CS323 Operating Systems

Filesystem implementation

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Topics covered in this lecture

- Filesystem implementation
- Special requirements (write back)

This slide deck covers chapters 40, 41 in OSTEP.

Last week: API and abstractions

- Highlevel API and abstractions
- Filesystem API
- Different names for different use cases
  - Inodes and devices
  - Path
  - File descriptor

Different names for different use cases

- inode/device id (53135/2)
  - Unique internal name
  - Records metadata about the file (size, permissions, owner)
- path (`/foo/bar/baz`)
  - Human readable name
  - Organizes files in a hierarchical layout
- file descriptor (5)
  - Process internal view
  - Avoids frequent path to inode traversal
  - Remembers offset for next read/write

Common file API

- `int open(char *path, int flag, mode_t mode)`
- `size_t read(int fd, char *buf, size_t nbyte)`
- `size_t write(int fd, char *buf, size_t nbyte)`
- `int close(int fd)`

What kind of on disk data structures do we need? How is data accessed?

This week: Abstractions -> Implementation

Virtual File System

- File systems (EXT4, NTFS, FAT) use different data structures
- A Virtual File System (VFS) abstracts from the real filesystem
- VFS abstracts the FS as objects with specific operations
  - Superblock (mount): a life filesystem
  - File (open): a file opened by a process ("open file description")
  - Directory entry cache: speeds up path to inode translation
  - Inode (lookup): a filesystem object (e.g., file or directory)
- System call logic (open, seek, ...) maps to VFS operations
- When implementing a new FS, implement the VFS API
- System calls are now independent of FS implementation

Challenge: renaming files

- How would you implement `rename`?

Challenge: renaming files

- How would you implement `rename`?
- Renaming only changes the name of the file
- Directory contains the name of the file
- No data needs to be moved, inode remains unchanged

Challenge: renaming files

- How would you implement `rename`?
- Renaming only changes the name of the file
- Directory contains the name of the file
- No data needs to be moved, inode remains unchanged
- Note, you may need to move the data if it is on another disk/partition!

Filesystem implementation

- A filesystem is an exercise in data management
- Given: a large set (N) of blocks
- Need: data structures to encode (i) file hierarchy and (ii) per file metadata
  - Overhead (metadata size versus file data) should be low
  - Internal fragmentation should be low
  - File contents must be accessed efficiently (external fragmentation, number of metadata accesses)
  - Define operations for file API

Filesystem implementation

- A filesystem is an exercise in data management
- Given: a large set (N) of blocks
- Need: data structures to encode (i) file hierarchy and (ii) per file metadata
  - Overhead (metadata size versus file data) should be low
  - Internal fragmentation should be low
  - File contents must be accessed efficiently (external fragmentation, number of metadata accesses)
  - Define operations for file API
- Many different choices are possible!
  - Similar to virtual memory!
  - Software implementation enables experimentation with strategies

Allocating file data

- Contiguous
- Linked blocks (blocks end with a next pointer)
- File-allocation tables (table that contains block references)
- Indexed (inode contains data pointers)
- Multi-level indexed (tree of pointers)

For each approach, think about fragmentation, ability to grow/shrink files, sequential access performance, random access performance, overhead of meta data.

File allocation: contiguous

Each file is allocated contiguously

File allocation: contiguous

Each file is allocated contiguously

- Terrible external fragmentation (OS must anticipate)
- Likely unable to grow file
- Excellent read and seek performance
- Small overhead for metadata

File allocation: contiguous

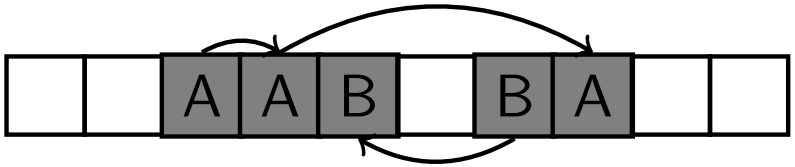
Each file is allocated contiguously

- Terrible external fragmentation (OS must anticipate)
- Likely unable to grow file
- Excellent read and seek performance
- Small overhead for metadata

- Great for read-only file systems (CD/DVD/BlueRay)

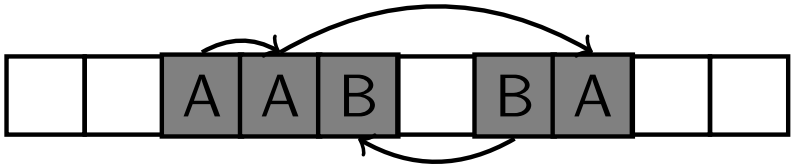
File allocation: linked blocks

Each file consists of a linked list of blocks



File allocation: linked blocks

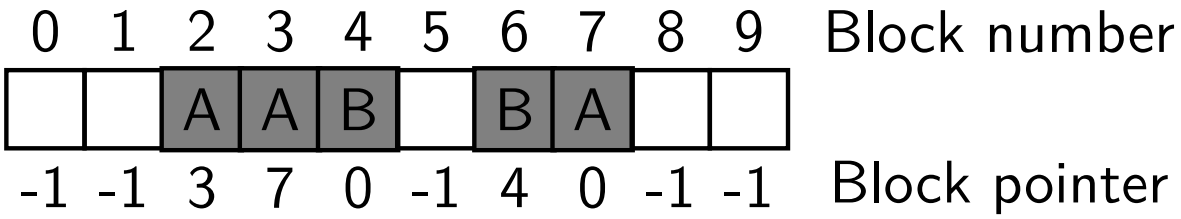
Each file consists of a linked list of blocks



- No external fragmentation
- Files can grow easily
- Reasonable read cost (depending on layout), high seek cost
- One pointer per block metadata overhead

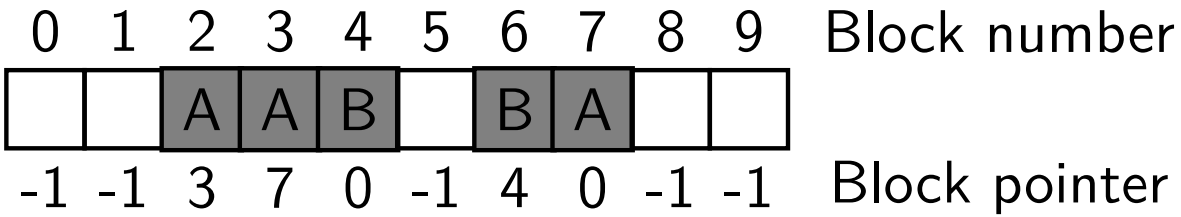
File allocation: File Allocation Table (FAT)

Idea: keep linked list information in a single table. Instead of storing the next pointer at the end of the block, store all next pointers in a central table



File allocation: File Allocation Table (FAT)

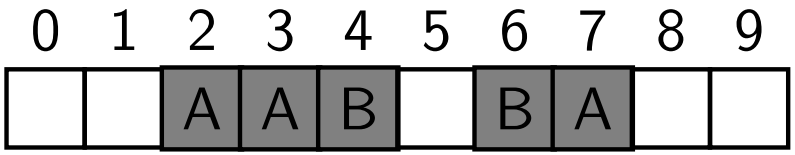
Idea: keep linked list information in a single table. Instead of storing the next pointer at the end of the block, store all next pointers in a central table



- No external fragmentation
- Files can grow easily
- Reasonable read and seek cost
- One pointer per block metadata overhead

File allocation: indexed

Idea: metadata contains an array of block pointers

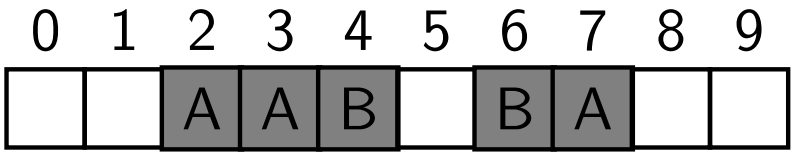


File A: 2, 3, 7, -1

File B: 6, 4, -1, -1

File allocation: indexed

Idea: metadata contains an array of block pointers



Block number

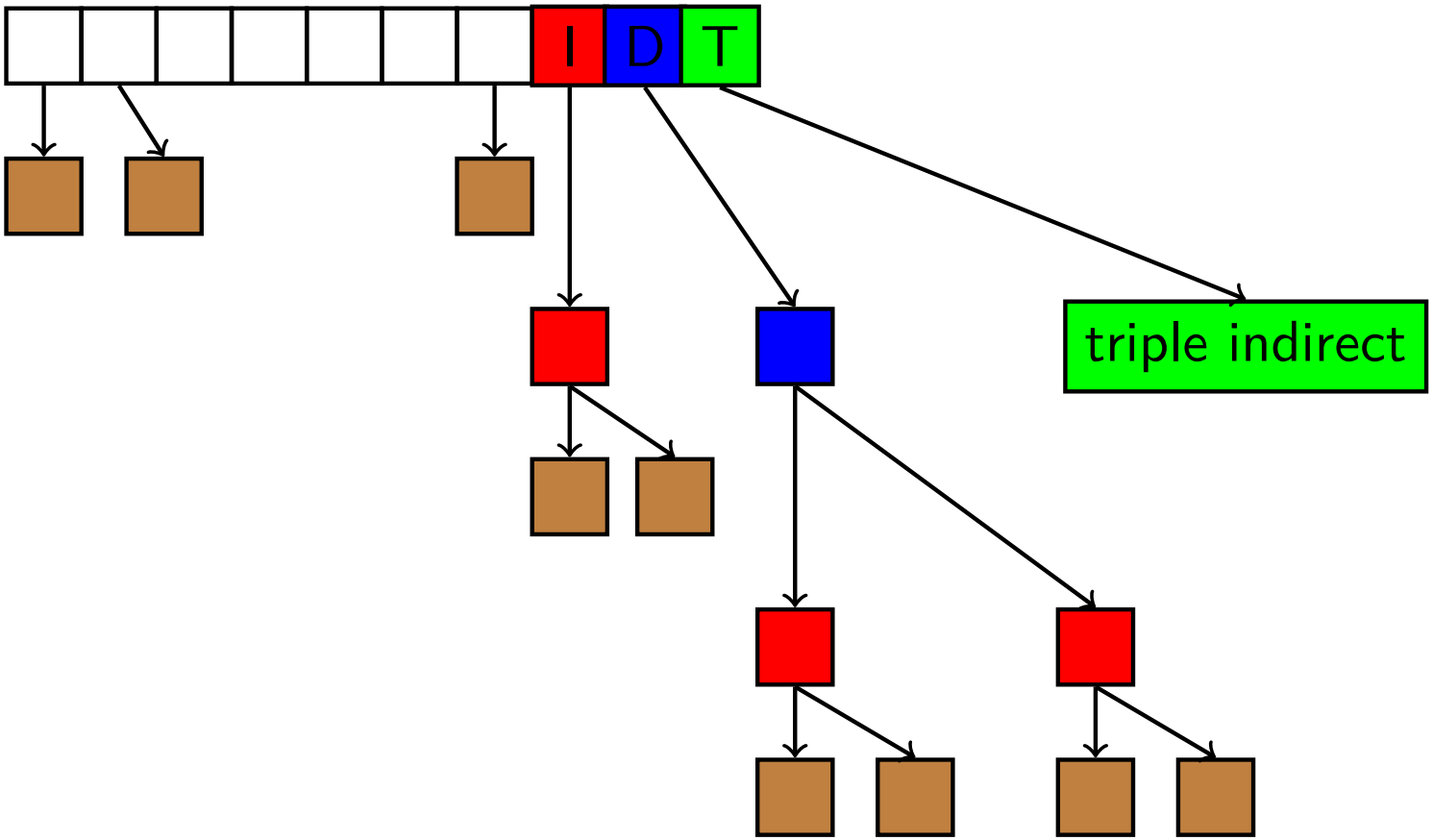
File A: 2, 3, 7, -1

File B: 6, 4, -1, -1

- No external fragmentation
- Files can grow easily up to maximum size
- Reasonable read and low seek cost
- *Large metadata overhead* (wastes space as most files are small)

File allocation: multi-level indexing (1/3)

Idea: have a mix of direct, indirect, double indirect, and triple indirect pointers



File allocation: multi-level indexing (2/3)

Idea: have a mix of direct, indirect, double indirect, and triple indirect pointers

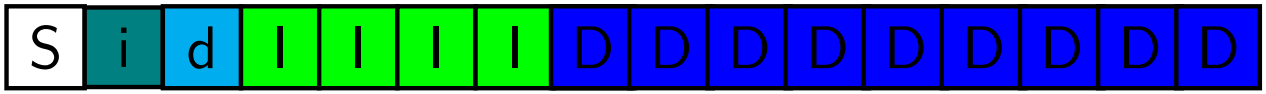
```
struct inode {
  umode_t          i_mode;
  unsigned short   i_opflags;
  kuid_t           i_uid;
  kgid_t           i_gid;
  unsigned int     i_flags;
  ...
  // direct pointers to data blocks
  struct dblock  *direct[10];
  // block of N ptrs to data blocks
  struct dblock  **dindirect;
  // block of N ptrs to each N ptrs to data blocks
  struct dblock  ***tindirect;
};
```

File allocation: multi-level indexing (3/3)

Idea: have a mix of direct, indirect, double indirect, and triple indirect pointers

- No external fragmentation
- Files can grow easily up to maximum size
- Reasonable read and low seek cost
- Low metadata overhead but needs extra reads for indirect/double indirect access

Simple FS



- Superblock (S): file system metadata
- Bitmaps (i, d): indicates free blocks
- Inodes (I): hold file/directory metadata, reference data blocks
- Data blocks (D): file contents, referenced by an inode

The inode size may be different (smaller) from the data block size.

Simple FS: superblock

- The superblock stores the characteristics of the filesystem
- What do you store in the superblock?

Simple FS: superblock

- The superblock stores the characteristics of the filesystem
- What do you store in the superblock?

- Magic number and revision level
- Mount count and maximum mount count
- Block size of the filesystem (1, 2, 4, 8, 16, 32, 64K for ext4)
- Name of the filesystem
- Number of inodes/data blocks
- Number of free inodes/data blocks
- Number of "first" inode (i.e., root directory)

Simple FS: inode

- The inode stores all file metadata
- What would you store in an inode?

Simple FS: inode

- The inode stores all file metadata
- What would you store in an inode?

- File type
- File uid, gid
- File permissions (for user, group, others)
- Size
- Access time
- Create time
- Number of links

Maximum file size

- Maximum file size is related to
  - Block size
  - Number of direct inodes
  - Number of indirect inodes
  - Number of double indirect inodes
  - Number of triple indirect inodes
- `blocksize * (direct + inodeblock + inodeblock^2 + inodeblock^3)`

Directories

- Directories are special files (`inode->type`)
- Store a set of file name to inode mappings
- Special entries `.` for current directory and `..` for parent directory

File operation: create /foo/bar

- Read root inode (locate directory data)
- Read root data (read directory)
- Read foo inode (locate directory data)
- Read foo data (read directory)
- Read/write inode bitmap (allocate inode)
- Write foo data (add file name)
- Read/write bar inode (create file)
- Write foo inode (update date, maybe allocate data block)

File operation: open /foo/bar

- Read root inode (locate directory data)
- Read root data (read directory)
- Read foo inode (locate directory data)
- Read foo data (read directory)
- Read bar inode (read file metadata)

File operation: write to /foo/bar

- First: `open("/foo/bar")`
- Read bar inode (read file metadata)
- Read/write data bitmap (allocate data blocks)
- Write bar data (write data)
- Write bar inode (update inode)

File operation: read from /foo/bar

- First: `open("/foo/bar")`
- Read bar inode (read file metadata)
- Read bar data (read data)
- Write bar inode (update time)

File operation: close /foo/bar

- No disk I/O

File operation: observations

- Path traversal and translation is costly
  - Reduce number of lookups (file descriptors!)
  - Introduce caching (dcache)
- Lookup aside, operations are cheap and local

Summary

- Filesystem implementation
- Inodes for metadata
- Bitmaps for inodes/data blocks
- Superblock for global metadata

Don't forget to get your learning feedback through the Moodle quiz!