
File System Management

Last modified: 07.05.2021

Contents

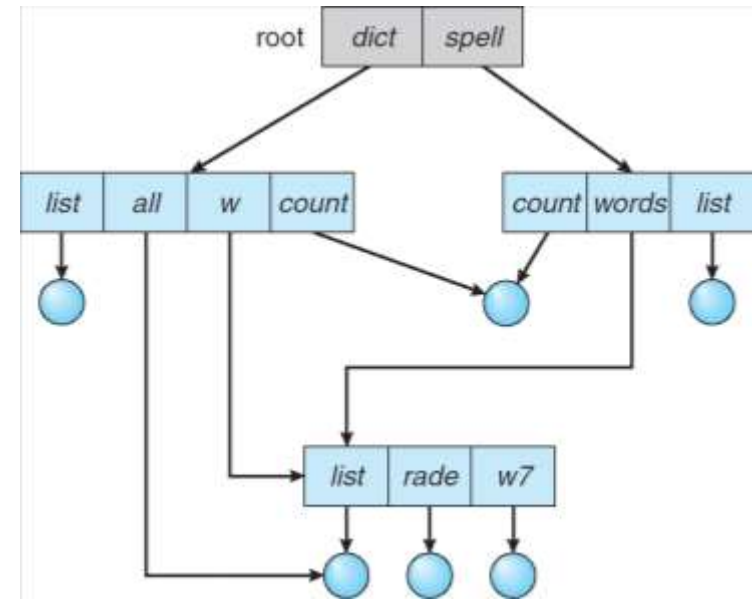
- Files and file systems
- File-System Structure
- Virtual File System and FUSE
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance. Buffering
- Recovery

Files and file systems

- **File**
 - A named collection of related information that is usually stored on secondary storage. A logical storage unit.
 - POSIX: „ **File** - an object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, symbolic link, socket, and directory. Other types of files may be supported by the implementation”
- Files are typically organized in a **file system**, which tracks file locations on the storage media and enables file operations, e.g.: create/remove, open, read, write, change default position in a file.
- File system creates a mapping of a file name into storage structure, containing physical file representation, by maintaining **file control blocks**:
 - UNIX: i-node number, permissions, size, dates
 - MSWin: NFTS stores info in master file table using relational DB structures
- There exist file systems (e.g. proc), which do not provide mapping to any storage media; files/folders are computed on request of a particular API function call, or the call is a means to access some internal kernel resource/property/information. In some file systems special files (UNIX: FIFO, /dev/*) give file-oriented access to extra functionality provided by OS kernel.

File system (FS)

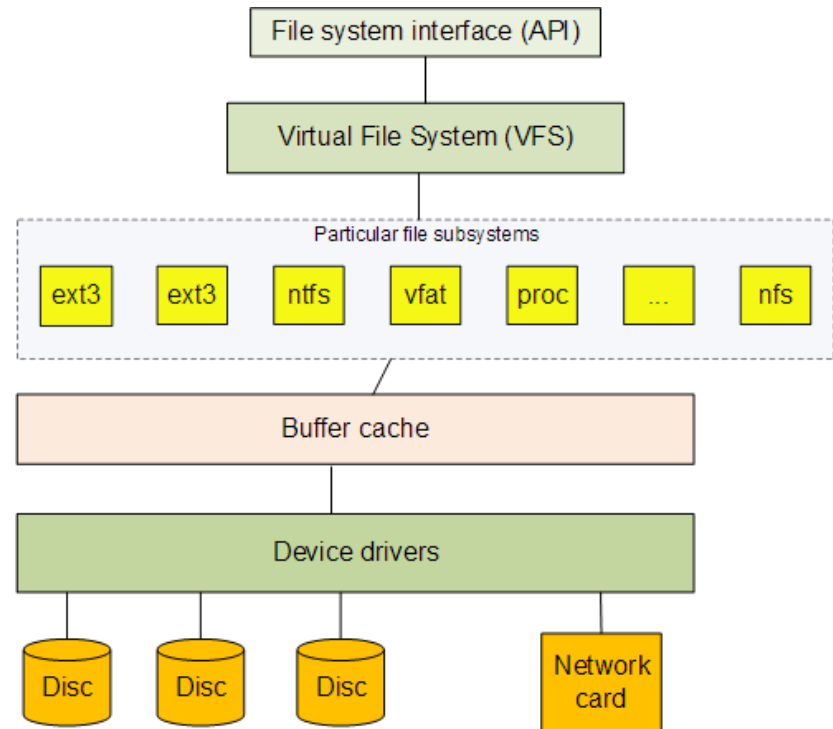
- File system maps a file name to the file content (some f. systems admit forks) and its attributes (mandatory, extended). File systems provide for grouping files in folders/catalogues. The folders are extra objects in FS (e.g., as special files)
- Logical relationships between files and folders are represented (API) as (acyclic) directed graph with one root node. Reference to a file/folder is just possible via name paths (w.r.t. FS root node). FS implementations of tree structure can introduce auxiliary objects, e.g. links.
- Enforcing assumed security rules and file system integrity OS tasks are typically supported by extra FS objects (ACL, journal).
- Prior to use, each FS has to be mounted - to be logically attached to a file system of a running operating system. Thus, each file of each mounted FS can be referred to with a global name path.



File system and operating system

OS is to provide

- API for file system operations
- Means to perform file system operations efficiently, enforcing assumed security rules and providing file system (FS) integrity.



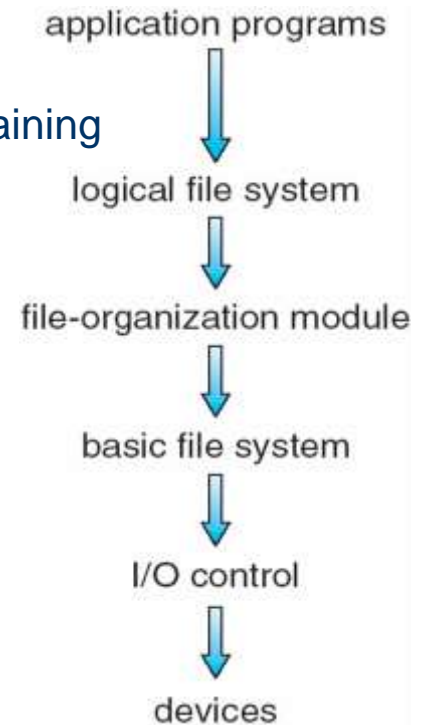
OS is to provide also

- means to perform
 - checking file system integrity
 - backup
 - reconstruction of a file system

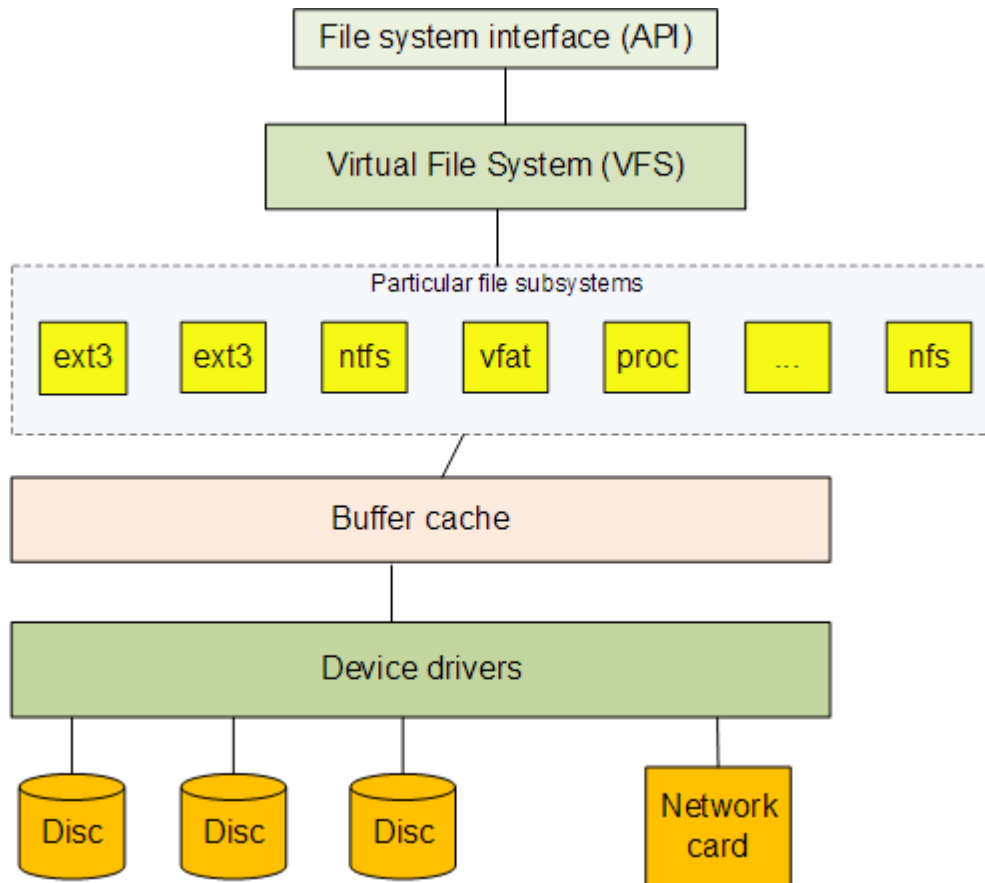
Layered structure of the file system management subsystem

File System Layers

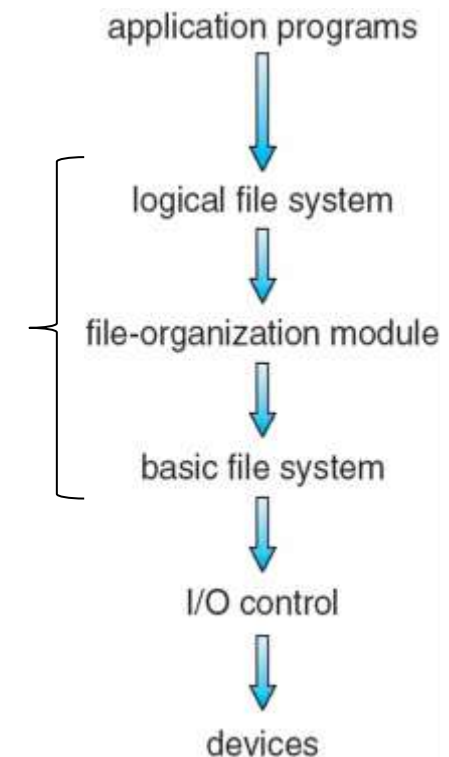
- **Logical file system** manages metadata information
 - Directory management
 - Translates file name into file number, file handle, location by maintaining **File Control Blocks (i-nodes)** in UNIX)
 - Protection
- **File organization module**
 - Translates logical block # to physical block #
 - Manages free space, disk allocation
- **Basic file system** given command like “retrieve block 123” translates to device driver. Also manages memory buffers and caches
 - Buffers hold data in transit
 - Caches hold frequently used data
- **Device drivers** manage I/O devices at the I/O control layer
 - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to disk controller
- **Disk** provides in-place rewrite and random access
 - I/O transfers performed in **blocks** of **sectors** (traditionally 512 bytes, nowadays 4kB)
 - Disk can be split into **partitions/volumes** with independent file systems.
 - **Volume control block (superblock, master file table)** contains volume details (total # of blocks, # of free blocks, block size, free block pointers or array, root dir. ptr...)



File system mgmt. subsystem



Hierarchical organization of a file system mgmt. subsystem

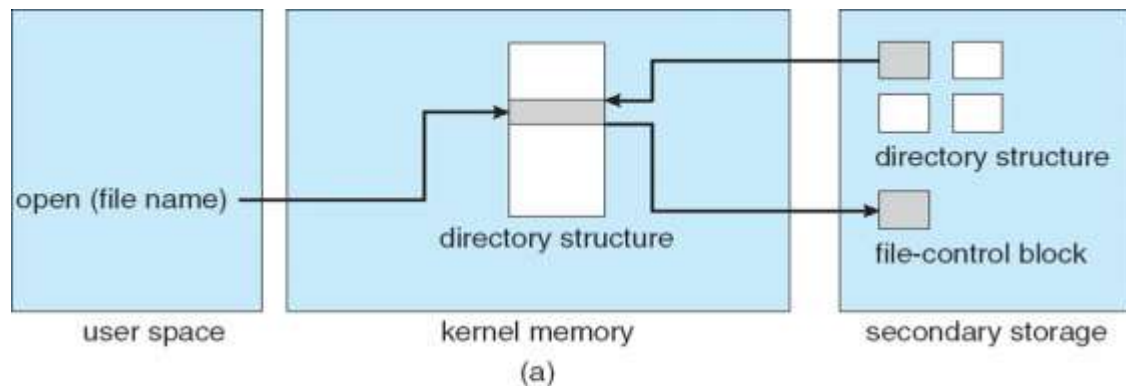


Layers of file system services

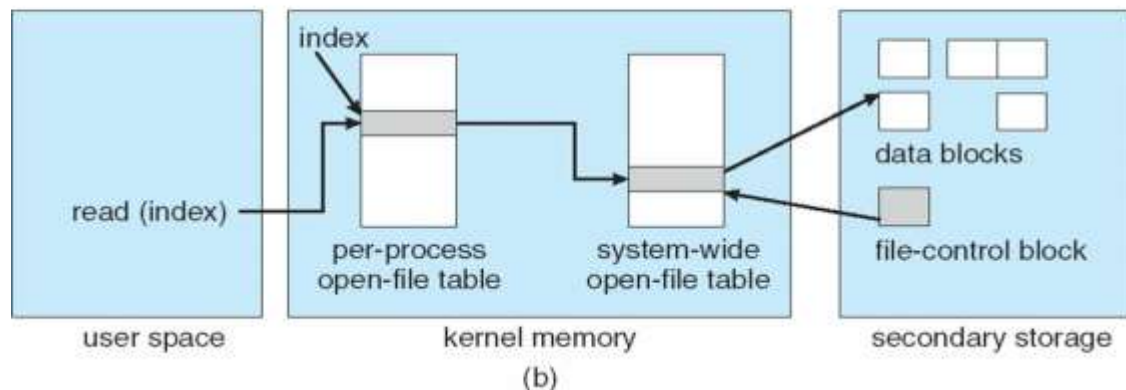
File System Interface Memory Structures

- Mount table storing file system mounts, mount points, file system types
- The following figures illustrate the necessary file system structures provided by the operating systems

- opening a file (a) →
- open returns a file handle for subsequent use

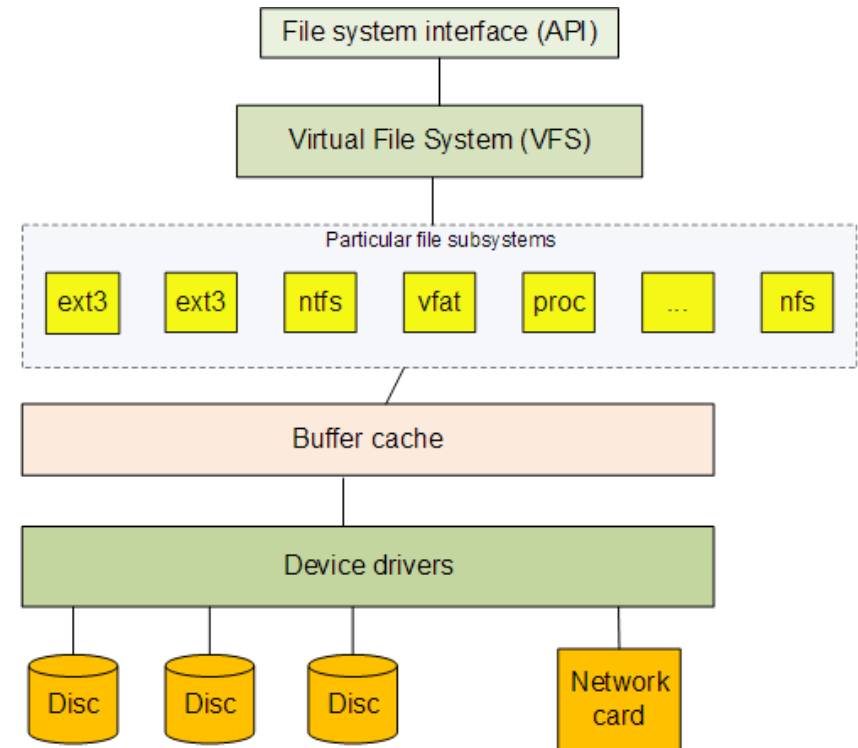


- reading a file (b) →
- buffers hold data blocks from secondary storage
- data from read eventually copied to specified user process memory address



Virtual File Systems

- Operating system can support
 - a fixed, small number of file systems
 - undefined number of file systems, sharing API
- **Virtual File Systems (VFS)** provides an object-oriented way of implementing file systems
 - Separates file-system generic operations from implementation details
 - Implements **vnodes** which hold local or network file details. The kernel maintains one **vnode** for each active node (file or directory)
 - Then dispatches operation to appropriate file system implementation routines



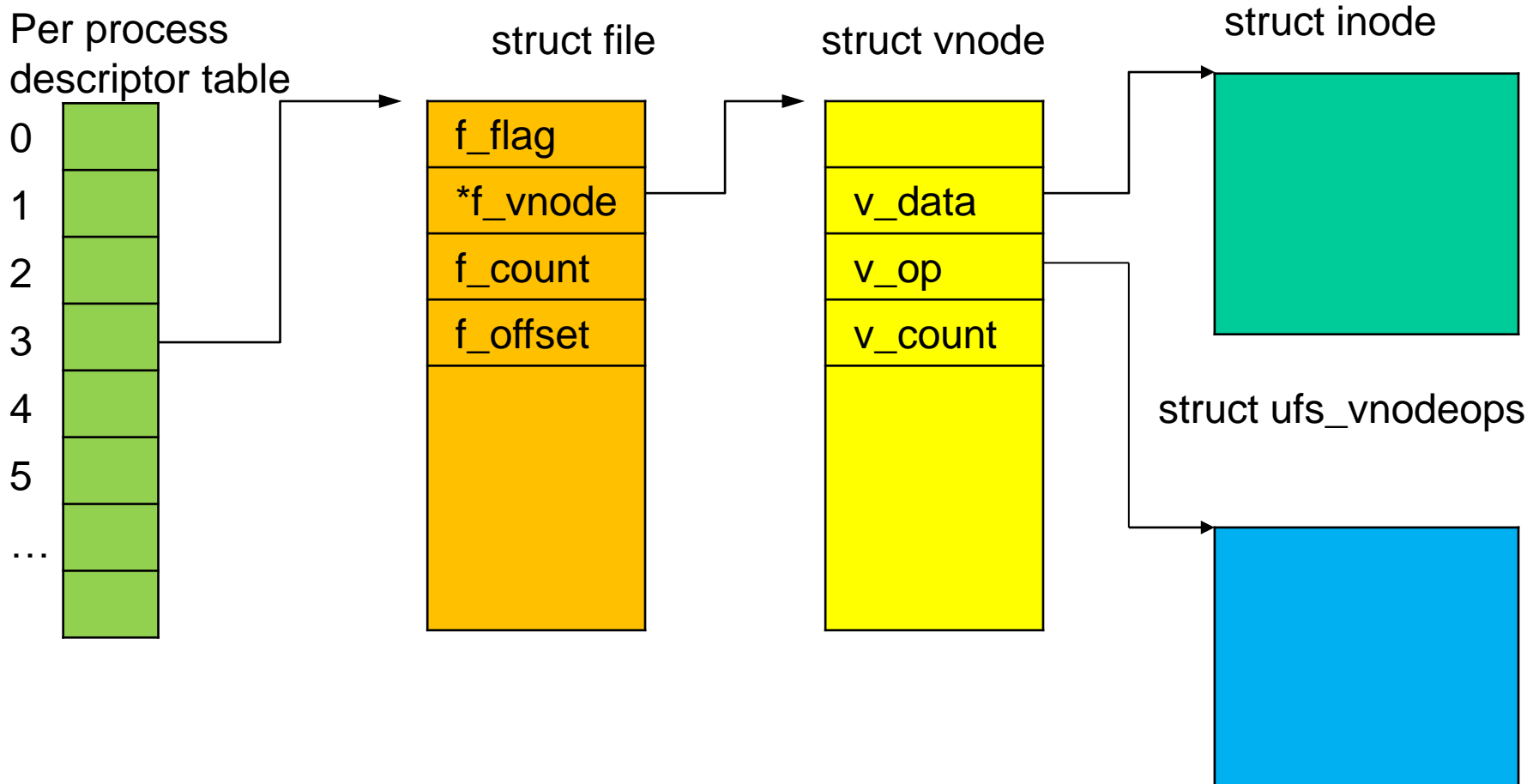
- The FS API is a VFS interface, rather than an interface to any specific type of file system

Linux Virtual File System

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- The Linux VFS handles the four types of objects:
 - The **inode object** structure represent an individual file
 - The **file object** represents an open file
 - The **superblock object** represents an entire file system
 - A **dentry object** represents an individual directory entry (file or directory names); cached for faster file/directory access.
- LVFS file operations include (see **struct file_operations** in **/usr/include/linux/fs.h**)
 - int open(. . .) — Open a file
 - ssize_t read(. . .) — Read from a file
 - ssize_t write(. . .) — Write to a file
 - int mmap(. . .) — Memory-map a file

Data structures

- Data structures which are used to map a file descriptor to a file (here: for UFS file system)

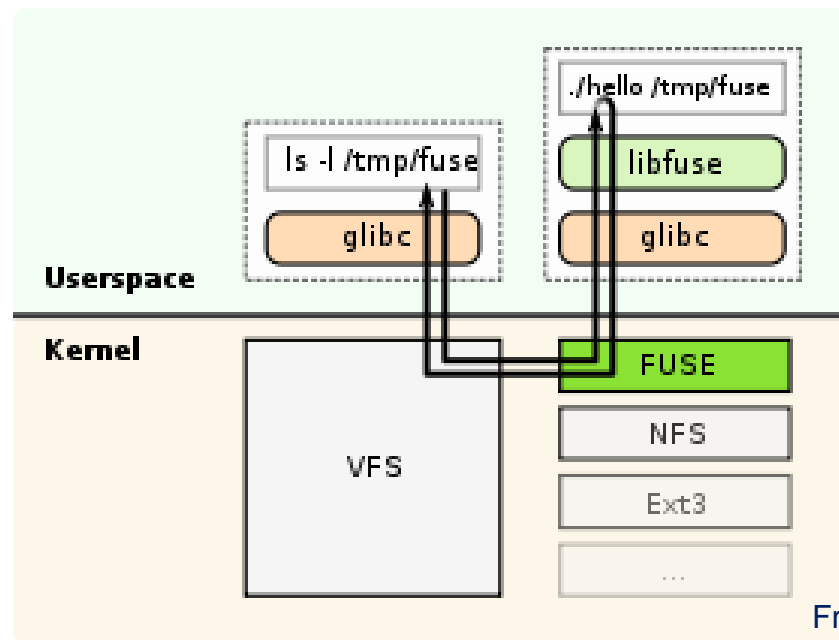


VFS extensions - FUSE

- Filesystem in **Userspace** (**FUSE**) is a software interface for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. File system code is run in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces.

A FUSE file system is typically implemented as a standalone application that links with libfuse library.

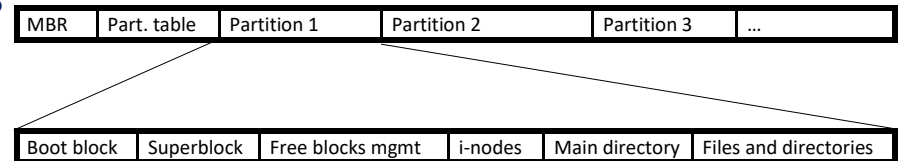
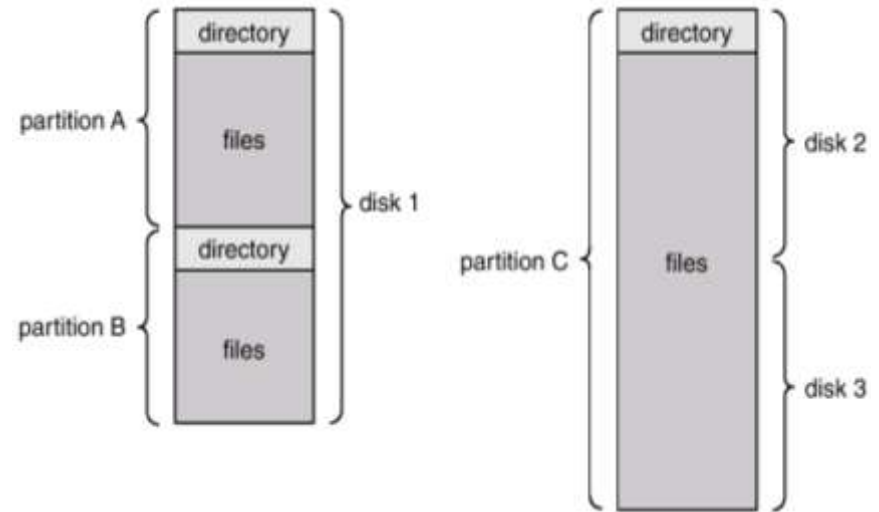
libfuse offers two APIs: a "high-level", synchronous, and a "low-level", asynchronous



From [Wikipedia](#)

Towards file system implementations

- Memory of storage devices is split into partitions (volumes, slices, minidisks...).
- Disk or partition can be used **raw** – without a file system or **formatted** with a file system. Disks or partitions can be **RAID** protected against failure.
- Formatted partition contains
 - a header (superblock),
 - the main directory and
 - a storage space for sub-directories and files (ordinary and some FS-specific special)
- Partition layout depends on the particular file system used for formatting (e.g. ext4, NTFS). The layout is parameterized, and the parameters (e.g. cluster size) are stored in the partition header.



- MBR – Master boot record – reads partition table and initiates load of the OS from active partition
- Boot block – boots OS stored in the partition

Partition of a logical partition

- Memory blocks of a formatted logical partition are allocated to:
 - Folders (catalogues)
 - Files: ordinary and special (characteristic to the file system used)
 - Sets of blocks
 - Free (unused)
 - Bad (or reserved)
 - Journal (option)
- File system can use allocation unit, which is different from the native allocation size of the device.
- Maximum number of file system objects of each type can be limited; the limits are assigned at the partition formatting time.

Directory Implementations

- **Linear list** of file names with pointer to the data blocks
 - Simple to program
 - Time-consuming to execute
 - Linear search time
 - Could keep ordered alphabetically via linked list or use B+ tree
 - **Hash Table** – linear list with hash data structure
 - Decreases directory search time
 - **Collisions** – situations where two file names hash to the same location
 - Only good if entries are fixed size, or use chained-overflow method
 - **B+tree** or other tree structures
- Folders can be represented with special files, which are directly readable by the system only. In any case processes should perform folder-related operations only via system functions.

Memory allocation for files

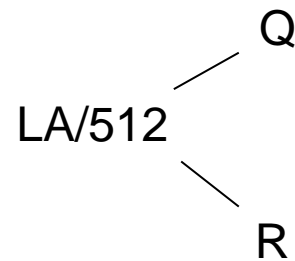
- File allocation method – determines how disk blocks are assigned to the set, which physically represents a file. We will discuss 3 basic allocation methods
 - **Contiguous allocation**
 - **Linked allocation**
 - **Indexed allocation**
- Real file systems mix the basic allocation methods. Other organizations are also used, e.g.:
 - **Heap** – to store a lot of data fast, before actual processing, based also on their content
 - **Extent** – a sequence of consecutive blocks, to improve I/O speed and reduce mgmt. overhead
 - **Slabs** – see slides on kernel memory allocation

Allocation Methods - Contiguous

- An 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 off-line (downtime)** or **on-line**

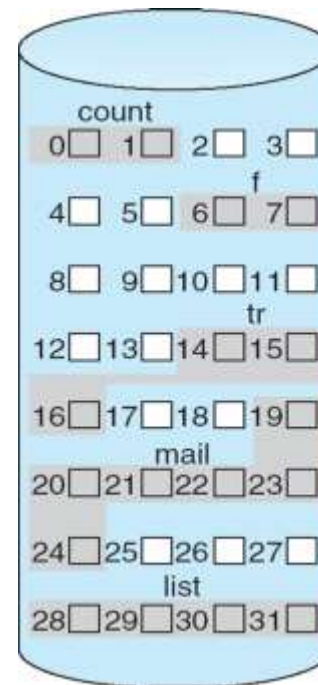
Contiguous Allocation

- Mapping from logical to physical



Block to be accessed = Q +
starting address

Displacement into block = R



directory

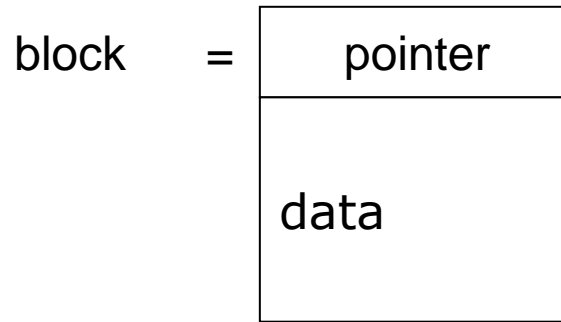
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-Based Systems

- Many newer file systems (i.e., Veritas File System , ext4, NTFS, ISO9660, UDF,XFS) use a modified contiguous allocation scheme
- **Extent-based file systems** allocate disk blocks in extents
- An **extent** is a contiguous sequence of disk blocks
 - Extents are allocated for file allocation
 - A file consists of one or more extents
 - Each extent is characterized by position of the first block and the number of blocks (no position of each block is stored – as is the typical case for non-extent-based systems). -> less memory used in File Control Block and allocation/deallocation is faster
- **Example.** UDF file system, used with DVDs, represents file size with 30 bits – which limits file size to 1GB. In effect movies (logical file) is represented with several extents, each of size $\leq 1\text{GB}$.

Linked Allocation

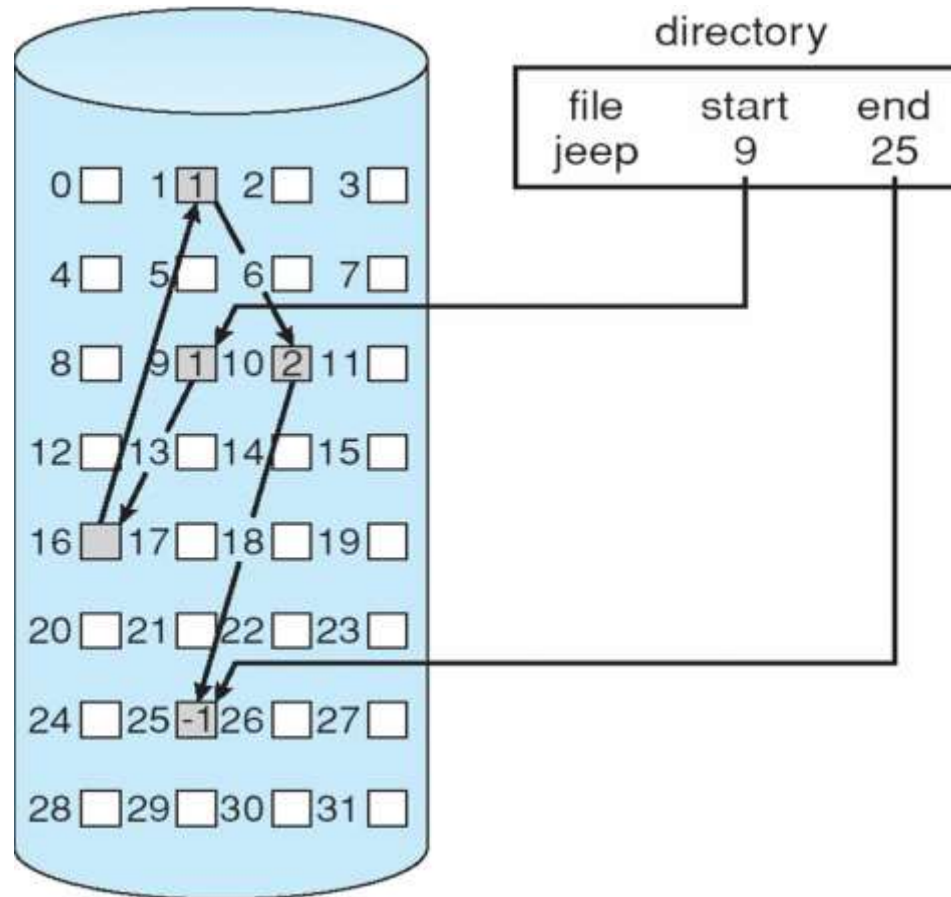
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk



- Features of linked allocation
 - Simplicity
 - No external memory fragmentation
 - No random access to blocks

Instead of single data blocks, clusters of data blocks can be used – to reduce memory overhead.

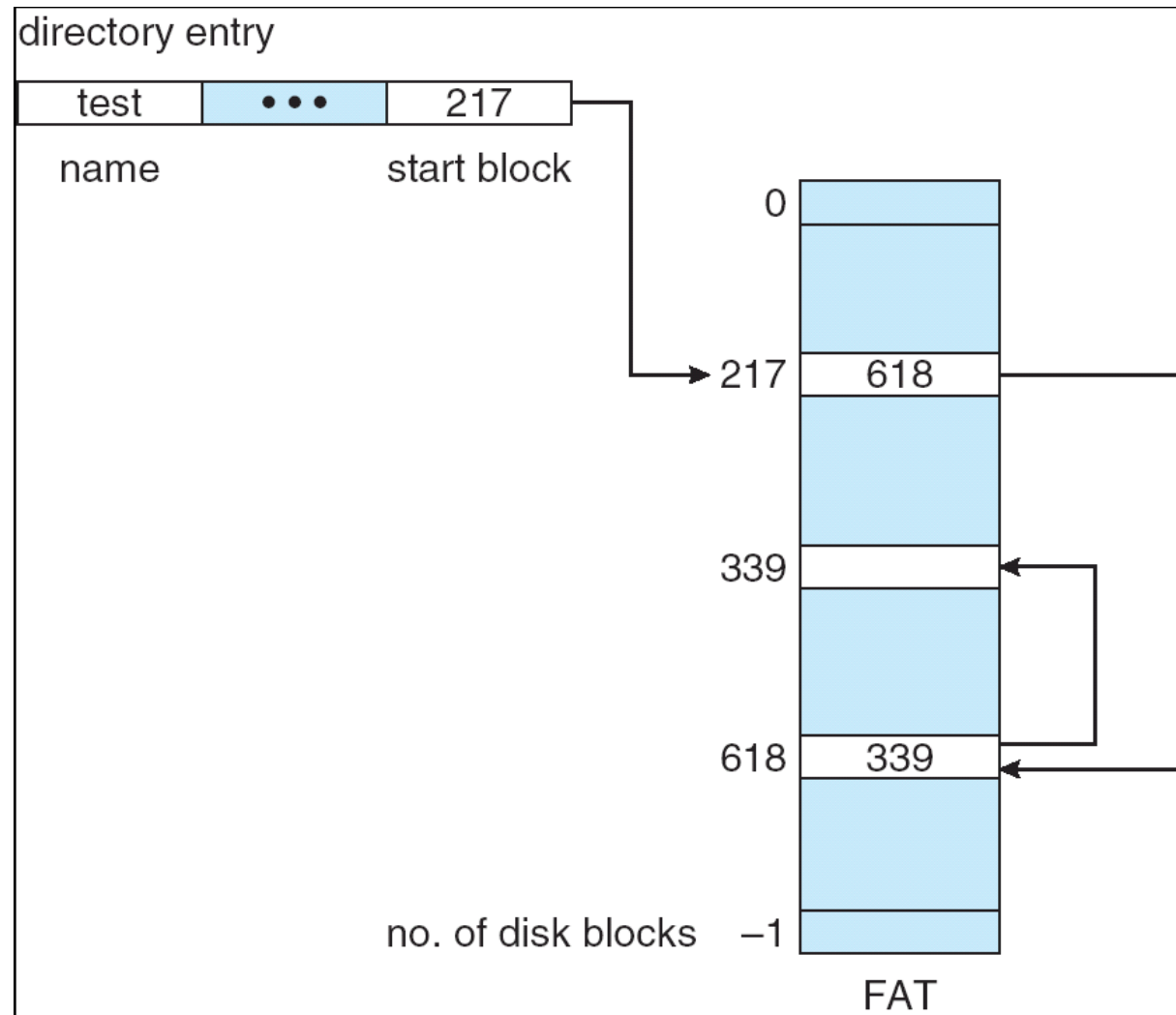
Linked Allocation - example



Allocation Methods – Linked (Cont.)

■ FAT (File Allocation Table) variation

- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation is simple

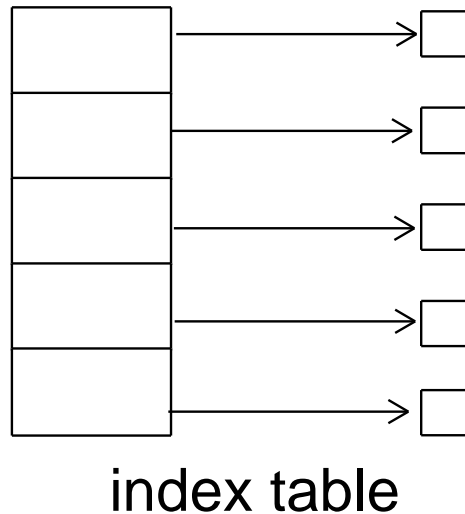


Allocation Methods - Indexed

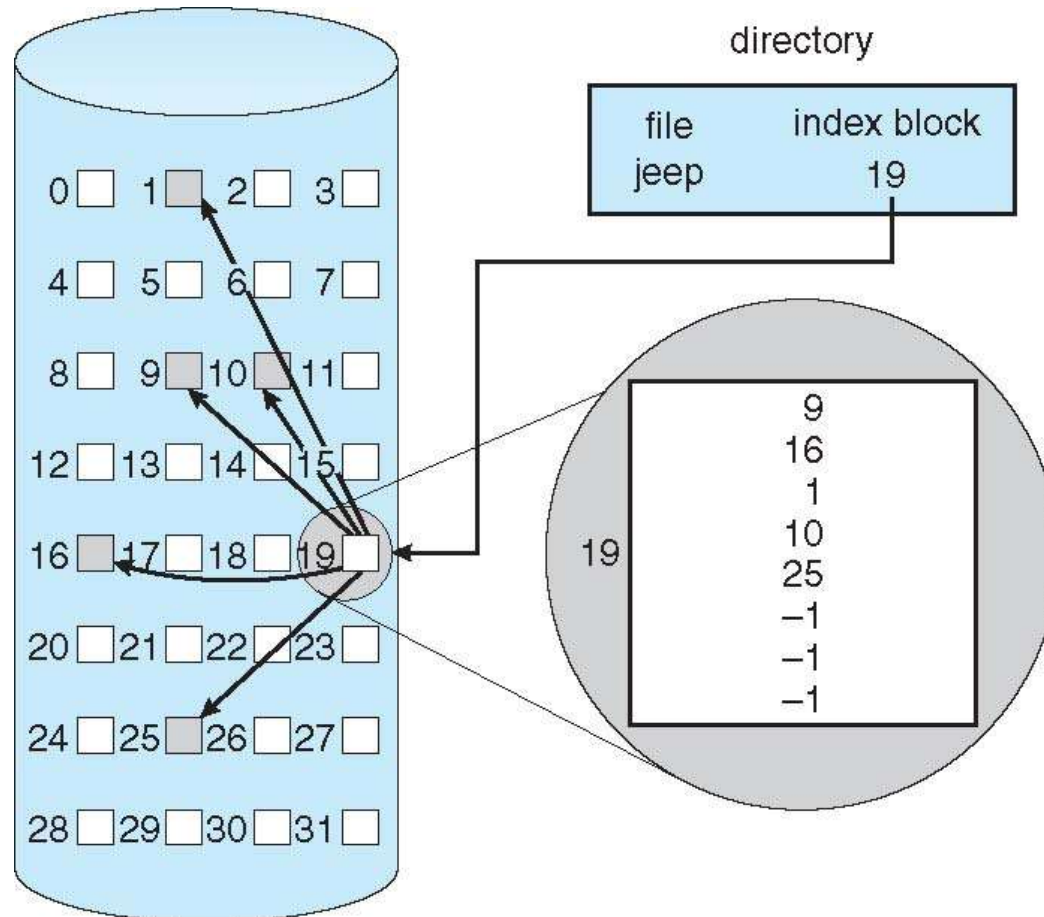
- Indexed allocation

- Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



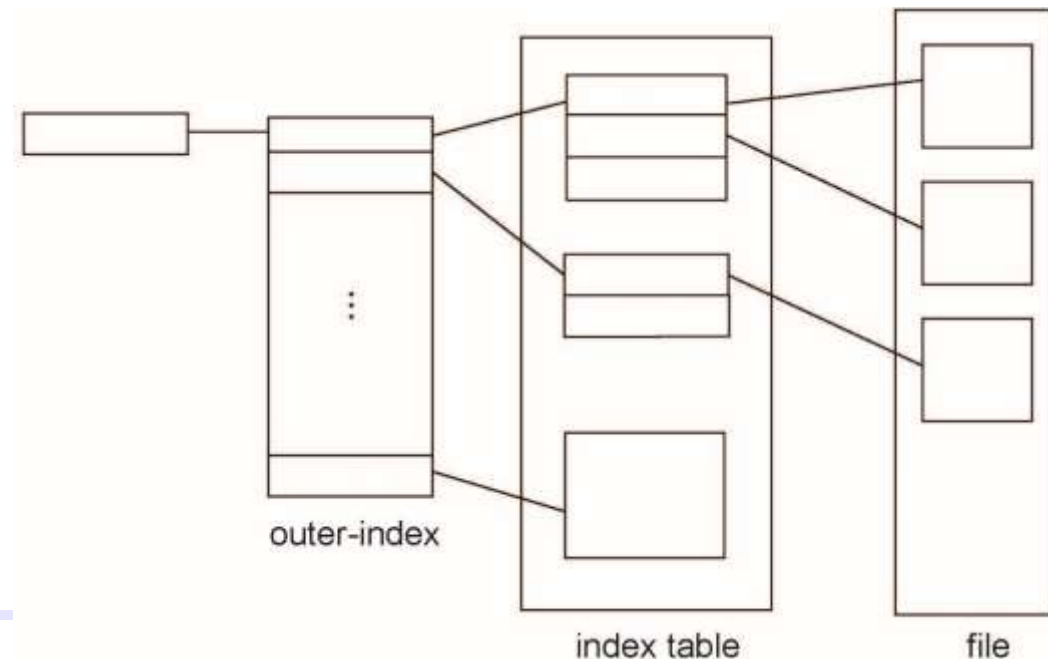
Example of Indexed Allocation



Indexed Allocation (Cont.)

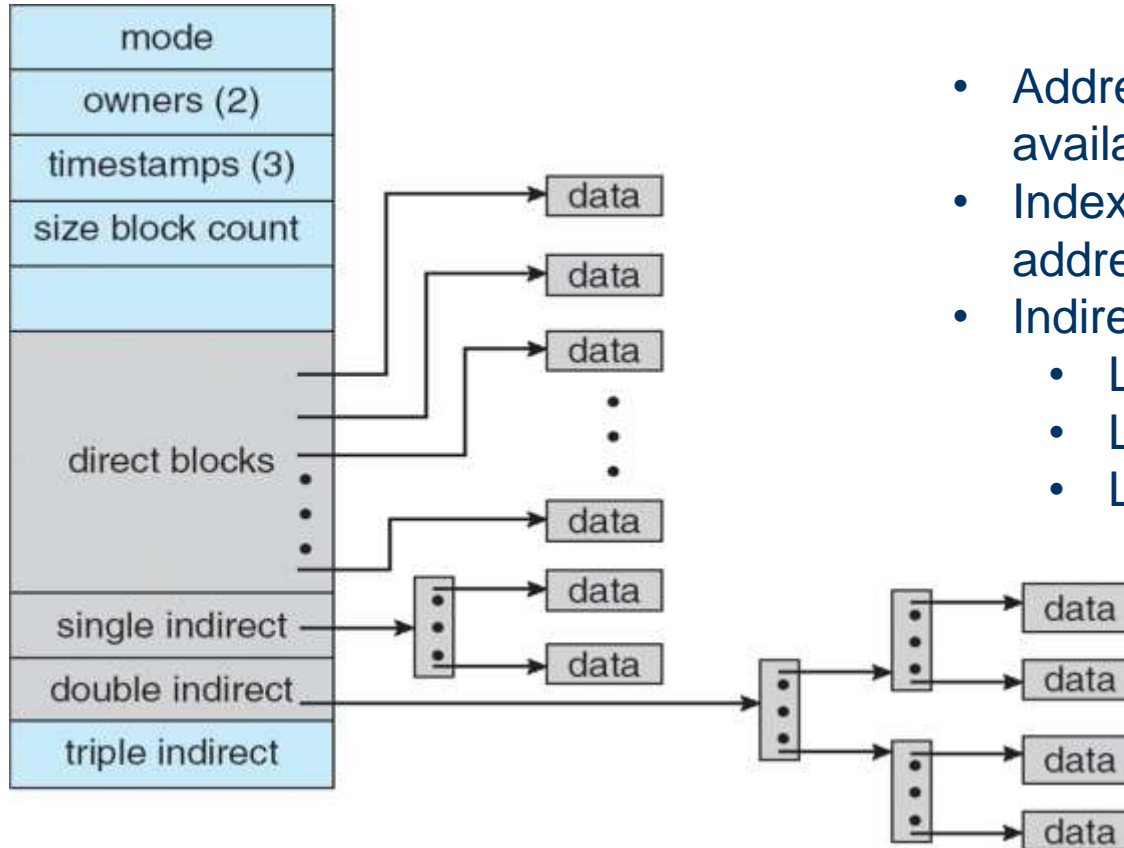
Features:

- Random access
- Dynamic access without external fragmentation, but there is overhead of index block
 - extra memory block for index
 - Extra time needed to find address of a block of a file (reading address from the index)
- Maximum file size is a product of a size of index block and size of file data block
- Two-level index makes significant increase of maximum file size possible
- Mapping from logical to physical address for a file of „unbounded” length is possible using linked scheme – **Link blocks of index tables**



Combined Scheme: UNIX/Linux

Example: 4K bytes per block, 32-bit addresses



- Addresses of 12 blocks available directly
- Index block holds 1024 addresses
- Indirect addressing
 - Level 1: 1024 addresses
 - Level 2: 1024^2 addresses
 - Level 3: 1024^3 addresses

- Using block clusters increases max. file size
- Some file systems (FFS/UFS) use block fragments (size: x512B) or blocks of different sizes (ext2), to reduce internal fragmentation.
- Allocation of related i-node/directory/file within a cylinder (or block) group improves access time

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
- **Bit vector** or **bit map** (n blocks)



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word of first “1” bit

Free-Space Management (Cont.)

- Bit map requires extra space

- Example:

- block size = 4KB = 2^{12} bytes

- disk size = 2^{40} bytes (1 terabyte)

- $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

- if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

- Cylinder/block groups – split disk partition into disjoint parts with own bit maps (FFS/UFS/ext2,...).

- Grouping

- Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)

- Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering

- Keep address of first free block and count of following free blocks (like in extents)
 - Free space list then has entries containing addresses and counts

Free-Space Management (Cont.)

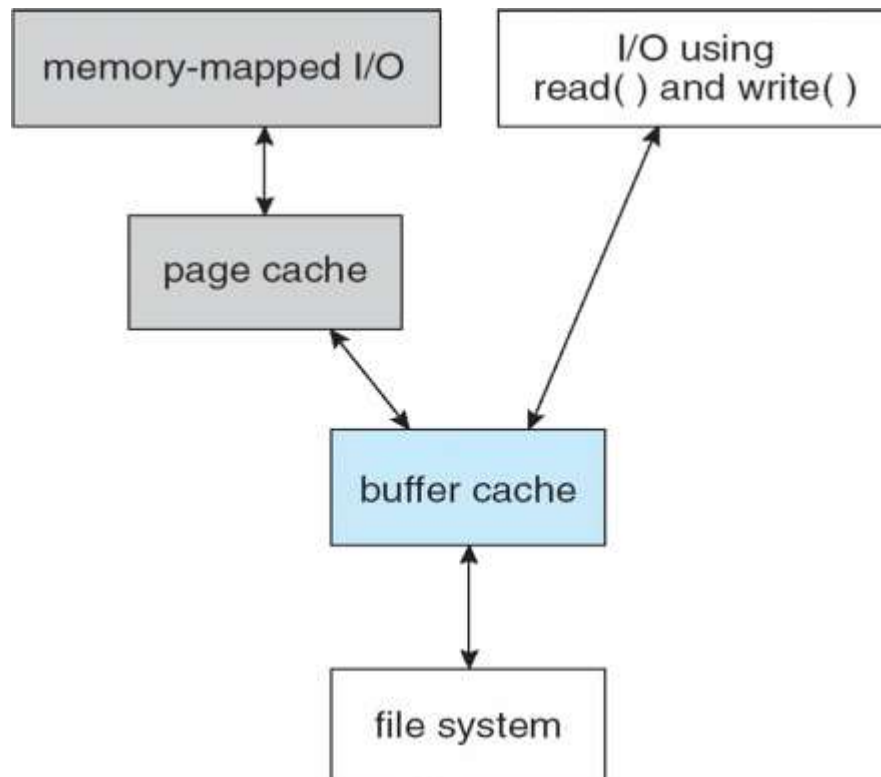
- Space Maps
 - Used in **ZFS**
 - Consider meta-data I/O on very large file systems
 - Full data structures like bit maps couldn't fit in memory -> thousands of I/Os
 - Divides device space into **metaslab** units and manages metaslabs
 - Given volume can contain hundreds of metaslabs
 - Each metaslab has associated space map
 - Uses counting algorithm to represent free space
 - But records (of allocations/freeing) to log file rather than file system
 - Log of all block activity, in time order, in counting format
 - Metaslab activity (allocation/freeing) -> load space map into memory in balanced-tree structure, indexed by offset
 - Replay log into that structure
 - Combine contiguous free blocks into single entry
 - Free-space list is being updated on disk as part of the transaction-oriented operations of ZFS. Subsequent block requests are stored in a log, etc.
 - In essence, the log plus the balanced tree make the free list

Efficiency and Performance

- Efficiency dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures
- Performance
 - Keeping data and metadata close together
 - **Buffer cache** – separate section of main memory for frequently used blocks
 - **Synchronous** writes sometimes requested by apps or needed by OS
 - No buffering / caching – writes must hit disk before acknowledgement
 - **Asynchronous** writes more common, buffer-able, faster
 - **Free-behind** and **read-ahead** – techniques to optimize sequential access
 - Reads frequently slower than writes

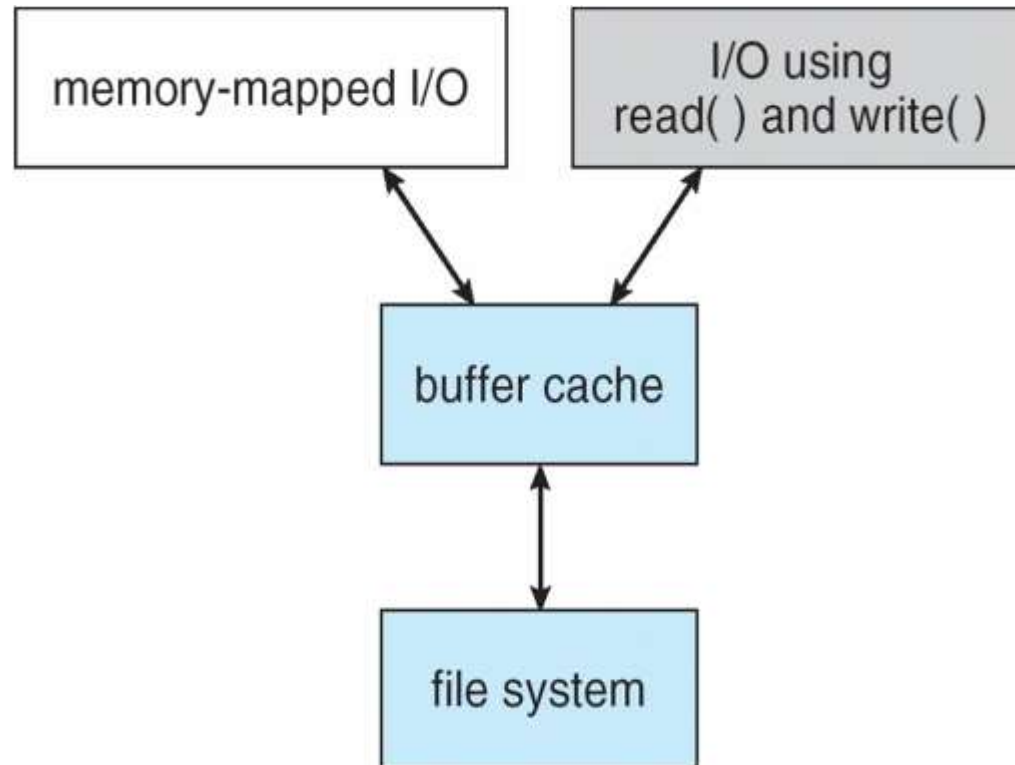
I/O Without a Unified Buffer Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache



I/O Using a Unified Buffer Cache

- A **unified buffer cache** uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
- But which caches get priority, and what replacement algorithms to use?



Remarks

- Adding instructions to the execution path to save one disk I/O is reasonable. Very approximate analysis:
 - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
 - http://en.wikipedia.org/wiki/Instructions_per_second
 - Typical disk drive at 250 I/O operations per second (IOPS)
 - $159,000 \text{ MIPS} / 250 = 630$ million instructions during one disk I/O
 - Fast SSD drives can provide more than 60,000 IOPS (<https://en.wikipedia.org/wiki/IOPS>)
 - $159,000 \text{ MIPS} / 60,000 = 2.65$ millions instructions during one disk I/O
- For some hard disks queuing of requests can even double transfer rate – due to internal optimization of head movements and operation order by the disk controller.
- SSD operations are very fast for limited time span

Example: What is an average I/O time for KiB block transfer on a 7200 RPM disk with a 5ms average seek time, 1Gib/sec transfer rate and 0.1ms controller overhead?

Ans.: 9.3 ms. Can you make your own calculations?

Log Structured File Systems

- **Log structured** (or **journaling**) file systems record each metadata update to the file system as a **transaction**
- All transactions are written to a log
 - A transaction is considered committed once it is written to the log (sequentially)
 - Sometimes to a separate device or section of disk
 - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
 - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed. Operations of aborted transactions (incomplete writes to the log) have to be undone though.
- Faster recovery from crash, removes chance of inconsistency of metadata
- Log can be implemented as a special file of journaling file system
- In some file systems it is possible to configure which FS operations are performed as transactions.

FS check and recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
 - Uses intrinsic redundancy of a file system to check for inconsistency
 - Can be slow and sometimes fails

Example consistency checking programs: fsck, chkdsk, sfc
- Use system programs to **back up** data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by **restoring** data from backup.
- Backup can be:
 - Physical (drive, partition) or logical (folders, files)
 - Full or incremental