DM510: File System Implementation

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

Today’s lecture

- Chapter 14 of course book
- Introduction to programming project 3: implement a file system

Overview

We assume the following basic abstraction of a hard drive is provided, e.g. by driver:

- Hard drive is array of **logical blocks** of specific size (e.g. 4 KB) that can be accessed by index

In this lecture, we are concerned with a (logical) file system responsible for

- Managing files, directories
- Managing protection and metadata
- Implementing file system operations, e.g. listing directory content, adding/removing files, etc.

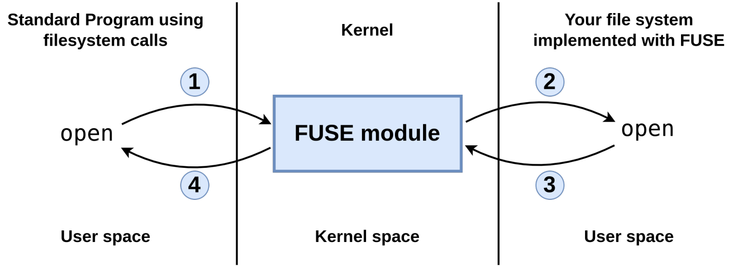
Project 3 and Concept For Very Naive File System

Overview

- For Project 3 you will implement your own file system using FUSE
- Simplifying assumptions: files may have fixed size, efficiency not important
- Choose **at least one** sophisticated file system features (e.g., from this lecture)

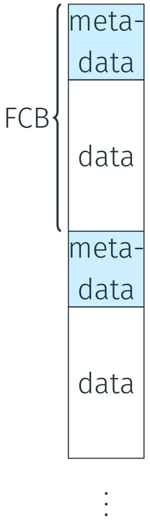
FUSE

- Framework to write a file system in user space that can be mounted into Linux
- You “only” need to implement the callback functions defined in FUSE


Standard Program using filesystem calls
Kernel
Your file system implemented with FUSE
1
2
open
FUSE module
open
4
3
User space
Kernel space
User space

Project 3 structure overview

- For each directory and each file there is a **file-control-block (FCB) / inode**, here containing both meta-data (name, directory, etc.) and data of file
- Persistent data can be an array of FCBs,
- For persistence, dump data into file (of regular hard drive) when unmounting and load when mounting



File system operations

- **Reading/writing:** Write into data part of correct FCB
- **Listing directory contents:** requires scanning through entire array of FCBs
- **Creating/remove files and directories:** Add/remove FCB (may require rearranging FCBs and resizing array)

Towards Realistic File Systems

Components

In storage

May not all be separate in specific implementation:

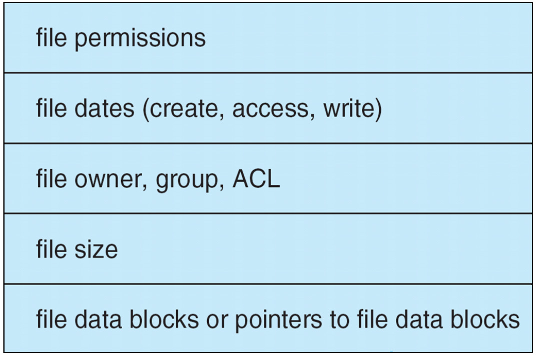
- File-control blocks / inodes
- Directory structure
- Data blocks
- Auxiliary data structures (free block lists, etc.)

In main memory

- Open file table
- Cache

File control block (FCB)

- Stores all file meta-data and data itself or where to find data
- In Unix usually called **inode**


file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Directory structure

- Manages filenames, paths, (links to) FCP
- **Many options:** linear list, sorted list, tree structure, hash table
- **Aspects to consider:** space efficiency, performance, reliability, flexibility in directory size

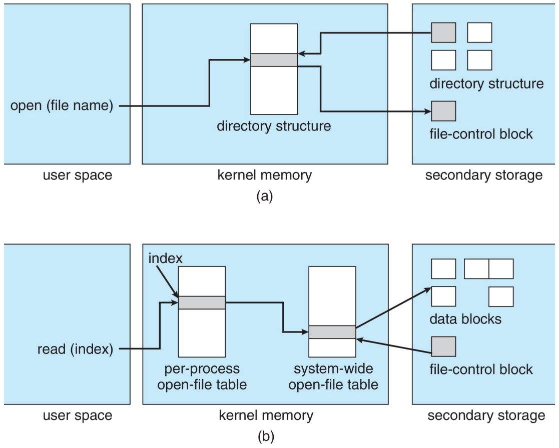
In-memory file system structures

For optimization: caches for frequently accessed data:

- Directory structures
- Free data block list
- Frequently accessed file contents

Non-persistent data associated with file systems:

- Open-file tables, containing e.g. current position in sequential file access, list processes that have file open



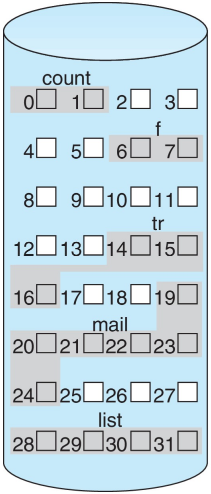
Allocation Methods

Contiguous allocation

- Each FCB contains pointer to first block and number of consecutively allocated blocks
- Very efficient access (both random and sequential)
- Leads to **external fragmentation**

Extent based system

- extent: contiguous blocks
- zero or more extents in file
- More flexible
- How to list extents for each file? *Combine with other approaches*

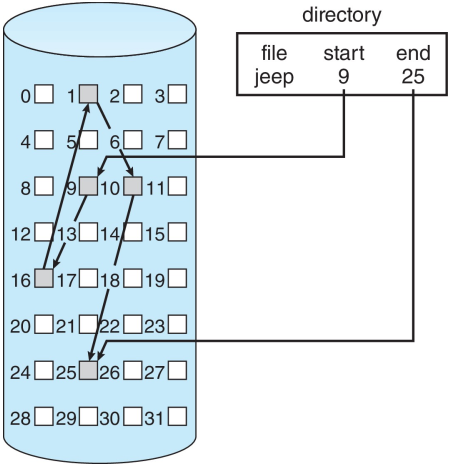


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

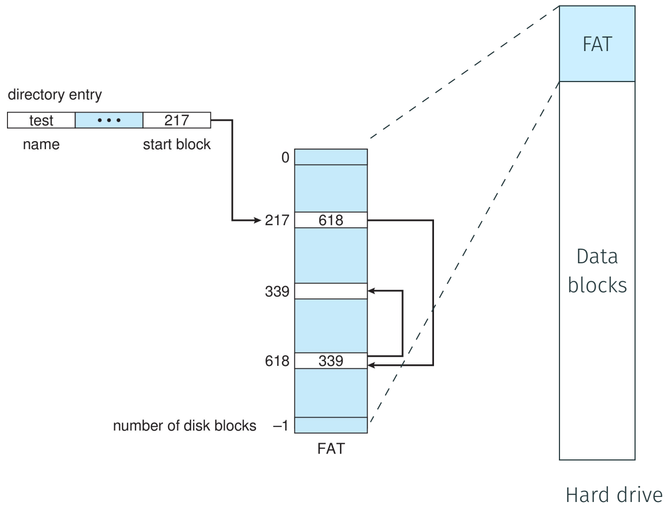
Linked allocation

- Blocks for one file form a linked list: FCB contains pointer to first and last block; each block contains pointer to next block of this file
- Storage overhead: Loses size of one pointer for each block
- Slow random access: need to traverse entire linked list, which is scattered over hard drive (bad locality)
- Weaknesses can be mitigated by **FAT**



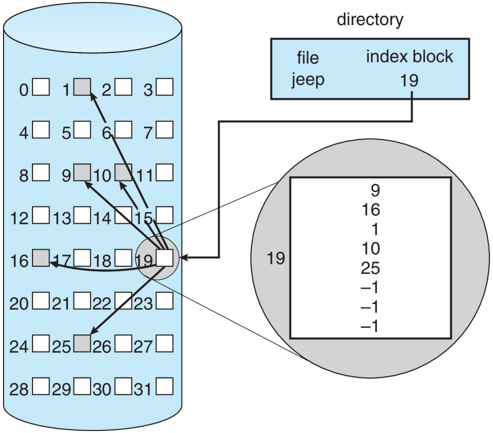
File-allocation table (for linked allocation)

- Store linking information in file-allocation table (FAT) in specific segment
- FAT has entry for each block, with index of next block or special value for end of list
- Better locality for random access + FAT can be cached



Indexed allocation

- Each file contains pointer to special **index block**
- Index block contains pointers to all other blocks of file
- Overhead for index block
- Number of blocks for a file limited by number of pointers that fit into block

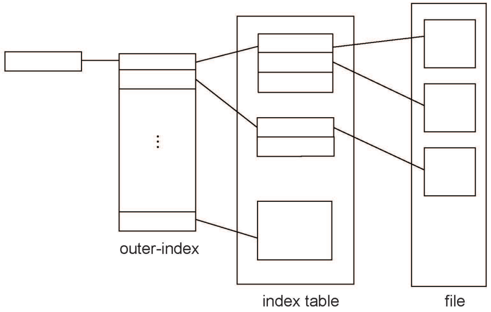


Indexed allocation

- Each file contains pointer to special **index block**
- Index block contains pointers to all other blocks of file
- Overhead for index block
- Number of blocks for a file limited by number of pointers that fit into block

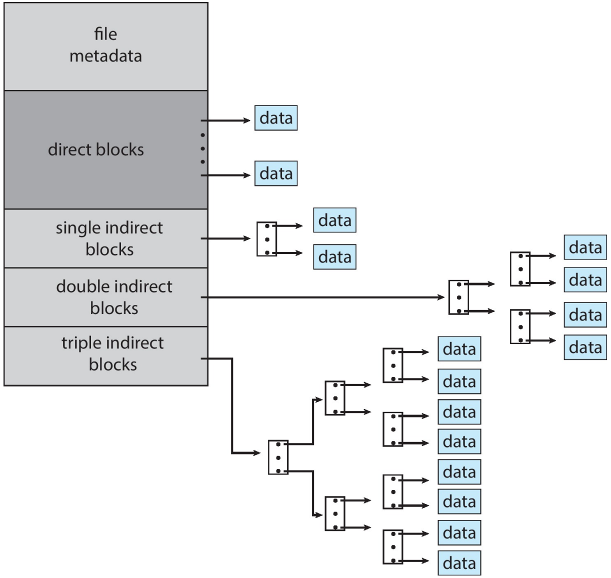
Large files

- Can use linked list of index blocks
- or multilevel indexing



Combined scheme

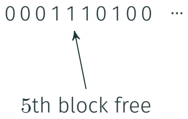
- Example: Unix UFS
- 4KB blocks, 32 bit addresses



Data structures for free blocks

Bit map

- For hard drive with n blocks, use vector $\mathrm{free} \in \{0,1\}^n$ of n bits.
- $\mathrm{free}[i] = 0$: ith block is occupied
- $\mathrm{free}[i] = 1$: ith block is free
- Space overhead: 1/bits-per-block



Data structures for free blocks

Bit map

- For hard drive with n blocks, use vector $\mathrm{free} \in \{0,1\}^n$ of n bits.
- $\mathrm{free}[i] = 0$: ith block is occupied
- $\mathrm{free}[i] = 1$: ith block is free
- Space overhead: $1/$bits-per-block

Linked list

- No space overhead
- Efficient for allocating a single block (or few)
- Contiguous allocation not easy

Data structures for free blocks

Bit map

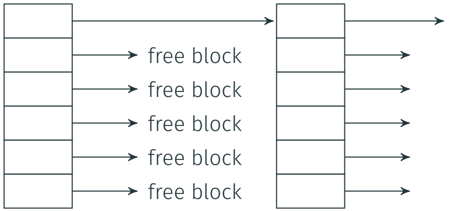
- For hard drive with n blocks, use vector $\text{free} \in \{0,1\}^n$ of n bits.
- $\text{free}[i] = 0$: ith block is occupied
- $\text{free}[i] = 1$: ith block is free
- Space overhead: 1/bits-per-block

Linked list

- No space overhead
- Efficient for allocating a single block (or few)
- Contiguous allocation not easy

Grouping

- Linked list of (not all) free blocks
- Inside each of the blocks in list there are pointers to other free blocks (not in the list)



Data structures for free blocks

Bit map

- For hard drive with n blocks, use vector $\mathrm{free} \in \{0,1\}^n$ of n bits.
- $\mathrm{free}[i] = 0$: ith block is occupied
- $\mathrm{free}[i] = 1$: ith block is free
- Space overhead: 1/bits-per-block

Linked list

- No space overhead
- Efficient for allocating a single block (or few)
- Contiguous allocation not easy

Grouping

- Linked list of (not all) free blocks
- Inside each of the blocks in list there are pointers to other free blocks (not in the list)

Counting

For more efficient use of contiguous areas of free blocks:

- Pointer to first free block and count of following free blocks
- Keep areas (first block and count) in linked list

Recovery

Recovery

- Modifications to file system often involves many write operations
- **Problem:** system crash may leave file system in inconsistent state, potentially resulting in data loss
- Here our focus is on file system, not file content

Consistency checking

- Programs, e.g. `fsck`, can attempt to detect and fix inconsistencies
- Very slow and not guaranteed to succeed

Backups

- Copy entire hard drive content to other storage device (disk, magnetic tape, etc.)
- If file system is broken, restore from most recent backup
- Expensive and may still lose recent data

Other solutions for recovery

Log structured file system

- File system maintains a log of transactions
- A transaction described the modification to perform
- All modifications are written to the log
- Transactions are asynchronously applied to the actual data
- After system crash, (incomplete) transactions from log are first applied

Not-in-place updates

- Instead of overwriting blocks with new data (in-place), create copy of block and modify it
- Once finished, change link from old copy to new copy