Operating Systems

File Systems

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2022

References

The content of these lectures is inspired by:

- The lecture notes of Prof. David Mazières.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

Goals of the lecture

- Get a global picture of the challenges associated with file systems implementation
- Study a complex software engineering problem
- See how the characteristics of HDDs can be taken into account in the software design
- Understand the main concepts used in the design of famous file systems (FAT, FFS, ext2, ext3, ext4, NTFS, btrfs, . . .)

Included in this lecture

Basic concepts associated with a file system

- Data blocks
- Inodes
- Bitmaps
- Extents

Advanced software engineering techniques

- Multi-level indexes
- Locality (to improve efficiency)
- Journaling (to deal with failures)
- Copy-on-write

Agenda

Agenda

Introduction

File system implementation

The Fast File System

Dealing with failures

Log-structured file systems

Writing blocks of data to disk is not that much fun

Disks provide a means to store data (and programs) reliably.

How to organize the data?

2 key abstractions

- Files: Array of bytes that can be read and written – associate bytes with a name.
- Directories: A list of files and directories – associate names with each other.

Operations on files

System calls

- *open()*: create/open a file
- *read()/write()*: read/write an opened file sequentially
- *close()*: close an opened file
- *lseek()*: move to an offset in a file
- *fsync()*: force write of dirty data to disk
- *rename()*: change name of a file
- *stat()*: get metadata of a file
- *link()*: associate a file to a directory
- *unlink()*: delete a file

About directories (UNIX)

Structure

- A tree structure with "/" being the root directory
- By default a directory includes 2 entries:
  - **.** : a reference to itself
  - **..** : a reference to the parent directory

System calls

- *mkdir()*: create a directory
- *rmdir()*: delete a directory – all files are unlinked first.
- *opendir()/readdir()/closedir()*

Disks versus memory

- Disk provide persistent storage
  - Data won't go away after reboot
- Disks are much slower than memory
  - Latency: ~ 50 ns for memory vs ~ 8 ms for disks (5 order of magnitude)
  - Throughput: > 1 GB/s for memory vs ~ 100 MB/s for disks (1 order of magnitude)
- Capacity of disks is usually much larger

Agenda

About file systems

Introducing comments

- All implemented in software
- One of the most complex part of OS
  - Active research topic
- Plenty of FS implementations

Purpose of a file system

- Translate name+offset to disks blocks
- Keep track of free space

About the tanslation of logical to physical location

We were solving similar problems with virtual memory.

What is easier with FS:

- CPU time is no big deal (compared to disks performance)
- Simpler access pattern (sequential access)

What is more complex with FS:

- Each layer of translation = potential access to disk
- Range is very extreme: Many files <10 KB, some files many GB

Observations related to performance

- FS performance is dominated by the number of disk accesses
  - Say each access costs $\sim$10 milliseconds
  - Touch the disk 100 extra times = 1 *second*
- Access cost dominated by movement, not transfer:
  - *seek time* + *rotational delay* + *bytes*/*diskBW*
  - 1 sector: 5ms + 4ms + 5μs ($\approx 512\,\mathrm{B}/(100\,\mathrm{MB/s})$) $\approx$ 9ms
  - 50 sectors: 5ms + 4ms + .25ms = 9.25ms
  - Can get **50x more data for only $\sim$3% extra 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

File system implementation

What we need to define and understand:

- The data structures of the file system
  - How the data and the metadata are organized
- The access methods
  - How the data and metadata are accessed during a call to open/read/write/...

Blocks

Blocks

- Disks are divided into blocks of fixed size
- Typically 4 KB blocks
- Numbered from 0 to N-1

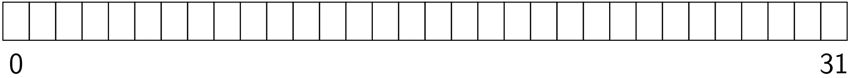


Figure: Abstract view of a disk = Array of blocks

Blocks

Blocks

- Disks are divided into blocks of fixed size
- Typically 4 KB blocks
- Numbered from 0 to N-1

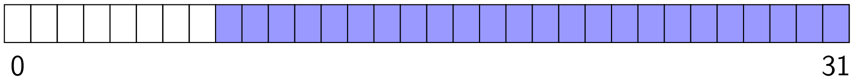


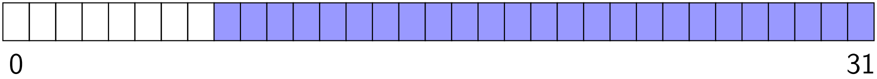
Figure: Abstract view of a disk = Array of blocks

- Most blocks are data blocks!
- They form the data region

Inodes

Inodes

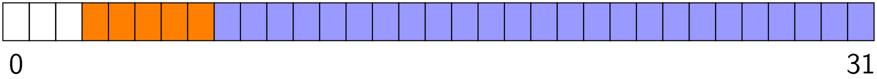
- Store the metadata for a file (which data blocks belong to the file, file size, owner, access rights, . . .)
- Inode stands for index node



Inodes

Inodes

- Store the metadata for a file (which data blocks belong to the file, file size, owner, access rights, ...)
- Inode stands for index node
- Inodes are stored in the inode table
- One block can contain multiple inodes

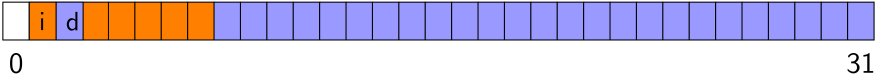


Tracking free space

We need a way to know if a data block or an inode is free.

Bitmap

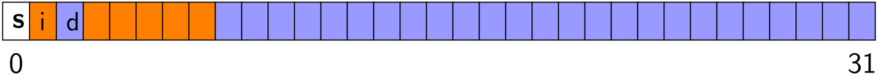
- Set of bits (one for each object)
- A bit set means the object is in-use.
- We use one inode bitmap and one data bitmap



The superblock

Superblock

- First block read when mounting a file system
- Contains information about the file system:
  - File system type
  - Number of data blocks and inodes
  - Beginning of the inode table
  - ...



Inodes: How to index the content of a file?

Indexing inodes

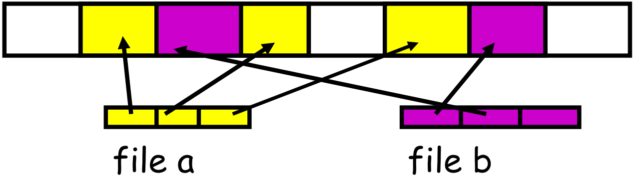
- An inode is identified by an inumber
- Corresponds to its index in the inode table
- Computing in which sector an inode is stored is easy (inputs: inode table start address, inumber, size of inode, size of block, size of sector)

Direct pointer

- An inode can include an array of direct pointers
  - Disk address of the data blocks belonging to the file

Example with direct pointers

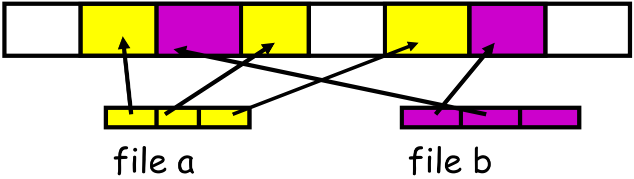
Figure by Prof D. Mazieres



Problem

Example with direct pointers

Figure by Prof D. Mazieres



Problem

- What if the file is big?

Inodes: How to index the content of a file?

Multi-level index

- Use indirect pointers
- Allocate an indirect block from the data-block region
  - Use this block to store direct pointers
  - With blocks of 4 KB and 4-bytes disk addresses, we can store 1024 addresses in one block.
- Instead of pointing to a block of data, we make the inode to point to an indirect block
- What if we want to support larger files?

Inodes: How to index the content of a file?

Multi-level index

- Use indirect pointers
- Allocate an indirect block from the data-block region
  - Use this block to store direct pointers
  - With blocks of 4 KB and 4-bytes disk addresses, we can store 1024 addresses in one block.
- Instead of pointing to a block of data, we make the inode to point to an indirect block
- What if we want to support larger files?
  - Use double indirect pointers

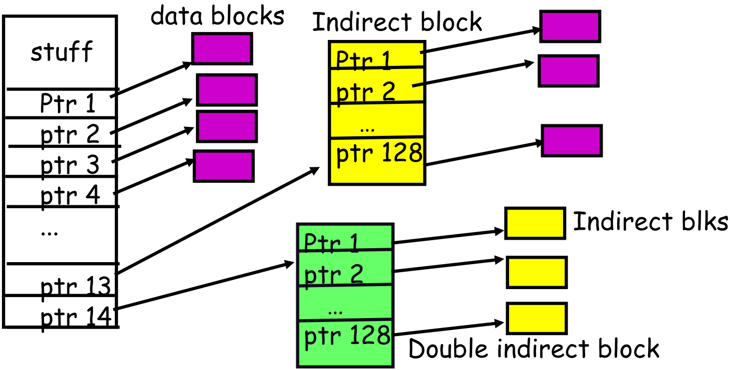
Multi-level index in practice

Several file systems (including Linux ext2 and ext3) use a multi-level index in the form of an unbalanced tree:

- The inode includes a few direct pointers (eg, 12 entries)
- If the file gets bigger, allocates an indirect block
  - Max file size becomes $(12 + 1024) \times 4$ KB.
- If the file gets bigger, allocate a double indirect block
  - Allocate a block that stores pointers to indirect blocks
  - Max file size becomes $(12 + 1024 + 1024^2) \times 4$ KB.
- What if the file gets bigger?
  - Use a triple indirect pointer.

Example of multi-level index

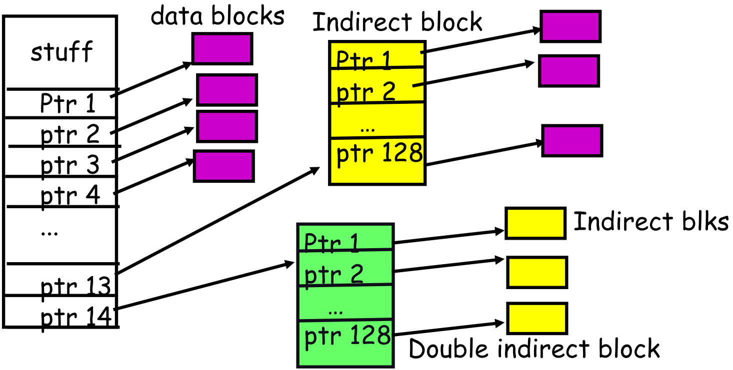
Figure by Prof D. Mazieres



Why such an imbalanced tree?

Example of multi-level index

Figure by Prof D. Mazieres



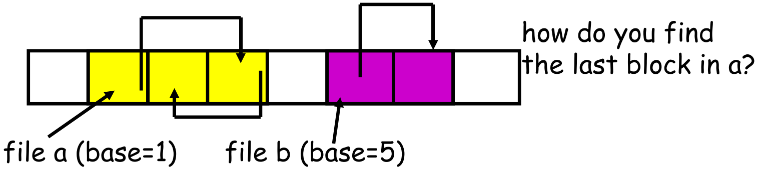
Why such an imbalanced tree?

- Recall that most files are small
- Optimized for this case: limit the number of indirections.

Alternatives to multi-level indexes

Linked-based approach

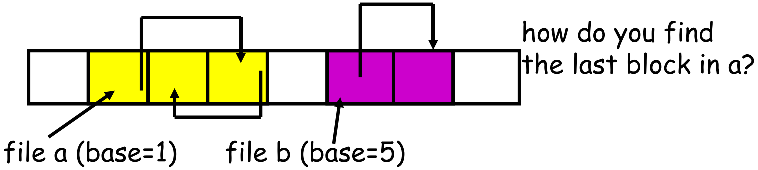
- An inode stores a single pointer to the first data block of the file
- Next block address is stored at the end of each data block



Alternatives to multi-level indexes

Linked-based approach

- An inode stores a single pointer to the first data block of the file
- Next block address is stored at the end of each data block
- Problem: Performance – large number of disk accesses to find the last block



Alternatives to multi-level indexes

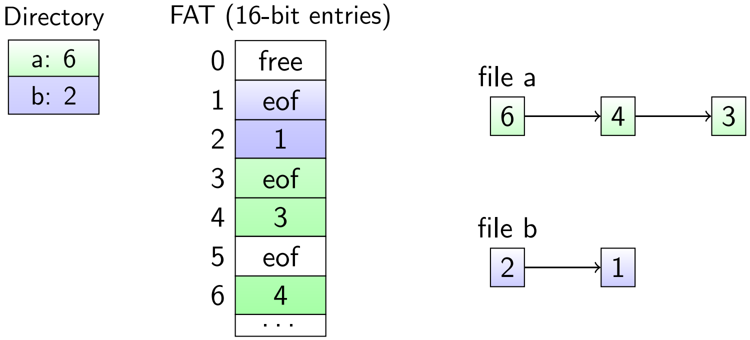
FAT

The old windows file system is linked-based:

- Improved with a FAT table (File Allocation Table)
  - Data structure stored in memory
  - The table contains an entry for each data block
  - An entry contains the index of the next data block
- FAT-16: $2^{16} = 65536$ entries, max FS size with 512-Byte blocks $= 32$ MiB

Example with FAT

Figure by Prof. D. Mazieres



- Drawback: pointer chasing
- Compared to pure linked-based approach, better because the FAT table can be loaded into memory

Alternatives to multi-level indexes

Use extents instead of pointers in index

- Goal: reduce the amount of metadata compared to pure index-based approaches
- Extent = disk pointer + length in blocks
- Avoids one entry per data block
- Multiple extents are used for flexibility
  - With a single extent per file, it might be hard to find a big enough contiguous free space on the disk to store a file.
- Example: Linux ext4
  - Backward compatibility with ext3: ext3 can be seen as ext4 with extents of size 1.

Directories

A directory

- A file of type directory (i.e., with metadata type= "directory")
- It has an inode that points to data-blocks
- Directory inodes and data blocks are stored in the inode table and data region of the file system
- Root dir has a pre-defined inumber ("2" in UNIX systems)

Data stored in a directory data block

- Information about the files and directories it contains
- For each entry:
  - The inumber
  - The name of the entry
  - (The size of the name)

Managing free space

Bitmap

- Tracks free inodes and free data blocks (2 separate bitmaps)
- Bitmaps are only accessed if a new allocation is needed

Allocation policy

- Looks for a set of contiguous data blocks when creating a new file
  - Ensures contiguous accesses (at least a few)
  - ext2 and ext3 do this (look for 8 contiguous blocks)

About performance

With our FS, what is the number of I/O when accessing a file?

- It depends on the length of the path (at least two reads per directory)
- For write/create operations, bitmaps and inodes need also be modified

Caching

- Most file systems use main memory as a cache to store frequently accessed blocks
- Cache for reads: can prevent most I/Os
- Cache for writes:
  - Impair reliability
  - Most FS cache writes between 5 and 30 seconds
  - Better I/O scheduling
  - Merge writes (eg, for the bitmaps)

Agenda

Introduction

File system implementation

The Fast File System

Dealing with failures

Log-structured file systems

Take a step back

Did we take into account the fact that we were dealing with a disk in the design of our file system?

Take a step back

Did we take into account the fact that we were dealing with a disk in the design of our file system? **No**

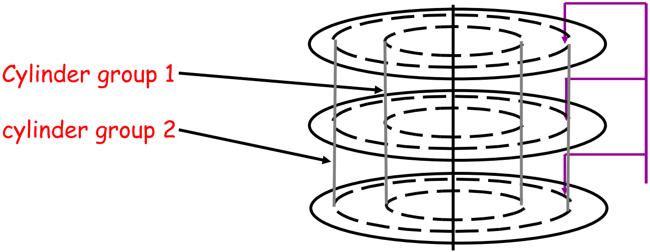
How bad is it?

- The presented design corresponds to the original UNIX file system by K. Thompson
- It has been shown that after some time, such a file system may deliver only 2% of overall disk bandwidth
- We lose all our time in seeks

The Fast File System (FFS)

Disk awareness

- Divide the disks in groups called cylinder groups
- Each cylinder group is a *mini* file system. It includes:
  - A copy of the superblock
  - Per-groups bitmaps
  - Per-groups inode and data blocks regions
- Allocate inode and data blocks for a file in the same group
  - They are guaranteed to be on close tracks/cylinders



The Fast File System (FFS)

Allocation policy

- Two ideas:
  - Keep related stuff together
  - Balance the load between groups

- For directories: Select a group with a low number of allocated directories and a high number of free inodes.
- For files: Place them in the same group as the directory they belong to.

The Fast File System (FFS)

Large files problem

- If a file fills the group it belongs to, the FFS allocation strategy is defeated
  - Other *related* files cannot be stored in the same group.

The Fast File System (FFS)

Large files problem

- If a file fills the group it belongs to, the FFS allocation strategy is defeated
  - Other *related* files cannot be stored in the same group.

Solution

- Only allocate the first data blocks in the same group as the directory
- Then place file chunks in different groups (chosen based on low utilization for instance)
- About chunk size:
  - It should be large enough for data transfer not to be dominated by seek time.
  - FFS uses the structure of inodes: each indirection block (and related data blocks) is placed in a different group.

Agenda

Problem with failures

Crash failures can occur at any moment (eg, power outage).

- Data saved on disk should still be available on restart after a crash.

Our file system may be impacted by such a crash!

- A crash may leave the file system in an inconsistent state

Inconsistent states

Update operations on the file system (create dir, create file, write file) require several I/O operations.

- What if a crash occurs before all operations related to an update are completed?
  - **The file system will be in an inconsistent state**

Illustration

- Append one data block to a file: requires 3 writes (data bitmap, the file inode, the data block)
  - Only data block is written: FS remains consistent, data is lost
  - Only inode is written: Inode points to trash, bitmap and inode are not consistent
  - Only bitmap is written: A data block is lost (space leak)

Solutions

Ideal solution

- Make all updates in one atomic step to avoid any inconsistencies
  - Impossible, the disk does one write at a time

2 existing techniques

- File system checker (fsck)
- Journaling

File system checker

Basic idea

- Let inconsistencies happen and try to fix them on restart
- Scan the file system (superblock, bitmaps, inodes) and check for inconsistencies

Comments

- Extremely inefficient!
- Checking the whole FS when maybe a single inode is inconsistent.

Journaling

Basic idea

- Write-ahead logging (database community)
- Write the update to be applied in a journal (also stored on disk) before actually running it
- If a failure occurs in the middle of the update, we can read the journal on restart and try again (or at least fix inconsistencies).

Comments

- Solution used by many FS including Linux ext3, Linux ext4 and Windows NTFS.
- Linux ext3 looks the same as ext2 except that a journal is added to the file system (one more region)

Journaling

Transactions

- Updates are saved in the journal as transactions (TxB: transaction begin, TxE: transaction end)

TxB id=1	I[v2]	B[v2]	D

Journaling

Transactions

- Updates are saved in the journal as transactions (TxB: transaction begin, TxE: transaction end)
- The TxE block is written only when the transaction becomes valid (all information regarding the update have been written to the journal)
  - Write of TxB and transaction data can be issued in parallel; Write of TxE is done only once first writes are finished

TxB id=1	I[v2]	B[v2]	D	TxE id=1	...

Journaling steps

Update operations:

- Journal write: Write the content of the transaction and wait for write to finish
- Journal commit: Write the transaction commit block (TxE) and wait for it to finish
- Checkpoint: Write the actual update to the disk

Recovery

- Replay all committed transactions (TxE has been written)
- Ignore uncommitted transactions

Note that to improve performance several updates can be aggregated in a single large transaction (Linux ext3)

More on journaling

Managing journaling storage space

- A circular buffer (the journal superblock stores the *begin* and *end* index)
- After a checkpoint, the indexes are updated correspondingly
- Prevents having to replay a lot of transactions on restart

Metadata journaling

- Journaling has a high cost: data are written twice
- How to avoid inconsistencies and avoid writing data twice?
  - Write data blocks directly in parallel with writing the transaction to the journal (before commit)
  - No inconsistency (in the worst case the data is lost)
  - Only metadata updates are committed in the journal
- Used by Linux ext3 (optional), and Windows NTFS

More on journaling: Block reuse

Quote from Stephen Tweedie (ext3 dev leader):

"What's the hideous part of the entire system? ... It's deleting files. (. . .) You have nightmares around what happens if blocks get deleted and then reallocated"

Problem

- Use of metadata journaling
- A directory is deleted, then a file is created and reuses the data blocks of the deleted directory.
  - Content of the file is not in the journal.
  - Content of data blocks for directories is considered as metadata (stored in the journal).
- A crash occurs and all operations related to the directory are still in the journal.
- How to prevent damaging the file by replaying operations related to the directory?

More on journaling: Block reuse

Solution

- Add *revoke* transactions to the journal
  - Deleting a directory adds a revoke transaction to the journal.
- Don't replay transactions related to revoked data blocks
  - On recovery, the journal is first scanned to look for revoked data blocks

Agenda

Motivation

Introduction comments

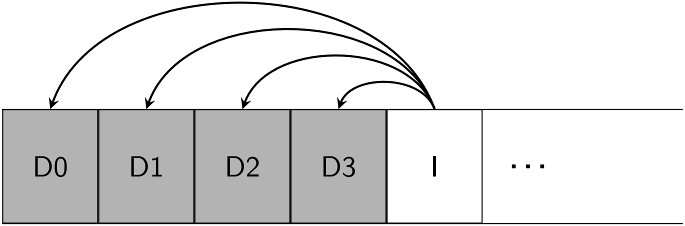
- With growing memory size, all I/O ops become update ops (reads hit the in-memory cache)
- Each update operation induces several I/O writes.
- Existing file systems induce small seeks and rotational delays for each update operation (write the bitmap, inode, data blocks).
  - True even when the disk is divided into cylinder groups

How to make all writes sequential?

Log-structured file systems

Basic idea

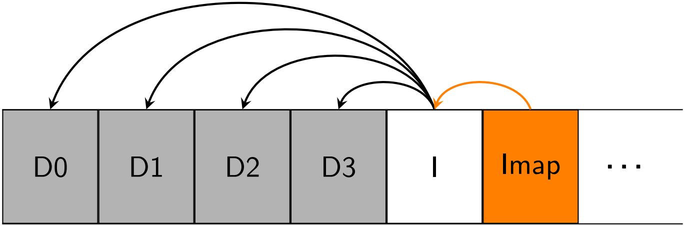
- Write all updates sequentially to the disk (data and metadata)
- Use write buffering to have large sequential writes to apply
- Copy-on-Write (CoW) strategy (Linux btrfs, Sun's ZFS).
  - Instead of overwriting existing content on update, always write to new portions of the disk.
  - Affordable as disk space becomes less expensive
- Examples: LFS (The Log-structures File System)



LFS

The Inode map (Imap)

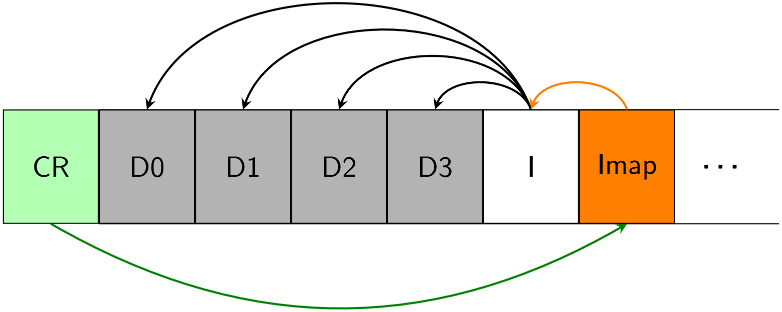
- How to find inodes?
- Solution: A new level of indirection
  - An inode map stores the address of the most recent version of each inode.
- Update of the inode map is part of the sequential updates
  - Only the modified chunks of the map are included in the update



LFS

The checkpoint region

- How to find the inode map chunks after restart?
- Solution: A checkpoint region that is updated periodically (every 30 seconds)



Garbage collection (GC)

We need to free space at some point. 2 problems have to be solved:

Determining if a block is still valid

- Store inode number (file it belongs to) and offset in file in each block
- Read the inode to determine if it still points to that block

Avoiding creating holes in the address space when cleaning

- The LFS cleaner creates new segments out of old still valid segments and write them again.

Limits of Log-structured File Systems

Performance

- Risks of fragmentation
  - Slowly growing files/ simultaneous growing files
  - Non-sequential modifications of files
- Performance slowdown when it nears maximum capacity
  - GC has to be run often

References for this lecture

- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau
    - Chapter 39: Files and Directories
    - Chapter 40: File System Implementation
    - Chapter 41: Fast File System
    - Chapter 42: FSCK and Journaling
    - Chapter 43: Log-Structured File System