File Systems

1DV512 – Operating Systems

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 25, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapters 10 & 11

Agenda

- Motivation and Introduction
- File System Interface
- File System Implementation
- Summary

Motivation

- For most users, the *file system* is the most visible aspect of a general-purpose operating system ⇒ FS describes how files are mapped onto physical devices, as well as how they are accessed and manipulated by both users and programs
- Accessing physical storage can often be slow, so FSs must be designed for efficient access ⇒ but also protection and — in some cases — file sharing and remote file access
- FS consists of two distinct parts:
  - a collection of *files* ⇒ each stores the respective data
  - a *directory* structure ⇒ organizes and provides information about all the files in the system

Agenda

- Motivation and Introduction
- File System Interface
- File System Implementation
- Summary

File Operations

- A *file* is a logical storage unit ⇒ a named collection of related information that is recorded on secondary storage
- Typical attributes of a file: *name* (human-readable), *identifier* (internal for the OS), *type*, *location*, *size*, *protection* (permissions), *timestamps*, *extended attributes*. . .
- Most OSs themselves do not distinguish between most file types based on their contents/structure (i.e., treat them simply as sequences of bytes) ⇒ one exception is *executable files*
- The basic file operations supported by OSs:
  - *create* ⇒ allocate space + add new *directory entry*
  - *open* ⇒ provide a *file handle* / *file descriptor* to be used for other operations
  - *write* ⇒ involves a *write pointer* to the position within the file
  - *read* ⇒ involves a *read pointer*; can coincide with the write pointer ⇒ then *current-file-position pointer*
  - *seek* ⇒ reposition within
  - *delete* ⇒ remove the directory entry and release disk space; can be unavailable due to *hard links* ⇒ multiple directory entries for the same file
  - *truncate* ⇒ erase contents, but keep attributes

File Access

Main file access strategies:

- *Sequential* access
  - Information in the file is processed in order, one *logical record* (or a byte) after the other
  - Operations: *read next*, *write next*, *reset/rewind* ⇒ no reading after reaching the end!
- *Direct/random* access
  - Based on the disk/block model ⇒ fixed-length logical records accessed rapidly in arbitrary order
- Further strategies can be built based on the direct access, e.g., *index* file(s)
  - Keep index in memory for fast determination of location of data to be operated on
  - If too large, index (in memory) of the index (on disk)
  - Often used for databases

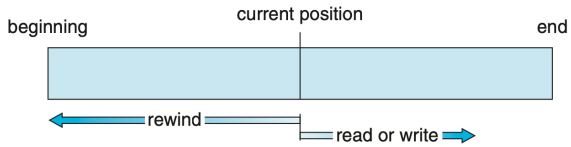

Fig. 13.4 in OSC book

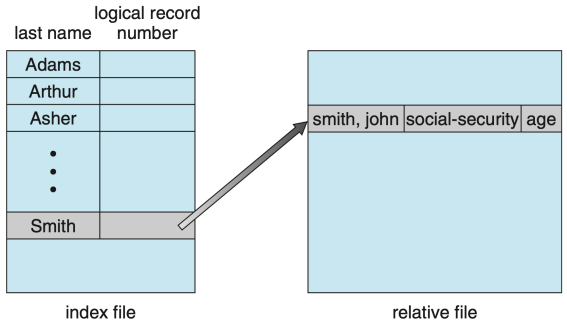

Fig. 13.6 in OSC book

File Directory

- *Directory* ⇒ a collection of nodes containing information about all files within FS
- Can be viewed as a symbol table that translates file names into their *file control blocks*
- The directory itself can be organized in many ways
- The basic directory operations supported by OSs:
  - *create* a file
  - *delete* a file ⇒ the FS might need to *defragment* the directory structure afterwards!
  - *search* for a file
  - *rename* a file ⇒ might involve relocating it, too
  - *list* a directory
  - *traverse* the file system

Directory Structures

- Single-level directory
  - A single directory for all users
  - Limitations incl. naming and grouping
- Two-level directory
  - Separate directory for each user ⇒ file *paths* must be introduced
  - Still grouping issues for each user's files
- Tree-structured directory
  - Further extension ⇒ the users can create *subdirectories* to group the files
  - Directories can be treated as files themselves ⇒ a special bit + list of files
  - Each process has a *current directory*
  - Paths can be specified in *absolute* or *relative* way, e.g., `../file.txt`
  - To delete a directory ⇒ all the contents must be deleted first recursively

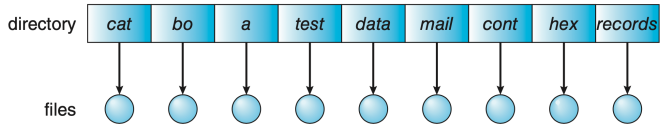

Fig. 13.7 in OSC book

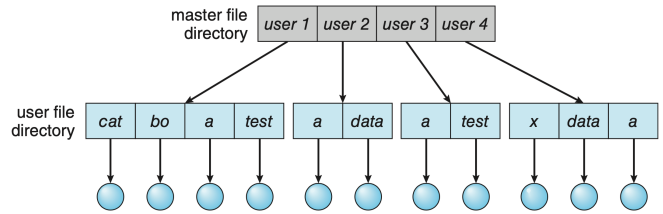

Fig. 13.8 in OSC book

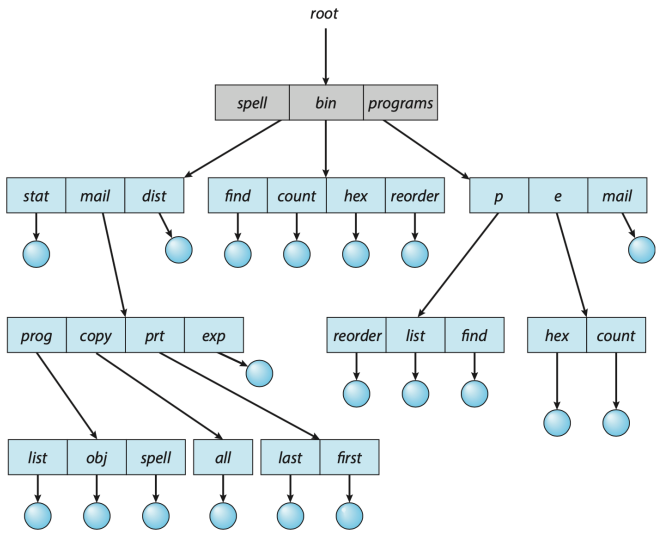

Fig. 13.9 in OSC book

Directory Structures (cont.)

- *Acyclic graph* directory
  - *Sharing* directories and files ⇒ not the same as copying!
  - *Link* ⇒ another name (pointer) to an existing file
  - To *resolve* the link ⇒ follow the pointer to locate the file
  - Deleting can lead to *dangling* pointers. . .
  - *Soft/symbolic link* ⇒ using a path; deleting the link does not affect the file, but the link can be dangling
  - *Hard/nonsymbolic link* ⇒ keeps a reference count in the file control block
  - Compared to a tree, traversal can become problematic!
- *General graph* directory ⇒ further efforts for traversal and deletion

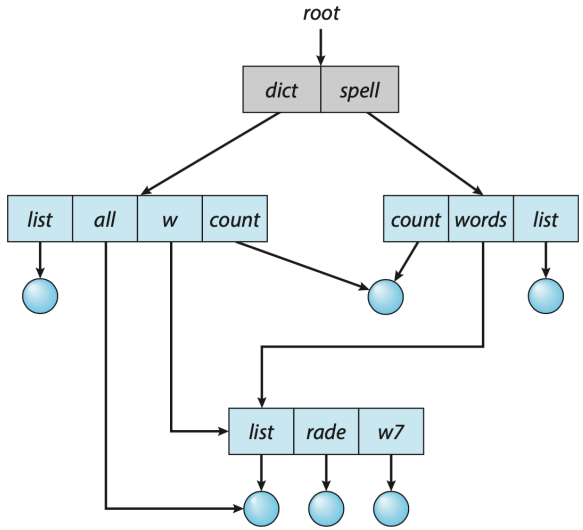

Fig. 13.10 in OSC book

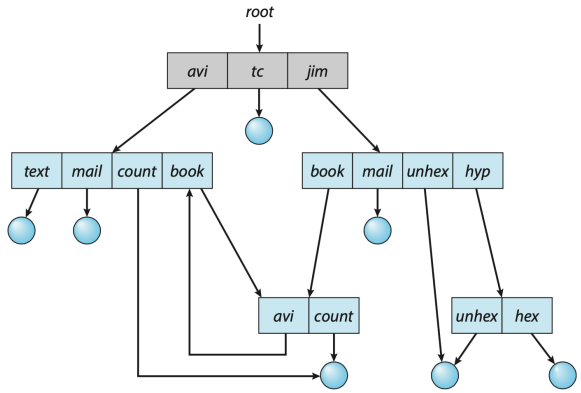

Fig. 13.11 in OSC book

File Protection

- We want to keep information safe from physical damage ($\Rightarrow$ *reliability*) and improper access ($\Rightarrow$ *protection*)
- File owner/creator should be able to control *what* can be done and by *whom*/*what*
- Common types of permitted file operations:
  - *read*
  - *write* $\Rightarrow$ for directories in Unix-like systems, also requires the execution permission!
  - *execute* $\Rightarrow$ also required for navigating into directories in Unix-like systems
  - *append*
  - *delete*
  - *list*
  - *modify attributes*
- *Permissions* for the file's *owner*, *group* members, and *other* users
- *Access control lists (ACL)* $\Rightarrow$ more fine-tailored policies for specific users, etc.

```
-rw-rw-r--   1 pbg   staff     31200  Sep 3 08:30   intro.ps
drwx------   5 pbg   staff       512  Jul 8 09.33   private/
drwxrwxr-x   2 pbg   staff       512  Jul 8 09:35   doc/
drwxrwx---   2 jwg   student     512  Aug 3 14:13   student-proj/
-rw-r--r--   1 pbg   staff      9423  Feb 24 2017   program.c
-rwxr-xr-x   1 pbg   staff     20471  Feb 24 2017   program
drwx--x--x   4 tag   faculty     512  Jul 31 10:31  lib/
drwx------   3 pbg   staff      1024  Aug 29 06:52  mail/
drwxrwxrwx   3 pbg   staff       512  Jul 8 09:35   test/
```

Sec. 13.4.2 in OSC book

Memory Mapped Files

- *Memory mapping* ⇒ mapping a disk block to a page (or pages) in memory
- Treat file I/O as memory access operations ⇒ simplifies and speeds up access
- Updates to the file contents not necessarily *synchronized (flushed)* to disk immediately ⇒ care must be taken!
- Memory-mapping system calls can also support copy-on-write functionality
- Shared memory is actually often implemented by memory mapped files

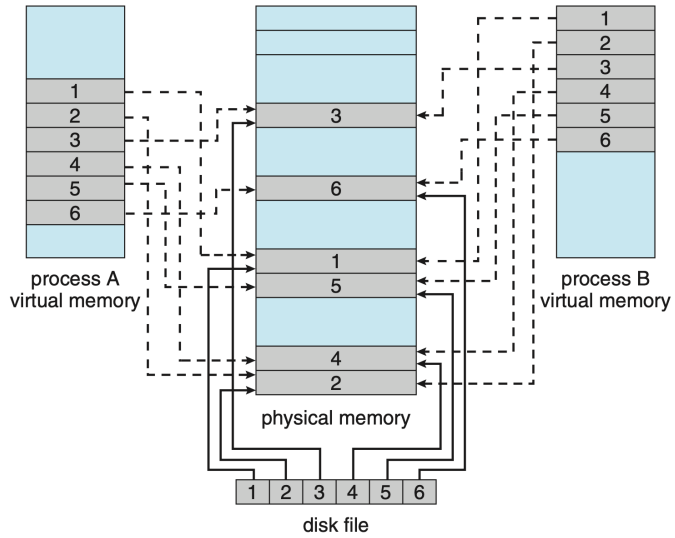

Fig. 13.13 in OSC book

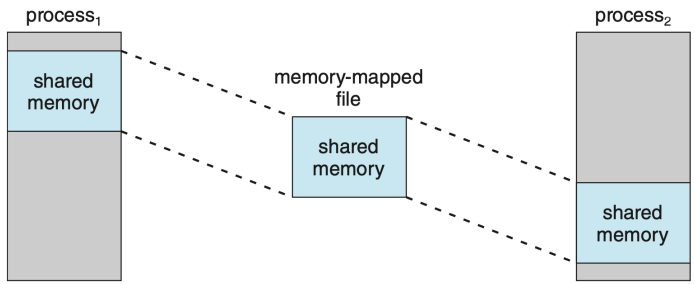

Fig. 13.14 in OSC book

Agenda

- Motivation and Introduction
- File System Interface
- File System Implementation
- Summary

Storage Organization and Mounting

- Disk can be subdivided into *partitions* (*slices*)
- Disk or partition can be used *raw* without a file system (e.g., for swap or special purposes), or formatted with a file system
- Entity containing file system known as a *volume*
- Each volume containing file system also tracks that file system's info in *device directory* or *volume table of contents*
- A file system must be *mounted* before it can be accessed ⇒ specific *mount point* is used

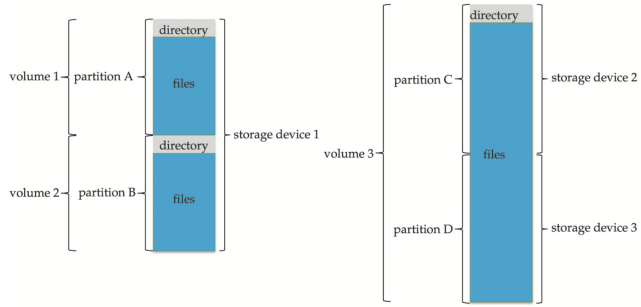

Fig. 15.1 in OSC book

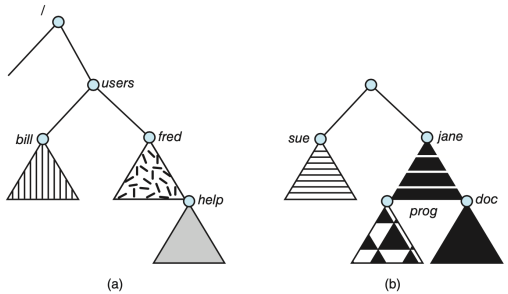

Fig. 15.3 in OSC book

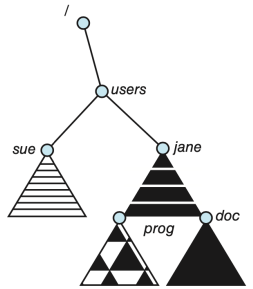

Fig. 15.4 in OSC book

File System Organization

File systems typically composed of many different levels:

- Bottom layer ⇒ direct/random access to storage devices, typically in *blocks* or *sectors* of 512 bytes or 4096 bytes
- *I/O control* ⇒ hardware-specific device drivers and interrupt handlers; deals with commands such as *"read drive1, cylinder 72, track 2, sector 10, into memory location 1060"*
- *Basic file system* ⇒ "block I/O subsystem" in Linux, deals with abstract hardware-independent commands (e.g., *"retrieve physical block 123"*); also concerned with I/O request scheduling, memory *buffers* (⇒ hold data in transit), and *caches* (⇒ hold frequently used data)
- *File organization module* ⇒ understands files, logical addresses, and physical blocks; translates between logical and physical blocks, also manages free space and disk allocation
- *Logical file system* ⇒ manages metadata information using *file control blocks* (⇒ *inodes* in Unix), incl. directory management and protection

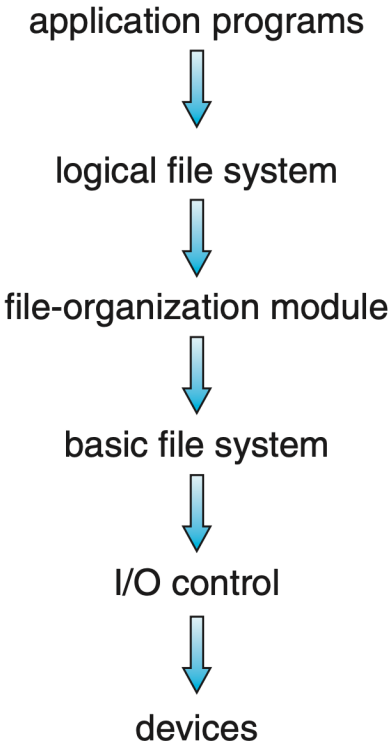

Fig. 14.1 in OSC book

On-Disk File System Structures

- We have system calls at the API level, but how do we implement their functions? ⇒ *On-disk* and *in-memory* structures
- *Boot control block* (per volume) contains info needed by system to boot OS from that volume ⇒ needed if volume contains OS, usually first block of volume
- *Volume control block* (*superblock*, *master file table*) contains volume details ⇒ total # of blocks, # of free blocks, block size, free block pointers or array
- *Directory structure* (per FS) organizes the files ⇒ names and inode numbers; implemented using a *linear list* or a *hash table*, typically
- *File Control Block (FCB)* (*inode*) (per file) contains many details about the file ⇒ inode number, permissions, size, dates; in some cases, e.g., NTFS, can use relational DB structures

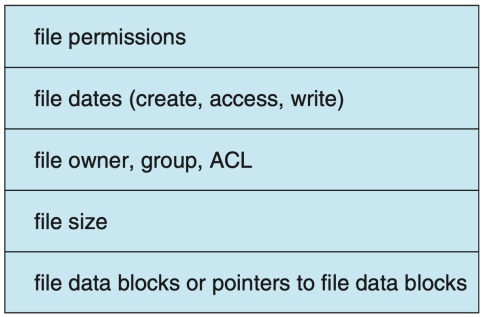

Fig. 14.2 in OSC book

In-Memory File System Structures

- *Mount table* ⇒ stores file system mounts, mount points, file system types
- *In-memory directory-structure cache* ⇒ the directory information of recently accessed directories
- *System-wide open-file table* ⇒ a copy of the FCB of each file and other info
- *Per-process open-file table* ⇒ pointers to appropriate entries in system-wide open-file table as well as other info
- *Buffers* ⇒ hold file-system blocks for read and write operations
- *File descriptors* (*file handles*) ⇒ pointers to entries in the per-process open-file tables, used to access file contents

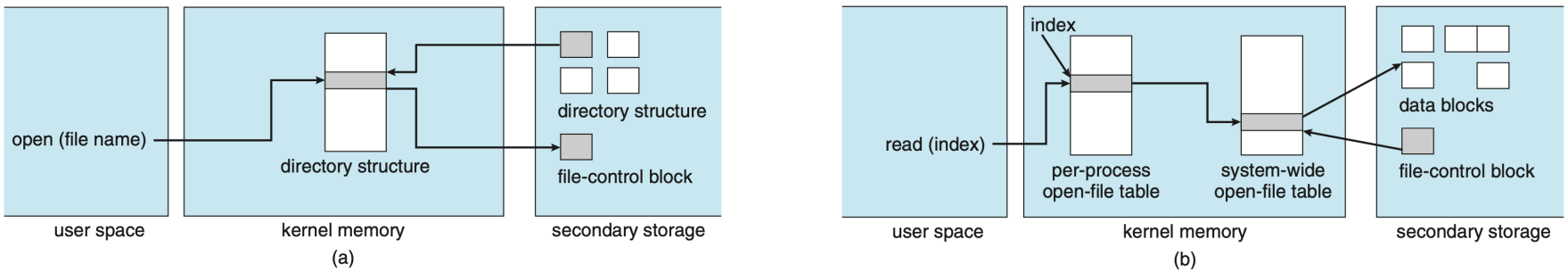

Fig. 14.3 in OSC book

Allocation Methods

Allocation method refers to how disk blocks are allocated for files:

- *Contiguous allocation* ⇒ each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple ⇒ only starting location (block#) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for compaction

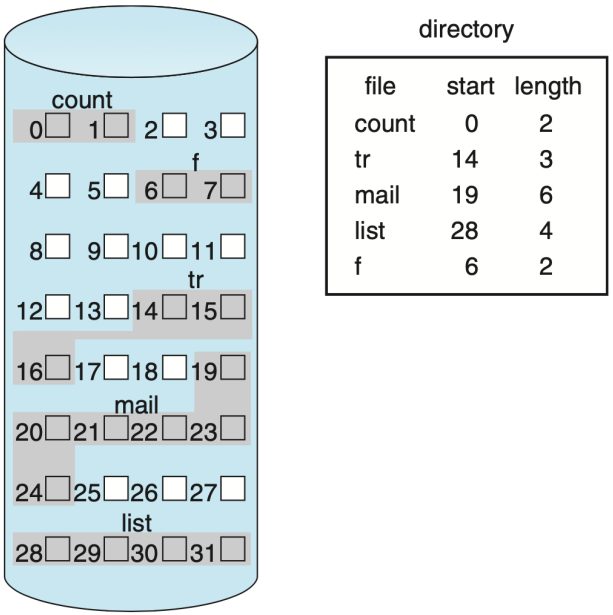

Fig. 14.4 in OSC book

Allocation Methods (cont.)

- *Linked allocation* ⇒ a file is a list of blocks
  - Blocks may be scattered anywhere on the device
  - The directory contains a pointer to the first and last blocks of the file
  - Avoids fragmentation problems and allows dynamic file growth
  - Problems include efficiency for random/direct file access, extra space for pointers, and reliability (bugs, hardware or power failures. . .)
- Improvements and additions:
  - *Clustering* blocks into groups ⇒ improves efficiency, but increases internal fragmentation
  - *File Allocation Table (FAT)* ⇒ a section of storage at the beginning of each volume is set aside to contain the table indexed by block number

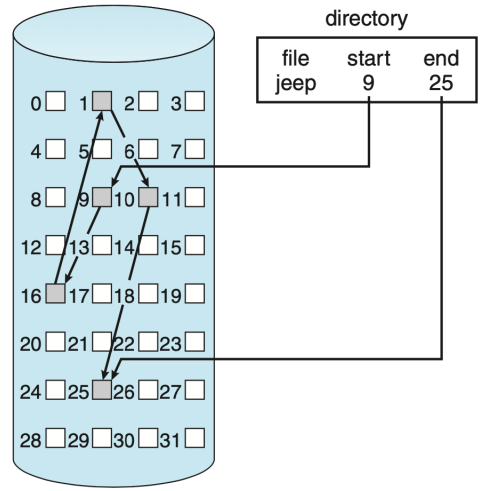

Fig. 14.5 in OSC book

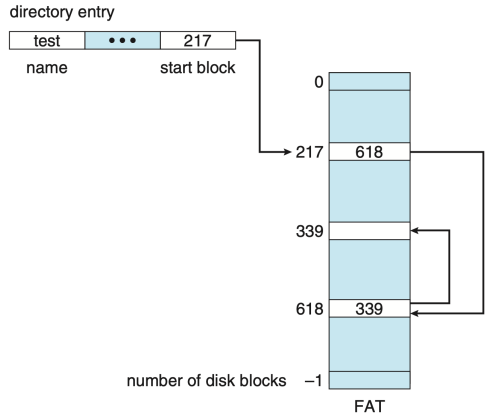

Fig. 14.6 in OSC book

Allocation Methods (cont.)

- *Indexed allocation* ⇒ each file has its own *index block(s)* of pointers to its data blocks
  - Support for random access dynamic access without external fragmentation ⇒ but overhead of the index block
  - Several strategies for how to store the index blocks ⇒ linked list of several blocks, multi-level index, combination of direct and indirect blocks. . .
  - Some performance problems due to scattered index blocks
- Example ⇒ Unix inode using a combined scheme with 15 pointers and 4KB per block
  - 12 pointers ⇒ direct blocks
  - 3 pointers ⇒ single, double, and triple indirect blocks, respectively
  - Similar approaches allow a huge number of blocks to be addressed

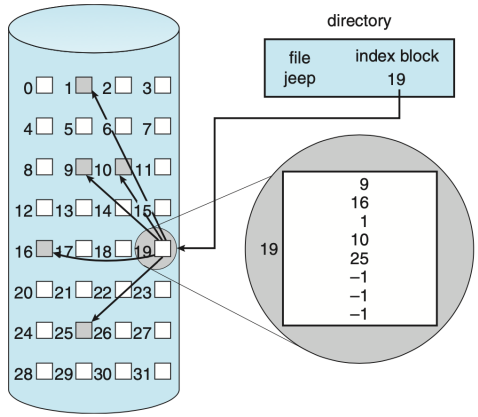

Fig. 14.7 in OSC book

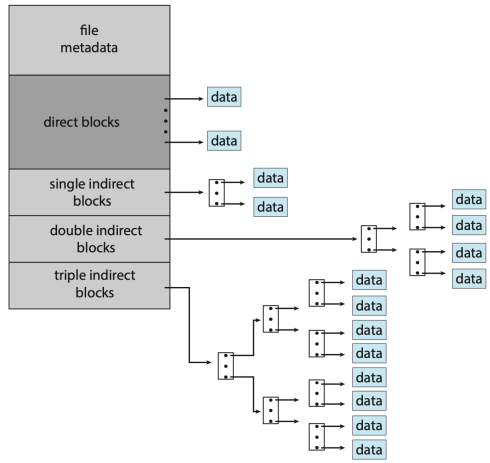

Fig. 14.8 in OSC book

Free Space Management

- File system maintains a *free-space list* to track available blocks/clusters
- Implementation options:
  - A *bit vector* with one bit per block
  - A linked list
  - *Grouping* $\Rightarrow$ modified free list with several block addresses grouped
  - *Counting* $\Rightarrow$ free list containing block address + number of subsequent contiguous blocks
  - *Space maps* in ZFS $\Rightarrow$ combining free list, counting, and activity logging techniques

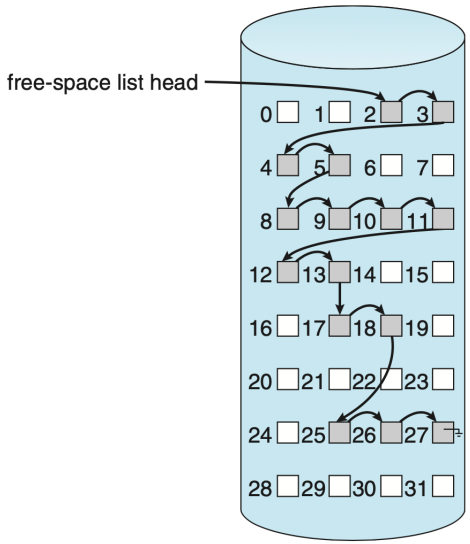

Fig. 14.9 in OSC book

File System Qualities

- *Efficiency* of disk space usage ⇒ depends on disk allocation and directory algorithms, etc.
- *Performance* ⇒ depends on support for buffers, cache, etc.
- *Page cache* ⇒ caches pages rather than disk blocks using virtual memory techniques and addresses
- *Unified buffer cache* ⇒ uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching; used in Linux and Windows, among others
- *Recovery* ⇒ supported via *consistency checking*, *journaling*, *backups*. . .

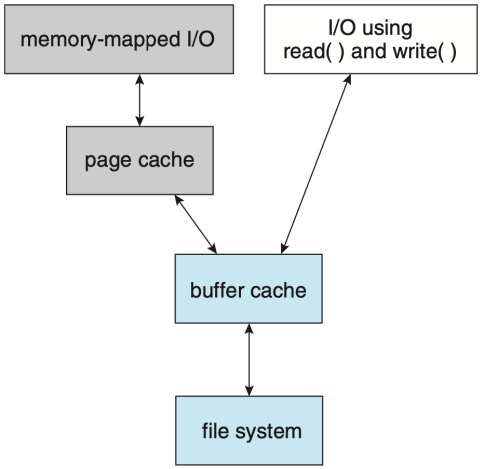

Fig. 14.10 in OSC book

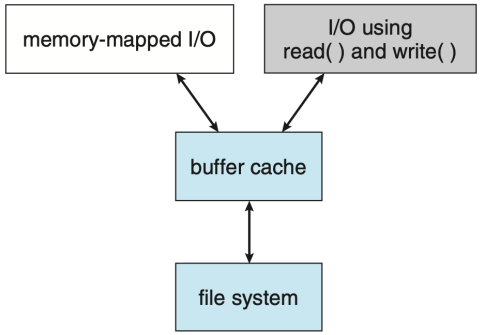

Fig. 14.11 in OSC book

Agenda

- Motivation and Introduction
- File System Interface
- File System Implementation
- Summary

Summary

- A file is an abstract data type defined and implemented by the OS
- A major task for the operating system is to map the logical file concept onto physical storage devices
- Within a file system, it is useful to create directories to allow files to be organized $\Rightarrow$ eventually, with hierarchical, tree-structured directory design, or even general graph structure
- File systems are often implemented in a layered or modular structure
- Files can typically be allocated space on the storage device in one of three ways $\Rightarrow$ contiguous, linked, or indexed allocation
- Depending on the operating system, the file-system space is seamless (mounted file systems integrated into the directory structure) or distinct (each mounted file system having its own designation)
- Most systems are multi-user and thus must provide a method for file sharing and file protection
- Numerous file systems exist, from FAT and ISO 9660 (CD-ROM), to $>$130 supported file systems in Linux