Operating Systems, fall 2002

Local File Systems in UNIX

Lior Amar,
David Breitgand
(recitation)
www.cs.huji.ac.il/~os

Classical Unix File System

- Traditional UNIX file system keeps I-node information separately from the data blocks;
- Thus accessing a file involves a long seek from the I-node to the data;
- Since files are grouped by directories, but I-nodes for the files from the same directory are not allocated consequently, many non-consecutive block reads should be done to access files in large and nested directories.
  - Refer to our example from the last class.
- Allocation of data blocks is sub-optimal because often next sequential block is on different cylinder

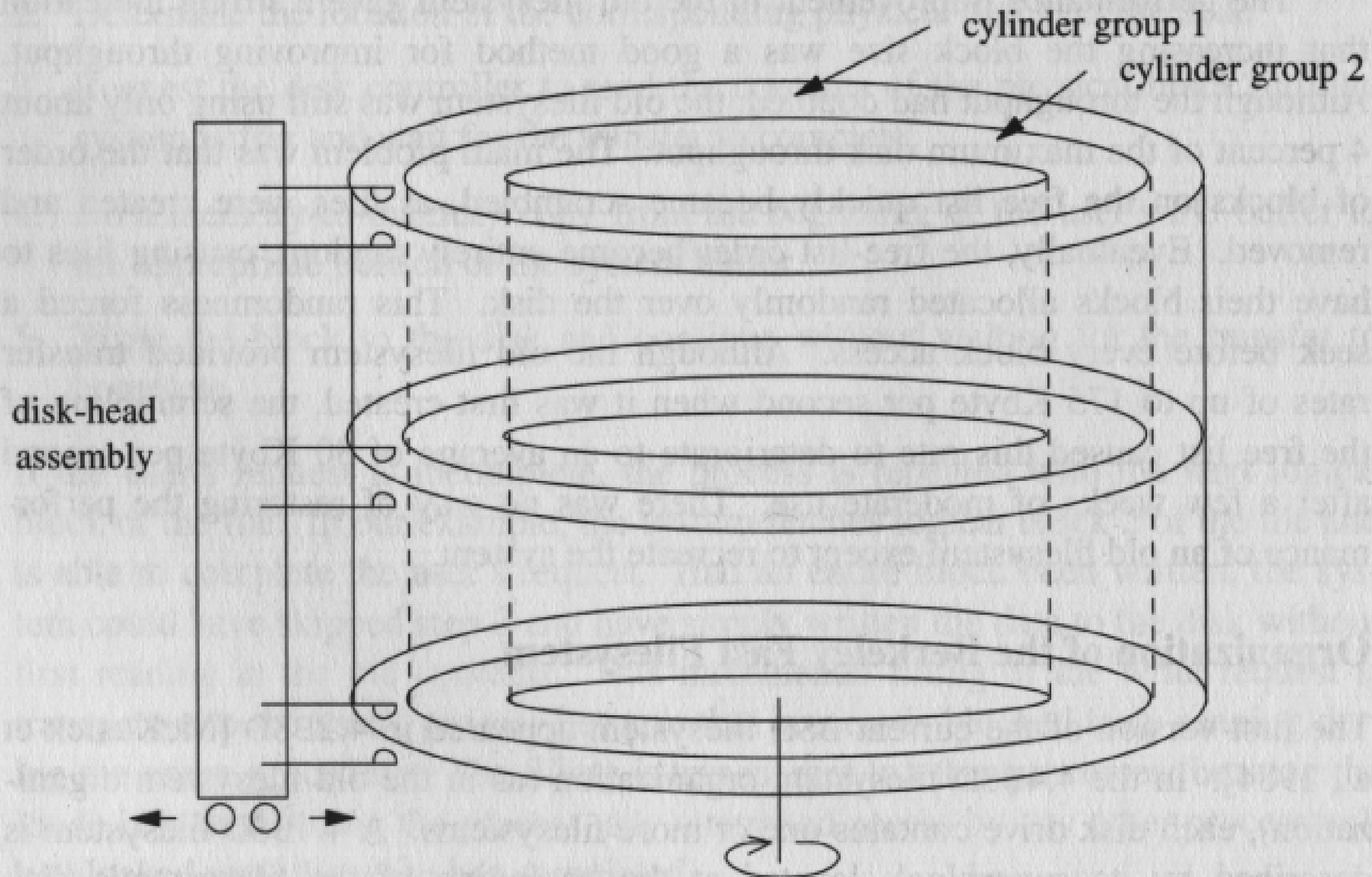
Classical Unix File System

- Allocation of data blocks is sub-optimal because often next sequential block is on different cylinder;
- Reliability is an issue, cause there is only one copy of super-block, and I-node list;
- Free blocks list quickly becomes random, as files are created and destroyed
- Randomness forces a lot of cylinder seek operations, and slows down the performance;
- There was no way to “de-fragment” the file system;
- Small block size (512-1024)
- 4% of maximal disk throughput.

Berkeley Fast File System

- Minimal block size is 4096 bytes
- Allows to files as large as 4G to be created with only 2 levels of indirection;
- Disk partition is divided into one or more areas called *cylinder groups;*

Figure 8.2 Layout of cylinder groups.



Cylinder Group

- Each cylinder group is one or more consecutive cylinders on a disk;
- Each cylinder group contains a redundant copy of the super block, and I-node information, and free block list pertaining to this group.
- Each group is allocated a static amount of I-nodes;
- The default policy is to allocate one I-node for each 2048 bytes of space in the cylinder group on the average;
- The idea of the cylinder group is to keep I-nodes of files close to their data blocks to avoid long seeks; also keep files from the same directory together.

Cylinder Group

- Where to place the information about each cylinder group?
- At the beginning of each group?
- BFFS uses varying offset from the beginning of the group to avoid having all crucial data on one plate;
- The offset between the beginning and the information is used for data blocks.

Block Size Considerations

- As block size increases the efficiency of a single transfer also increases, and the space taken up by I-nodes and block lists decreases;
- However, as block size increases, the space is wasted due to internal fragmentation.

Block Size Considerations

- Measured 12 million files;
- 1000 file systems
- Total size: 250 G
- Mean file size: 22K
- Median: about 2K
- How to accommodate smaller files with larger blocks?
- However all this was back in 1993…

Table 8.2 Amount of space wasted as a function of block size.

Percent total waste	Percent data waste	Percent inode waste	Organization
0.0	0.0	0.0	data only, no separation between files
1.1	1.1	0.0	data only, files start on 512-byte boundary
7.4	1.1	6.3	data + inodes, 512-byte block
8.8	2.5	6.3	data + inodes, 1024-byte block
11.7	5.4	6.3	data + inodes, 2048-byte block
15.4	12.3	3.1	data + inodes, 4096-byte block
29.4	27.8	1.6	data + inodes, 8192-byte block
62.0	61.2	0.8	data + inodes, 16384-byte block

Solution

- Divide block into *fragments*;
- Each fragment is individually addressable;
- Fragment size is specified upon a file system creation;
- The lower bound of the fragment size is the disk sector size;
- Example:

Figure 8.3 Example of the layout of blocks and fragments in a 4096/1024 filesystem. Each bit in the map records the status of a fragment; a "-" means that the fragment is in use, whereas a "1" means that the fragment is available for allocation. In this example, fragments 0 through 5, 10, and 11 are in use, whereas fragments 6 through 9 and 12 through 15 are free. Fragments of adjacent blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6 through 9 cannot be allocated as a full block; only fragments 12 through 15 can be coalesced into a full block.

bits in map	----	--11	11--	1111
fragment numbers	0-3	4-7	8-11	12-15
block numbers	0	1	2	3

File System Consistency

- Due to various failures, e.g., hardware failures, or power failures, file system can potentially enter an *inconsistent state;*
- Metadata modifications (e.g., directory entry update, I-node update, free blocks list update, etc.) should be *synchronous*
- *Why can't we relay on cache for these operations?*

File System Consistency

- A specific sequence of operations also matters.
- We cannot guarantee that the last operation does not fail, but we want to minimize the damage, so that the file system can be more easily restored to a consistent state.
- Consider *file creation*:
  - write directory entry, update directory I-node, allocate I-node, write allocated I-node on disk, write directory entry, update directory I-node.
  - But what should be the order of these synchronous operations to minimize the damage that may occur due to failures?

File Creation

Correct order is:

1) allocate I-node (if needed) and write it on disk;
2) update directory on disk;
3) update I-node of the directory on disk.

What should it be for file deletion?

Where is the Bottleneck?

- **Synchronous operations for updating the meta-data:**
  - Should be synchronous, thus need seeks to I-nodes;
  - In BFFS is not a great problem as long as files are relatively small, cause directory, file data blocks, and I-nodes should be all in the same cylinder group.
- **Write-back of the dirty blocks:**
  - Real problem, because the file access pattern is random; different applications use different files at the same time, and the dirty blocks are not guaranteed to be in the same cylinder group.

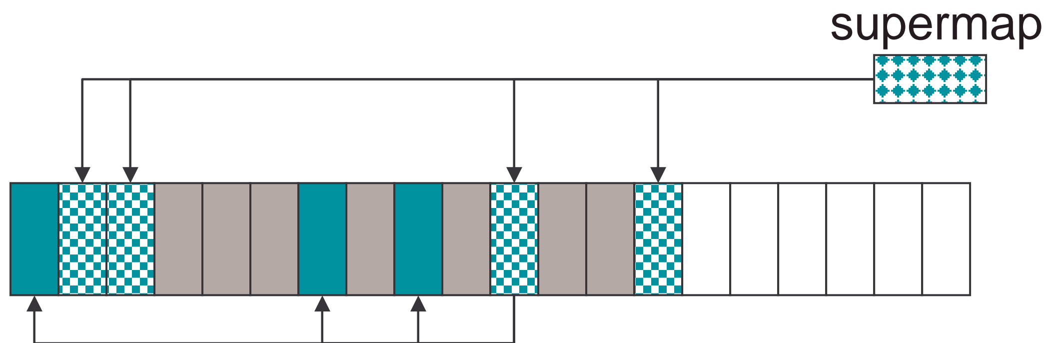
Log Structured File System

- Ousterhout & Douglis (1992)
- Caching is enough for good read performance
- Writes is the real performance bottleneck
  - writing-back cached user blocks may require many random disk accesses
  - write-through for reliability denies optimizations
    - logging solves the problem for metadata

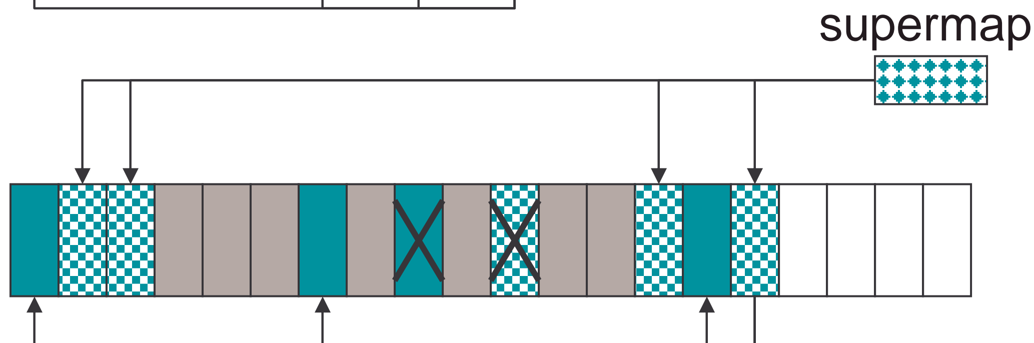
Log Structured File System

- The idea: everything is log
- Each write - both data and control - is appended to the sequential log
- The problem: how to locate files and data efficiently for random access by Reads
- The solution: use a floating file map

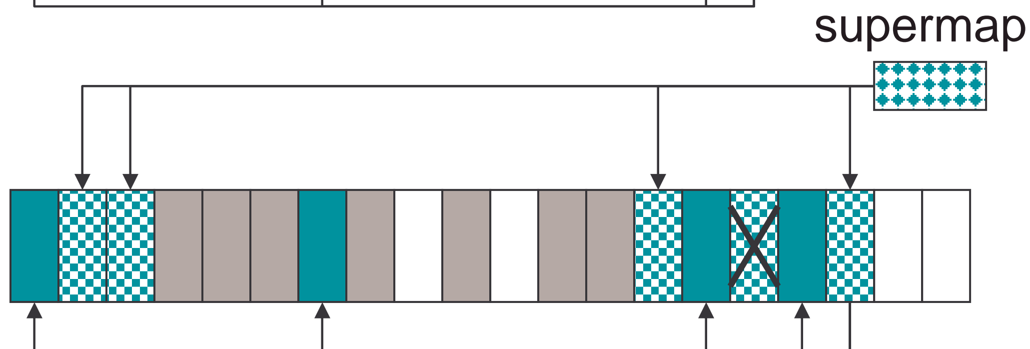
Log structured file system



Before



After block change



After block addition