them.
- Superblock written after all intermediate blocks/metadata
- System just restored to the last superblock; always a consistent (perhaps old) version of the FS
- Recovery time went from 2 hours to < 1 second

New research

- We are developing a new file system that uses S3 as backed
  - immutable object storage
- Very similar to log structured FS, each S3 object is similar to a segment
- Use local SSD as a cache...

Performance of local SSD, disaggregated storage, geographical distribution...

Other stuff

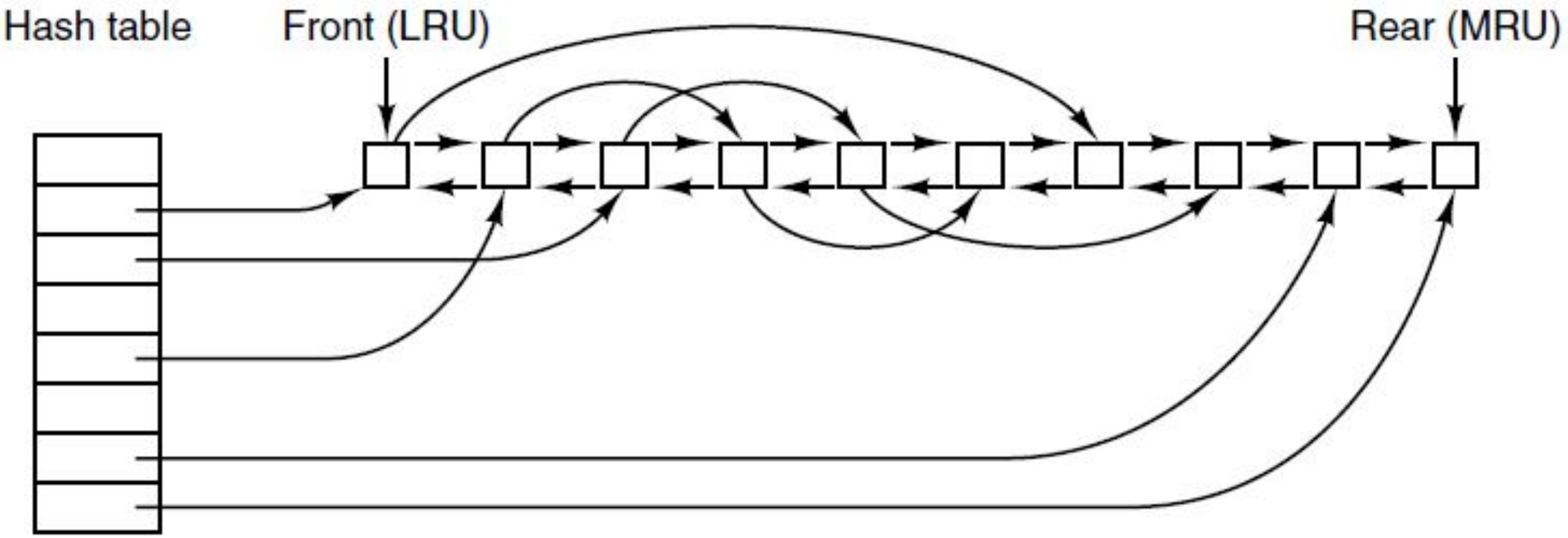
Backup: the book talks about...

- There are two reasons for backup:
  - Disaster recover
  - Dealing with Human stupidity
- Book spends several pages, describing important OS concepts:
  - logical versus physical
  - incremental versus full
  - offsite/security

NO!!! Computers cattle not pets

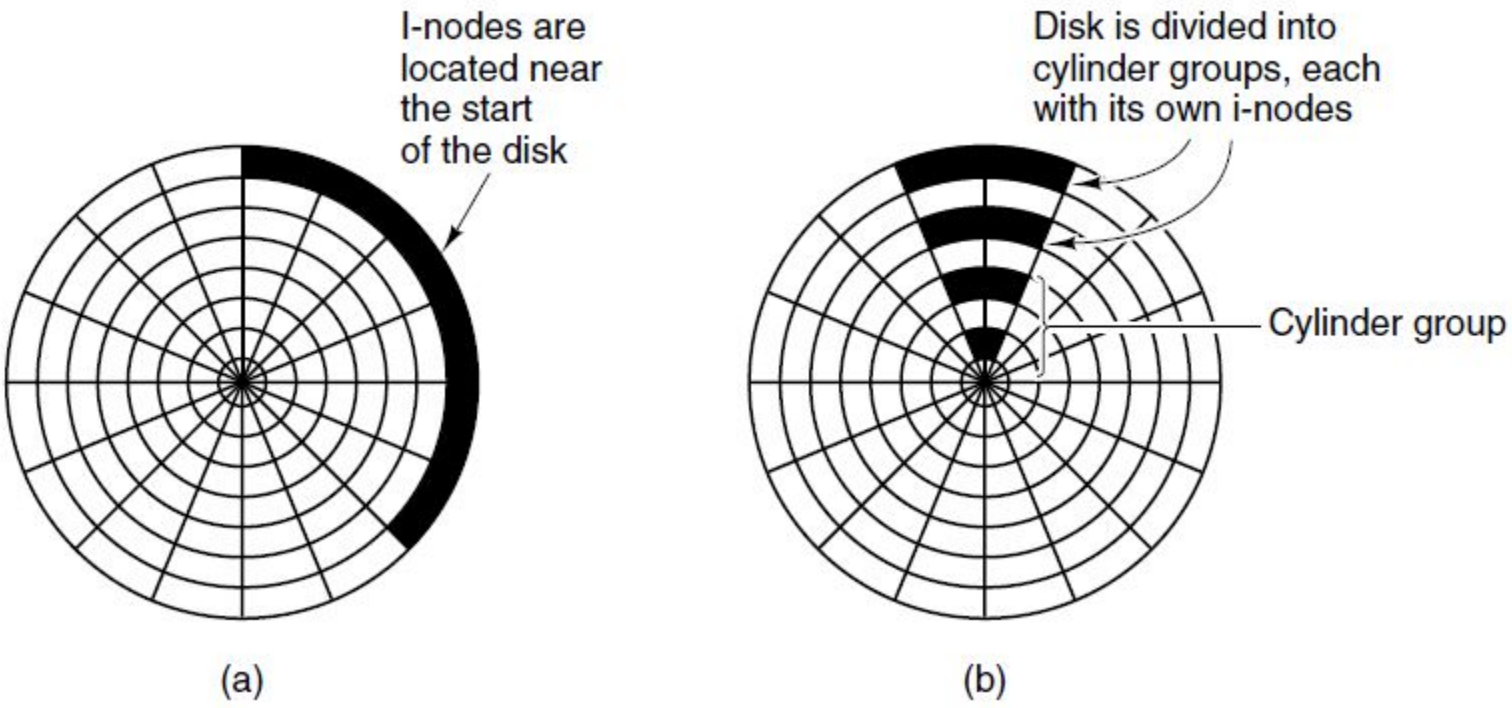
- data stored in DB with DB specific policies
- passwords in cloud password manager
- code in github
- music in cloud
- configuration is code...

Page cache



- We discussed when discussing MM, all modern OSes have a unified cache anonymous/file
- key point is that read/write references allow us to keep LRU
- Book sayed write out meta-data rapidly important… see log/soft update discussion

Thinking about the disk


I-nodes are located near the start of the disk
Disk is divided into cylinder groups, each with its own i-nodes
Cylinder group
(a)
(b)

- Putting inodes in fixed place great if you want to snap up a bunch, and if cache hit likely (a)
- Some file systems put in cylinder near blocks (or segments)