File-System Implementation

Politecnico di Torino

Department of Control and Computer Engineering

System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN– Politecnico di Torino

Introduction

- File system provides the mechanism for on-line storage and access to file contents, including data and programs.
- File systems usually reside permanently on secondary storage, which is designed to hold a large amount of data.
- This chapter is concerned with issues surrounding file storage and access on the most common secondary-storage media, hard disk drives and nonvolatile memory devices.

Objective

- Describe the details of implementing local file systems and director structures
- Discuss block allocation and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Look at recovery from file system failures describe the WAFL file system as an example

File System Structure

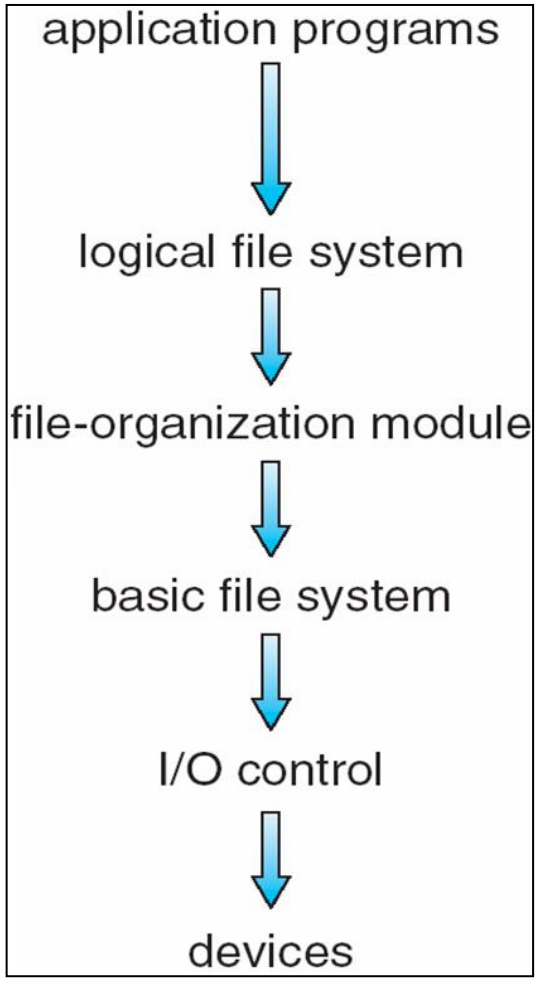
- Disks provide most of the secondary storage on which file systems are maintained.
  - A disk can be rewritten in place. It is possible to read a block from the disk, modify the block, and write it back into the same block.
  - A disk can access directly any block of information it contains.

File System Structure

- File systems provide efficient and convenient access to the storage device by allowing data to be stored, located and retrieved easily.
- A file system has two different design problems:
  - Defining how the file system should look to the user
    - Defining a file and its attributes, the operation allowed on a file, the directory structure for organizing files.
  - Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

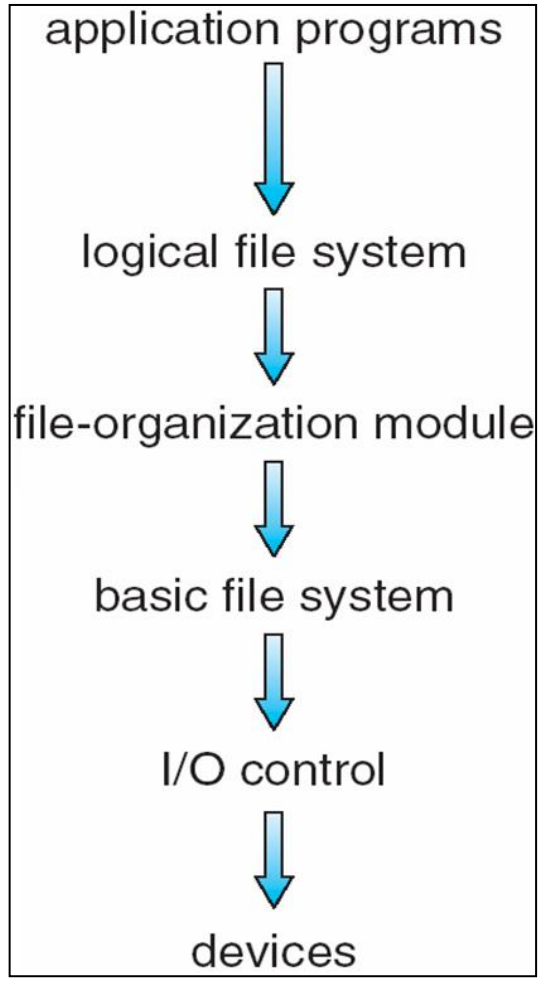
Layered File System

- File system is composed of many different levels:
  - I/O control
  - Basic file system
  - File-organization module
  - Logical file system
  - Application programs



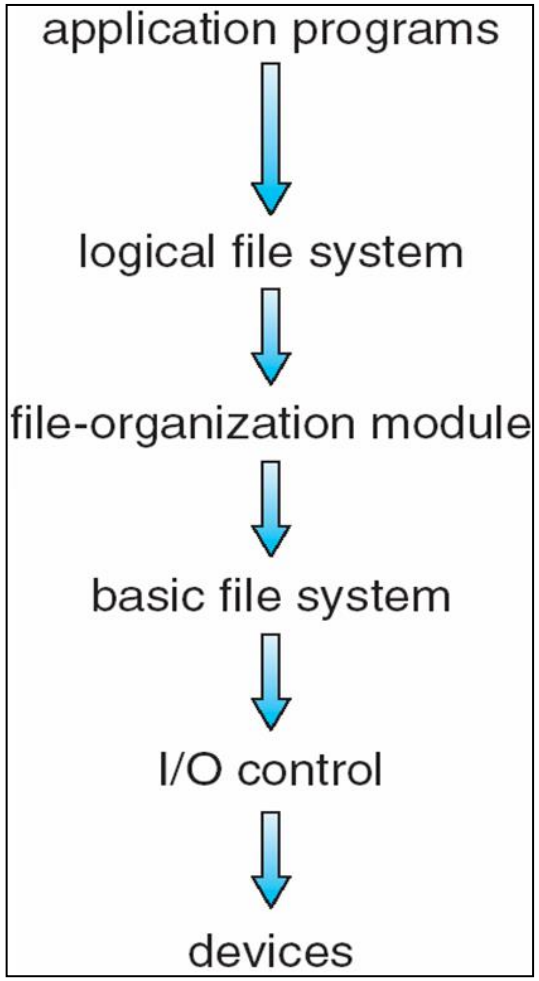
Layered File System

- File system is composed of many different levels:
  - I/O control
    - Device drivers and interrupt handlers to transfer information between the main memory and the disk system.
    - Device driver can be thought of a translator.
  - Basic file system
    - Issuing generic commands to the appropriate device driver to read and write blocks on the storage device.
    - Managing the memory buffers and caches holding various file system, directory and data blocks.
  - File-organization module
    - Knows about files and their logical blocks.
    - Each file's logical blocks are numbered from 0 (or 1) through N.
    - Including the free-space manager for tracking unallocated blocks and providing these blocks to the file-organization module when requested.



Layered File System

- File system is composed of many different levels:
  - I/O control
  - Basic file system
  - File-organization module
  - Logical file system
    - Managing **metadata** information including all of the file-system structure except the contents of the files.
    - Managing the directory structure to provide the file-organization module with the information needed.
    - Maintaining file structure via **file-control blocks** containing information about the file, such as ownership, permissions and location of the file contents.
  - Application programs



Layered File System

- Benefits of layer structure:
  - Duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems.
  - Each file system can have its own logical file-system and file-organization modules.
- Disadvantages:
  - Layering can introduce more operating-system overhead, which may result in decreased performance

File System Layers

- Many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has **UFS**, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with **extended file system** ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- **Boot control block** contains info needed by system to boot OS from that volume
  - If the disc does not contain an operating system, this block can be empty
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (superblock, master file table) contains volume details
  - Total number of blocks, number of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, stored in master file table

File System Implementation

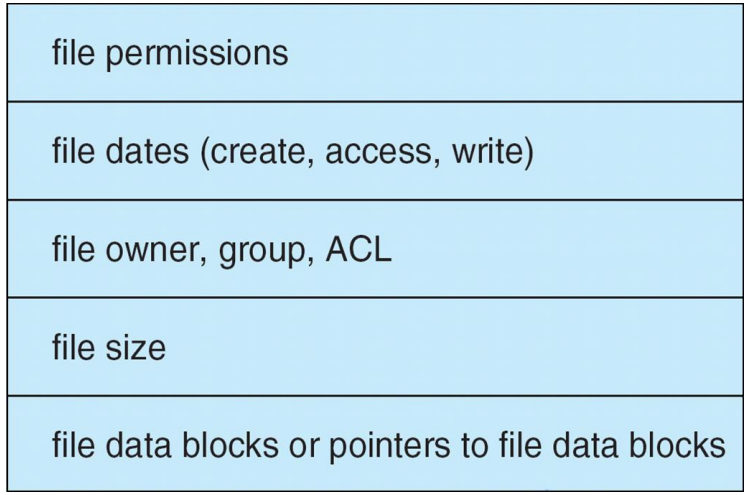
- Per-file **File Control Block (FCB)** contains many details about the file such as ownership, permissions and location of the file contents:
  - It has a unique identifier number to allow association with a directory entry.
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

In-Memory File System Structures

- The in-memory information is used for both file-system management and performance improvement via caching.
- The data are loaded at the mount time, updated during file-system operation and discarded at dismount.
- Several types structures may be included:
  - An in-memory **mount table** contains information about each mounted volume.
  - An in-memory directory structure cache holds the directory information of recently accessed directories.
  - The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
  - Buffers hold file-system blocks when they are being read from or written to a file system.

In-Memory File System Structures

- To create a new file system, a process calls the logical file system.
  - The logical file system knowns the format of the directory structures.
  - To create a new file, it allocates a new FCB.
  - The system then read the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the file system.



Partitions and Mounting

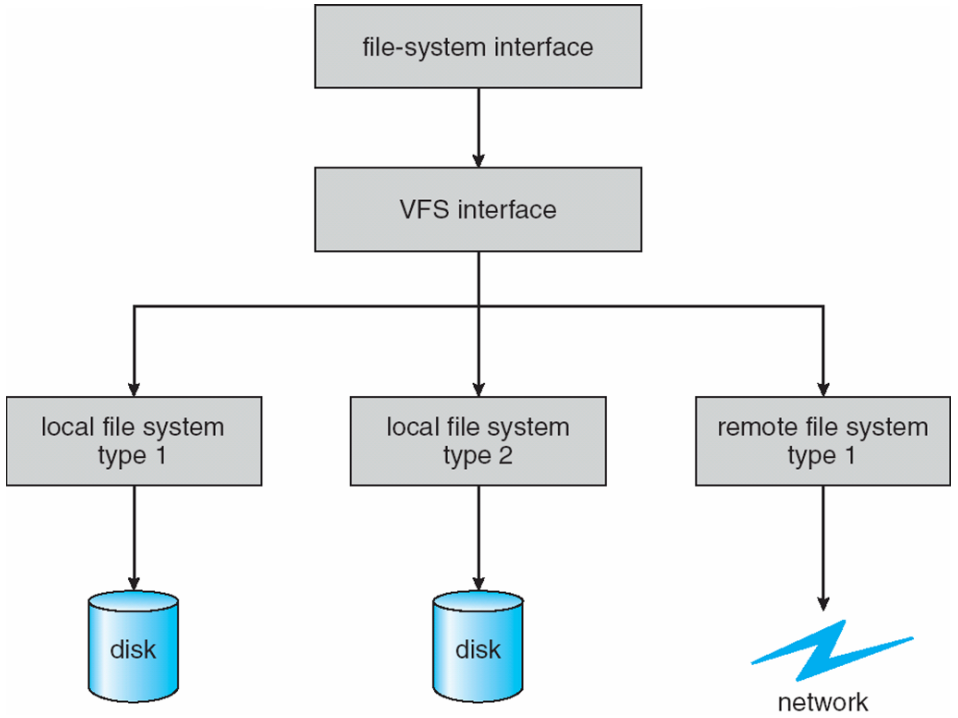
- Partition can be a volume containing a file system ("cooked") or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Virtual File Systems

- **Virtual File Systems (VFS)** on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

Virtual File Systems

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



Virtual File System Implementation

- For example, Linux has four object types:
  - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
    - • **`int open(. . .)`**—Open a file
    - • **`int close(. . .)`**—Close an already-open file
    - • **`ssize t read(. . .)`**—Read from a file
    - • **`ssize t write(. . .)`**—Write to a file
    - • **`int mmap(. . .)`**—Memory-map a file

Directory Implementation

- Directory allocation and directory management algorithms significantly affects the efficiency, performance and reliability of the file system.
  - Linear List
  - Hash Table

Directory Implementation

- Directory allocation and directory management algorithms significantly affects the efficiency, performance and reliability of the file system.
  - Linear List
    - Using the linear list of file names with pointers to the data blocks.
    - This method is simple to program but time-consuming to execute.
    - To create a new file, first searching the directory to be sure that no existing file has the same name.
    - Then, adding a new entry at the end of the directory.
    - To delete a file, searching the directory for the named file and then releasing the space allocated to it.
    - The disadvantage of a linear list of directory is that finding a file requires a linear search.
    - Directory information is used frequently and users will notice if access to it is slow.

Directory Implementation

- Directory allocation and directory management algorithms significantly affects the efficiency, performance and reliability of the file system.
  - Hash Table
    - A linear list stores the directory, but a hash data structure is also used.
    - The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
    - It greatly decrease the directory search time.
    - The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods

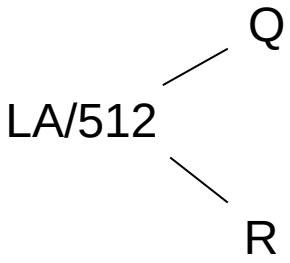
- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

Allocation Methods - Contiguous

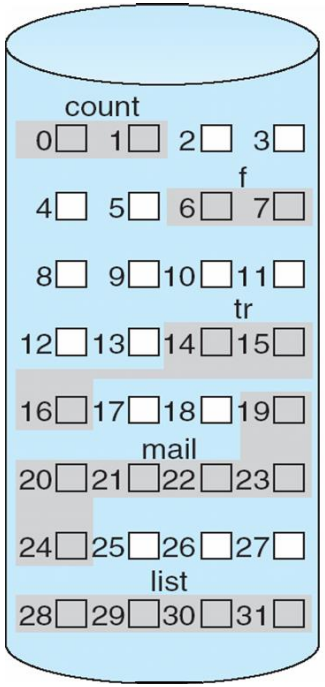
- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Accessing file:
    - For sequential access, the file system remembers the address of the last block references and read the next block.
    - For direct access to block *i* of a file that starts at block *b*, we can immediately access block *b+i*.

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

- **Problems of contiguous allocation:**
- Suffering from external fragmentation.
  - Files are allocated and deleted; the free storage space is broken into little pieces.
  - Largest amount of contiguous memory is not sufficient for a request.
  - **Solution**: copy an entire file system onto another device and copying the file back onto the original device by allocating contiguous space from this one large hole: **compacting** all free space into one large hole.

Extent-Based Systems

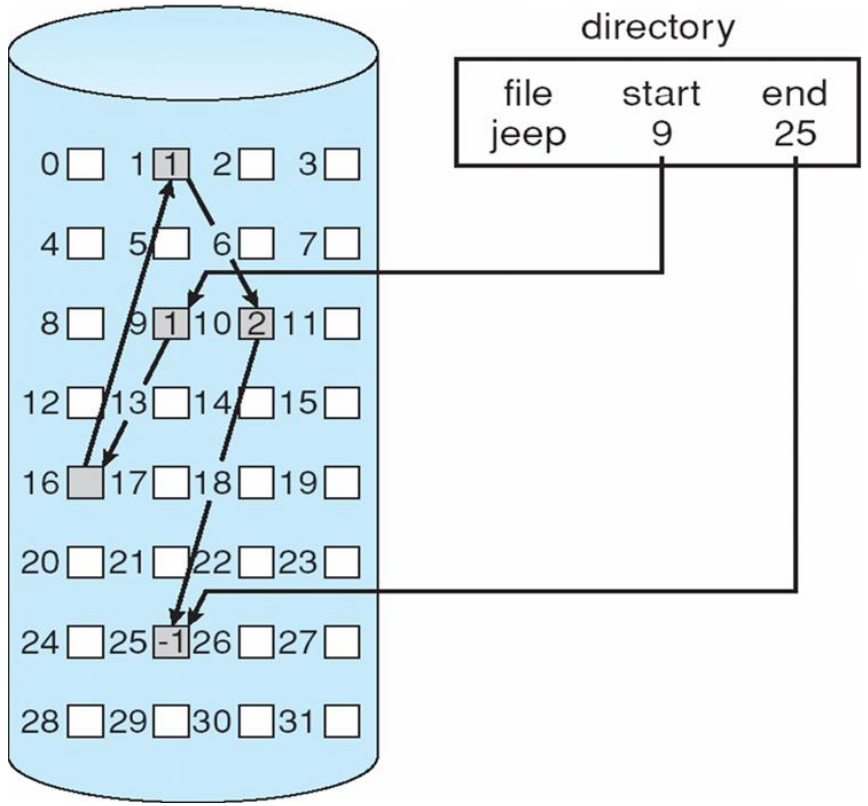
- **Problems of contiguous allocation:**
- Determining how much space is needed for a file.
  - When a file is created, the total amount of space it will need must be found and allocated which is difficult to estimate.
  - If we allocate too little space to a file, we may find the file cannot be extended. **Solutions**:
    1. The user program can be terminated with an error message and run the program again.
    2. Finding a larger hole, copying the contents of the file to the new space and release the previous space.
  - Time Consuming Solutions

Extent-Based Systems

- **Problems of contiguous allocation:**
- Determining how much space is needed for a file.
  - Modified contiguous-allocation scheme:
  - A contiguous chuck of space is allocated initially.
  - If the amount proves not to be large enough, another chunk of contiguous space, known as an **extent** is added.
  - The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

Allocation Methods - Linked

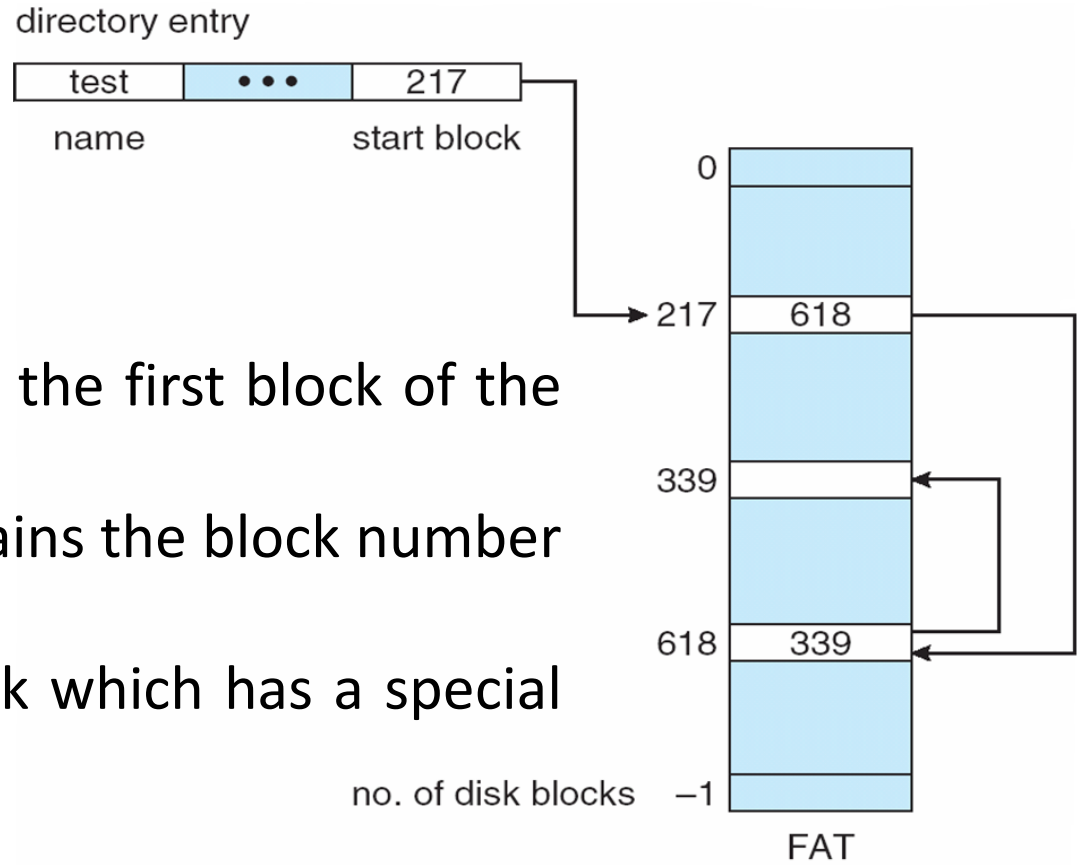
- **Linked allocation** – each file a linked list of storage blocks
  - File ends at nil pointer
  - No external fragmentation
  - **Each block contains pointer to next block**
  - No compaction, external fragmentation
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Reliability can be a problem
  - Locating a block can take many I/Os and disk seeks
  - Disadvantages: it can be used only for sequential access files. To find the i^{th} block of file, we must start from the beginning of the file.



Allocation Methods - Linked

- An important variation on linked allocation is the use of a **File Allocation Table (FAT)** variation

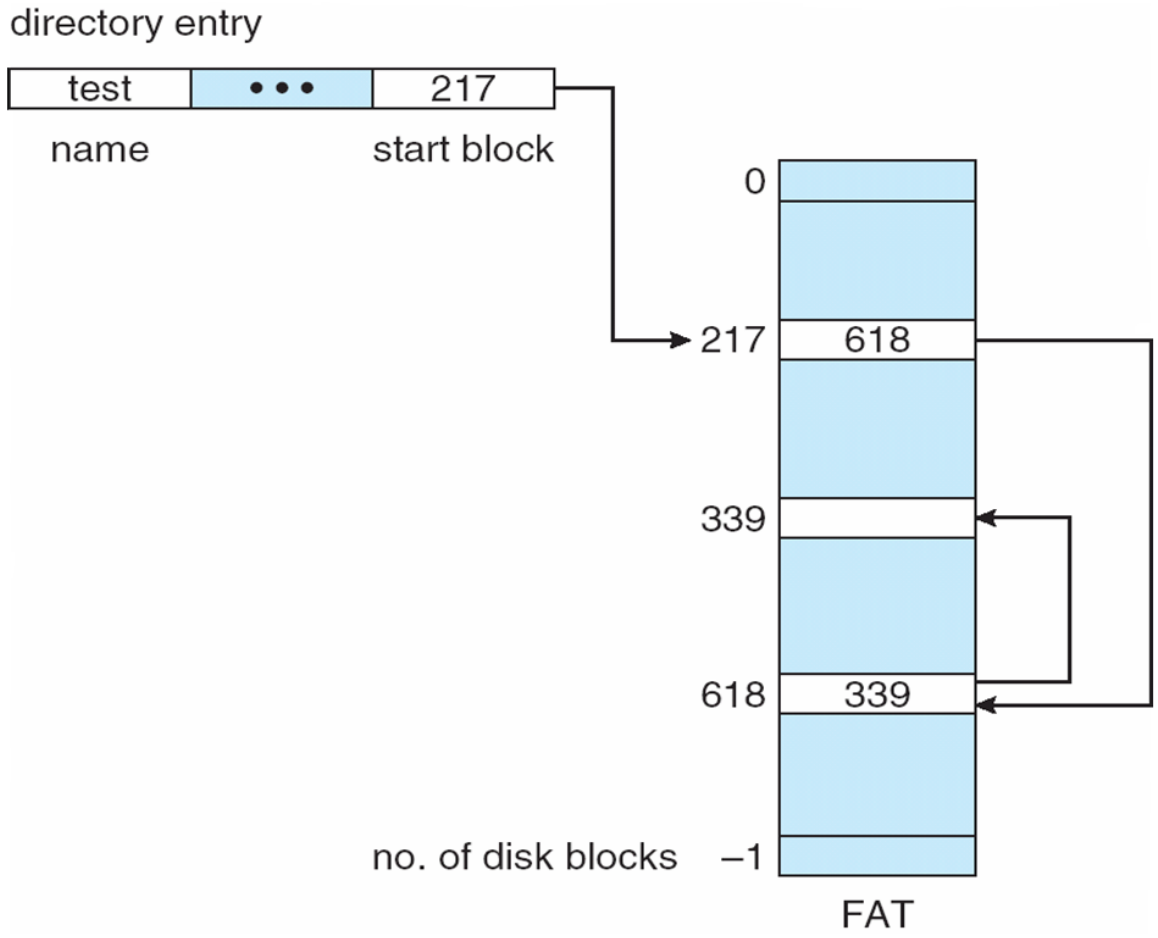
- A section of storage at the beginning of each volume is set aside to contain the table.
- The table has one entry for each block and is indexed by block number.



- The directory entry contains the block number of the first block of the files.
- The table entry indexed b that block number contains the block number of the next block in the file.
- This chain continues until it reached the last block which has a special end-of-file value as the table entry.

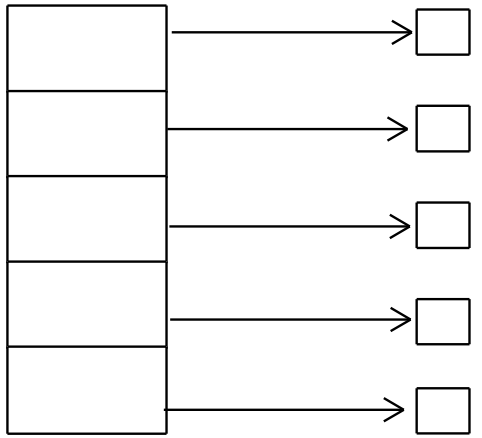
File Allocation Table

- An example of FAT structure fir a file consisting a disc blocks 217, 618, and 339.



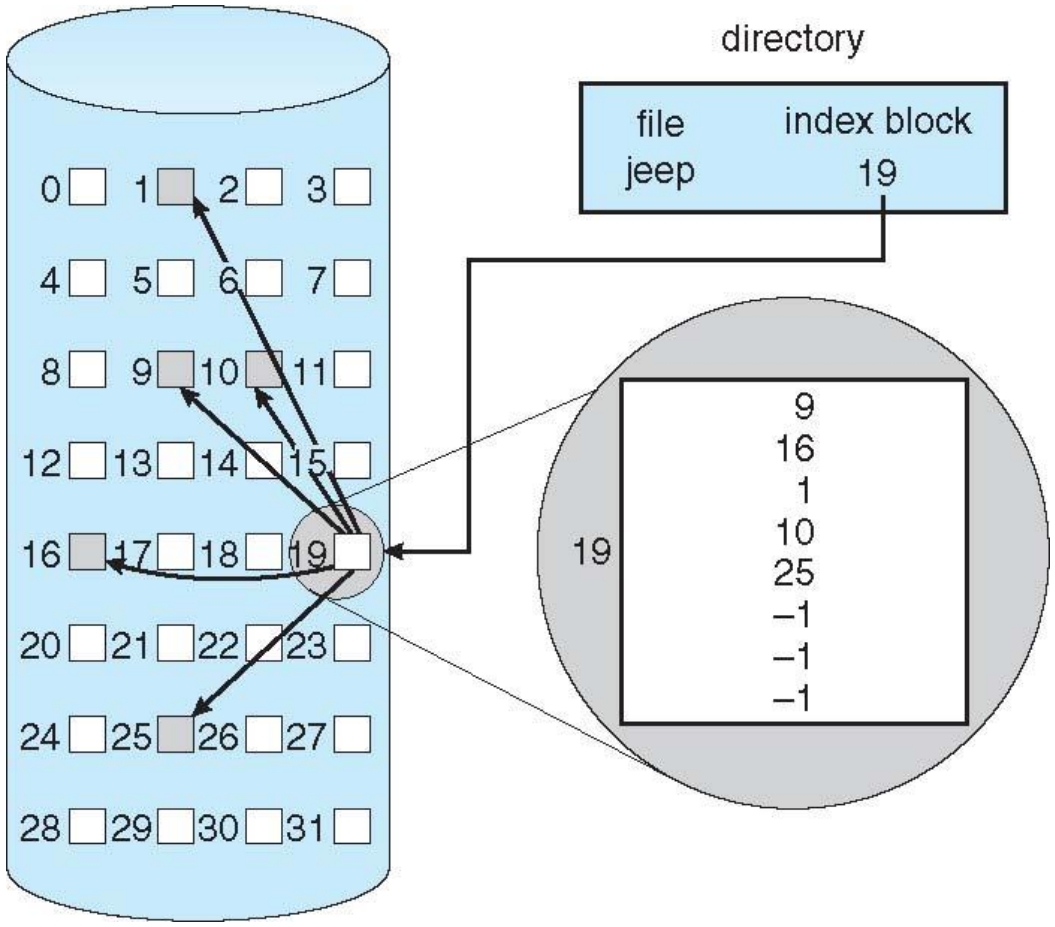
Allocation Methods - Indexed

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- In the absence of a FAT, linked allocation cannot support efficient direct access.
- **Indexed allocation** solve this problem by bringing all the pointers together into one location: the **index block**
  - Each file has its own index block(s) of pointers to its data blocks



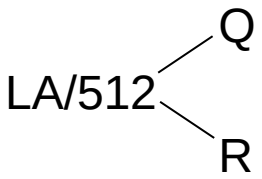
Example of Indexed Allocation

- Each file has its own index block which is an array of storage-block addresses.
- The i^{th} entry in the index block points to the i^{th} block of the file.
- The directory contains the address of the index block.
- To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.



Indexed Allocation

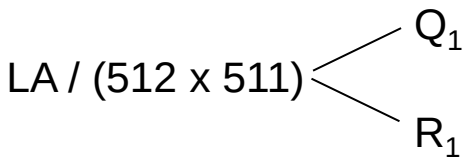
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index table



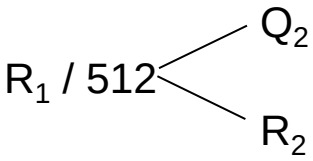
Q = displacement into index table
R = displacement into block

Indexed Allocation - Mapping

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)
- Linked scheme – Link blocks of index table (no limit on size)



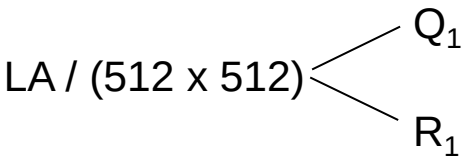
Q_1 = block of index table
R_1 is used as follows:



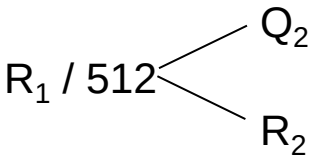
Q_2 = displacement into block of index table
R_2 displacement into block of file:

Indexed Allocation - Mapping

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

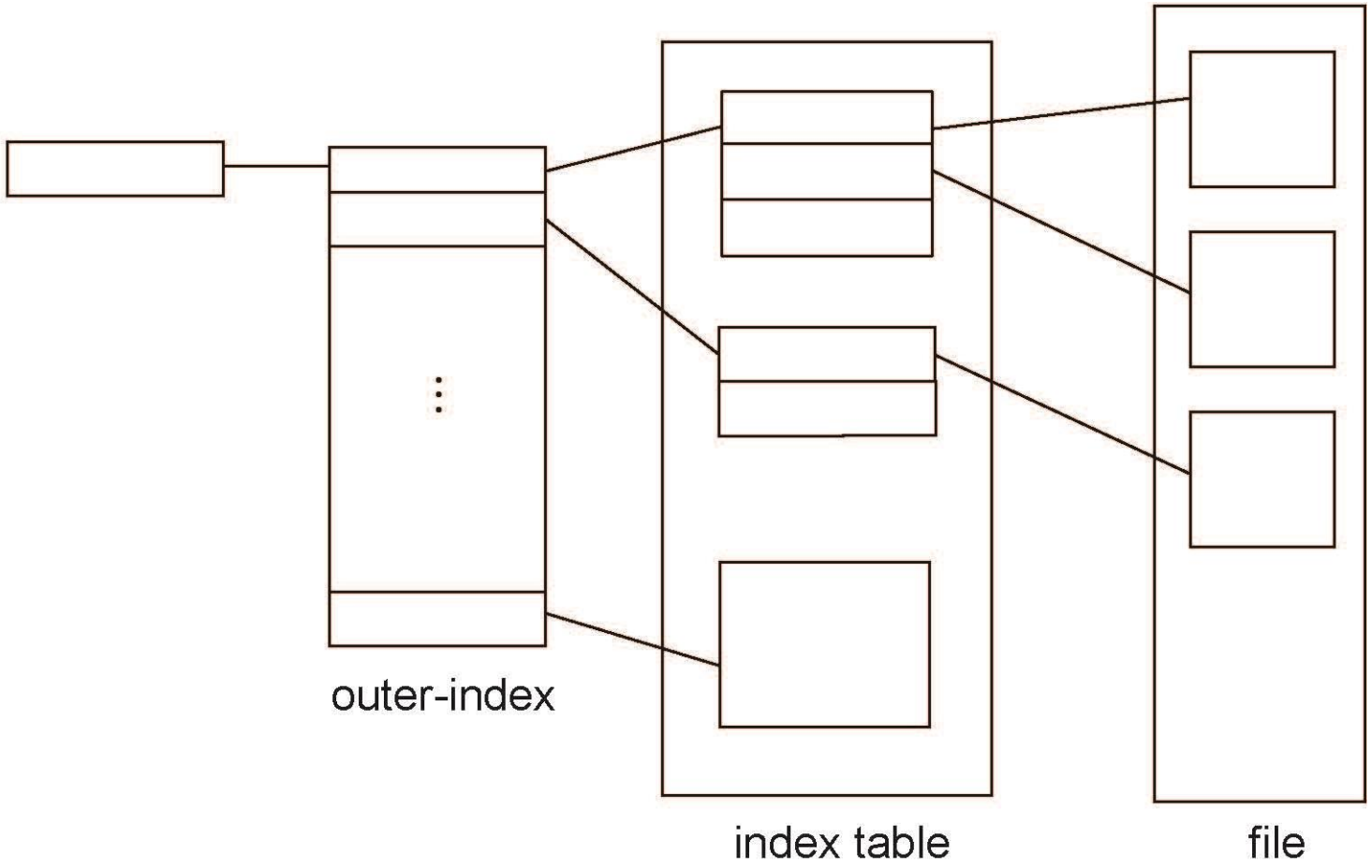


Q_1 = displacement into outer-index
R_1 is used as follows:



Q_2 = displacement into block of index table
R_2 displacement into block of file:

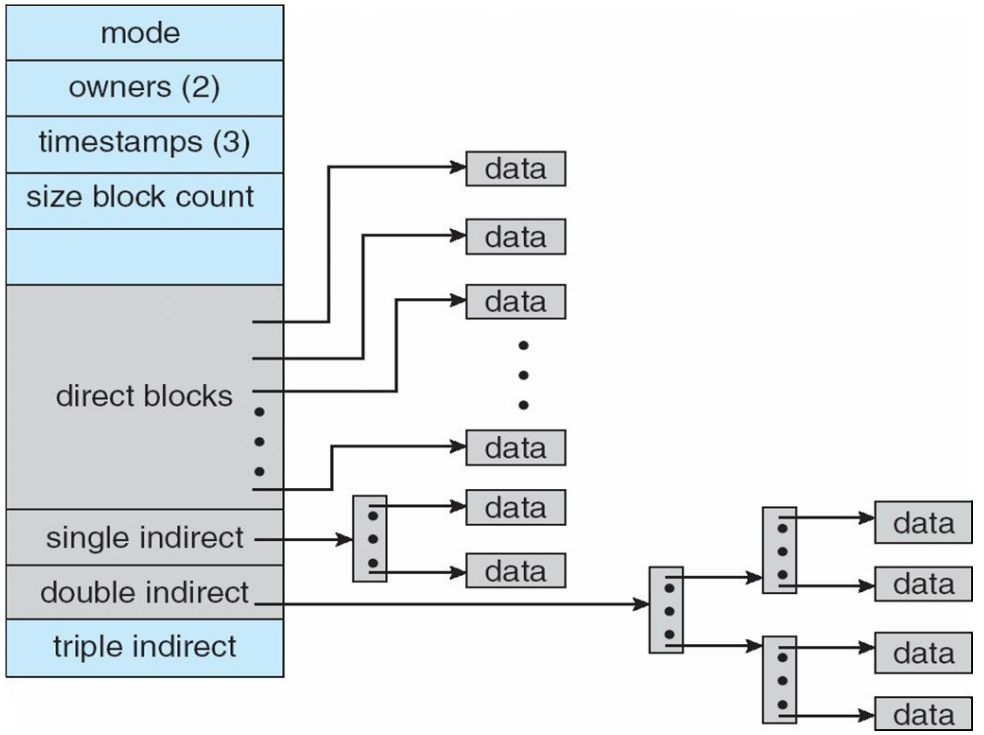
Indexed Allocation - Mapping



Combined Scheme: UNIX UFS

- TBD

4K bytes per block, 32-bit addresses



More index blocks than can be addressed with 32-bit file pointer

Performance

- The allocation methods vary in their storage efficiency and data-block access time.
- Before selecting an allocation method, we need to determine how the system will be used.
  - The system with mostly sequential access should not use the same method as a system with mostly random access.

Performance

- The allocation methods vary in their storage efficiency and data-block access time.
- Before selecting an allocation method, we need to determine how the system will be used.
  - **Contiguous allocation requires** only one access to get a block.
  - We can easily keep the initial address of the file in memory, we can calculate the address of the i^{th} block.
  - For linked allocation keep the address of the next block in memory and read it directly.

Performance

- The allocation methods vary in their storage efficiency and data-block access time.
- Before selecting an allocation method, we need to determine how the system will be used.
  - For **linked allocation** keep the address of the next block in memory and read it directly.
  - For sequential access it works, for direct access, an access to the i^{th} block might require i block read.

Performance

- The allocation methods vary in their storage efficiency and data-block access time.
- Before selecting an allocation method, we need to determine how the system will be used.
  - In **indexed allocation**, if the index block is already in memory, then the access can be made directly.
  - Keeping the index block in memory requires considerable space.
  - If this memory space is not available, then we should read first the index block and then the desired data block.

Performance

- The allocation methods vary in their storage efficiency and data-block access time.
- Before selecting an allocation method, we need to determine how the system will be used.
  - Therefore, some systems support direct-access files by using contiguous allocation and sequential-files by using linked allocation.
  - The types of access must be declared when the file is created.
  - Some systems combine contiguous allocation with indexed allocation:
    - Using contiguous allocation for small files
    - Automatically switching to an indexed allocation if the file grows large.

Performance

- Adding instructions to the execution path to save one disk I/O is reasonable
  - 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/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Free-space management

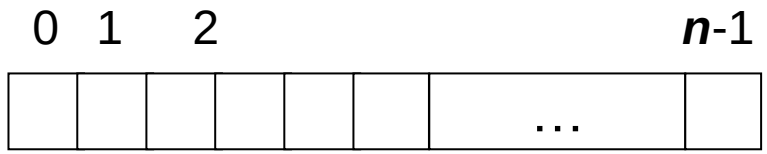
- Since storage space is limited, we need to reuse the space from deleted files for new files.
- File system maintains **free-space list** to track available blocks/clusters.
- To create a file system:
  - Search free space list for thee required amount of space and allocate that space to the new file.
  - The space is then removed from the free-space list.
  - When the file is deleted, its space is added to the free-space list.

Free-space management

- The free-space list is implemented as **bitmap** or **bit vector**.
- Each block is represented by 1 bit.
  - If the block is free, the bit is 1. if the block is allocated, the bit is 0.
  - For example, consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26, and 27 are free and the rest of the blocks are allocated. The free space bitmap would be:
    - 001111001111110001100000011100000…
  - Advantages: Simplicity and efficiency in finding the first free block

Free-space management

- One technique for finding the first free block is to sequentially check each word in bitmap to see whether that value is not 0.
  - A 0 valued-word contains only 0 bits and represents a set of allocated blocks.
  - The first non-0 word is scanned for the first 1 bit.



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

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

- Benefits: its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on disc.
- Drawbacks: inefficient unless the entire vector is kept in main memory. Keeping it in memory is possible for smaller devices but not necessarily for larger ones.
  - A 1.3 GB disk with 512-byte blocks would need a bitmap over 332KB to track its free blocks, although clustering blocks in groups of four reduces this number to around 83KB per disk.
  - A 1TB disk with 4KB blocks would require 32MB to store its bitmap. ($2^{40}/2^{12} = 2^{28}$ bits = 2^{25} bytes = 2^5MB)

Free-space management

- Bit map requires extra space
  - Example:

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

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

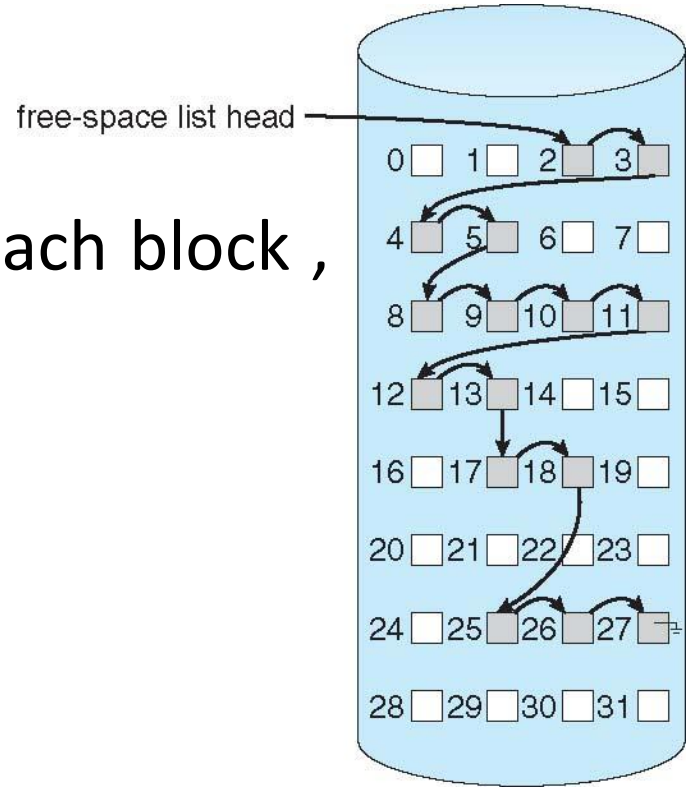
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

Linked Free Space List on Disk

- Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in file system and caching it in memory.
- The first block contains a pointer to the next free block and so on.



- Not efficient since to traverse the list, we must read each block , which requires substantial I/O times on HDDs.

Free Space Management

- 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
    - Free space list then has entries containing addresses and counts

Free Space Management

- 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
  - But records to log file rather than file system
    - Log of all block activity, in time order, in counting format
  - Metaslab activity -> load space map into memory in balanced-tree structure, indexed by offset
    - Replay log into that structure
    - Combine contiguous free blocks into single entry

Efficiency and Performance

- Efficiency depends 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

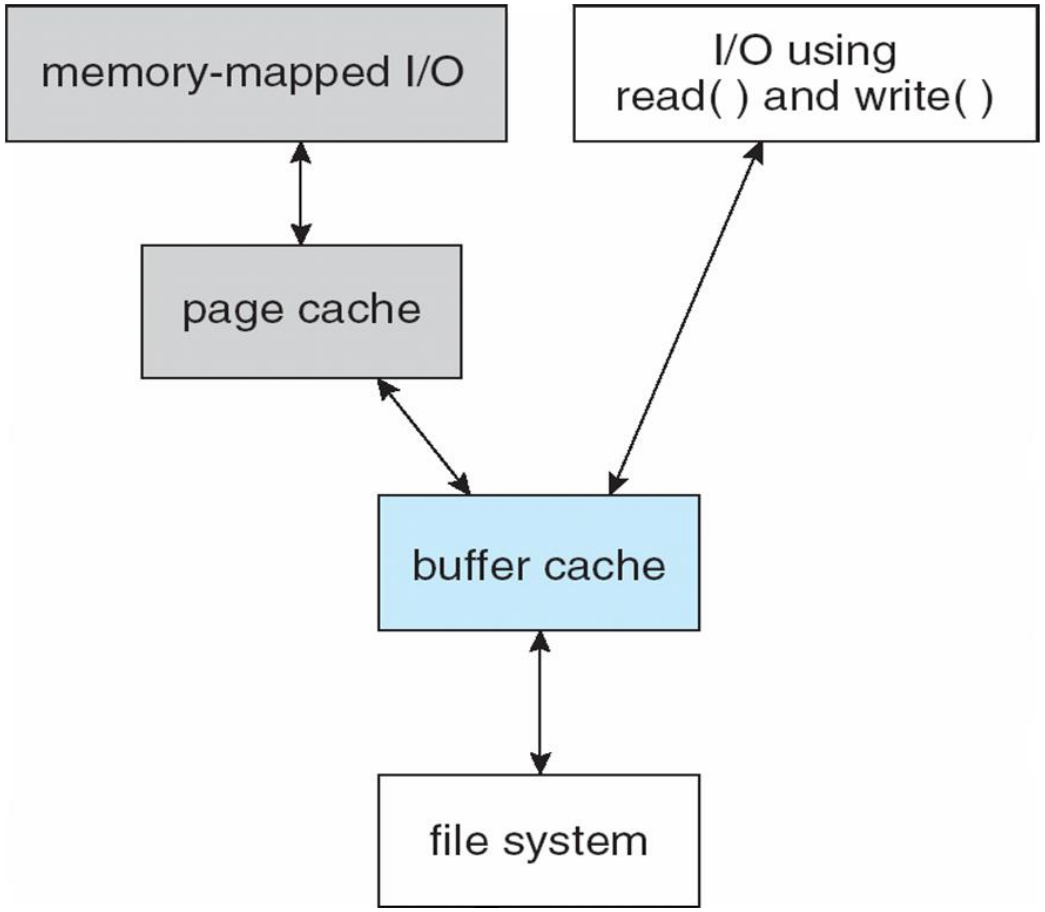
Efficiency and Performance

- 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

Page 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
- This leads to the following figure

I/O Without a Unified Buffer Cache

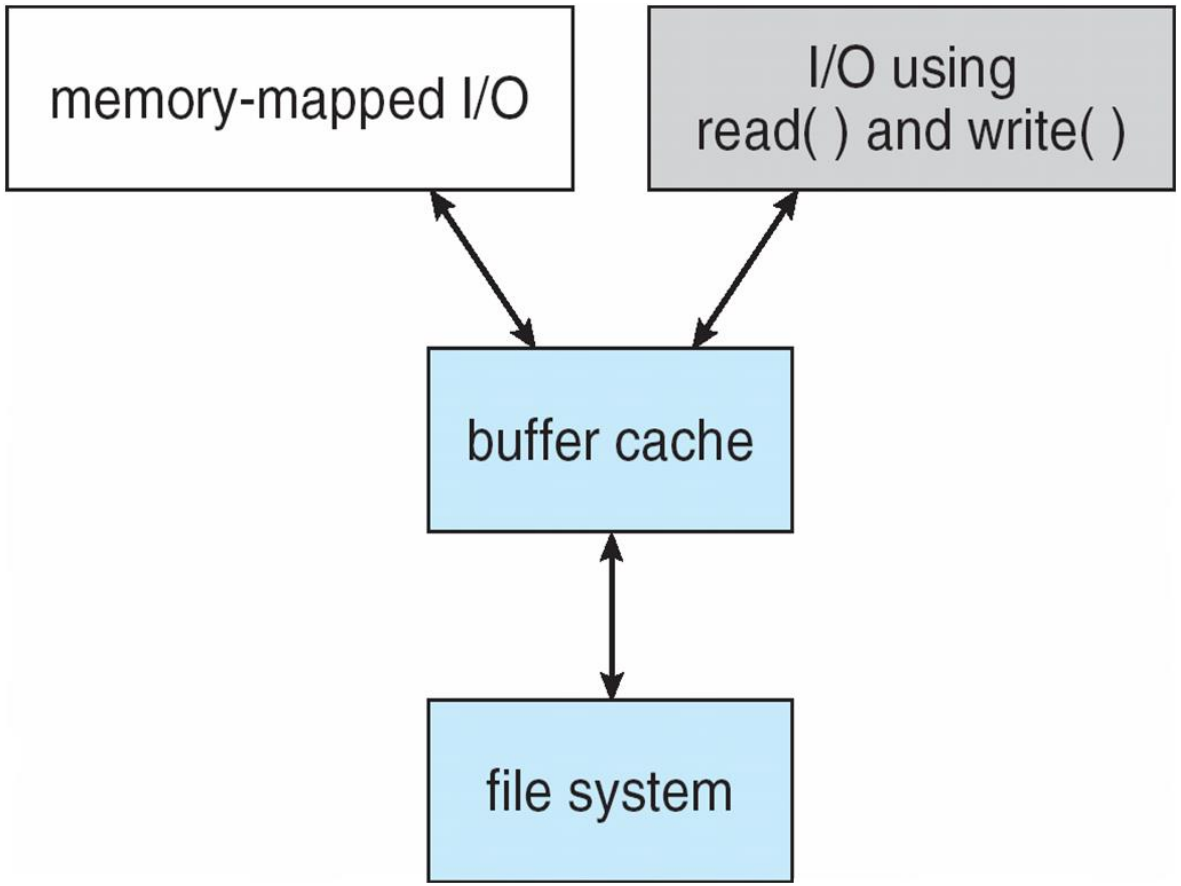


Unified Buffer Cache

- 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?

I/O Using a Unified Buffer Cache



Recovery

- Files and directories are kept in main memory and on the storage volume
- Care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency.
  - **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
  - 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

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
- Faster recovery from crash, removes chance of inconsistency of metadata

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs)

- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet

The Sun Network File System (NFS)

- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

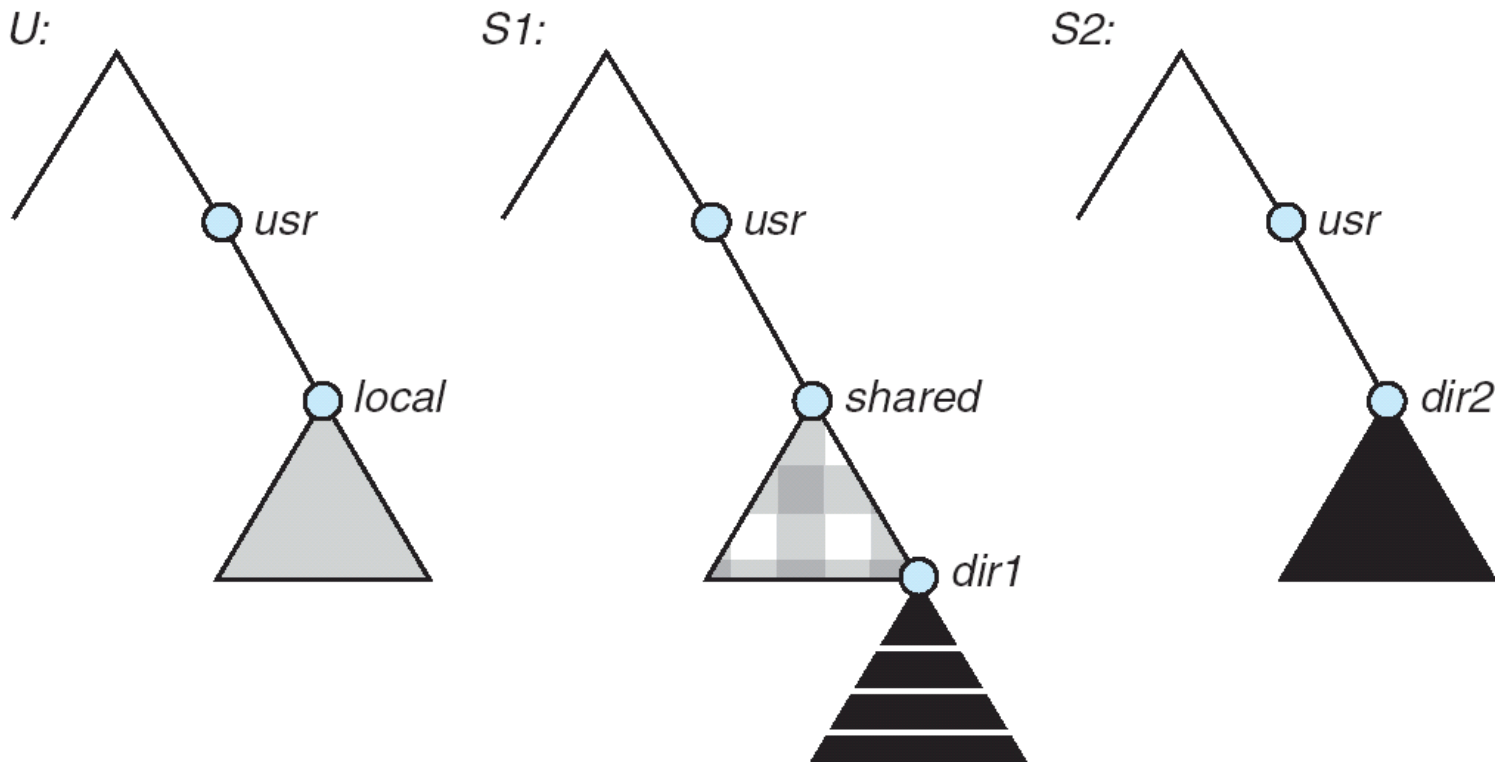
The Sun Network File System (NFS)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media

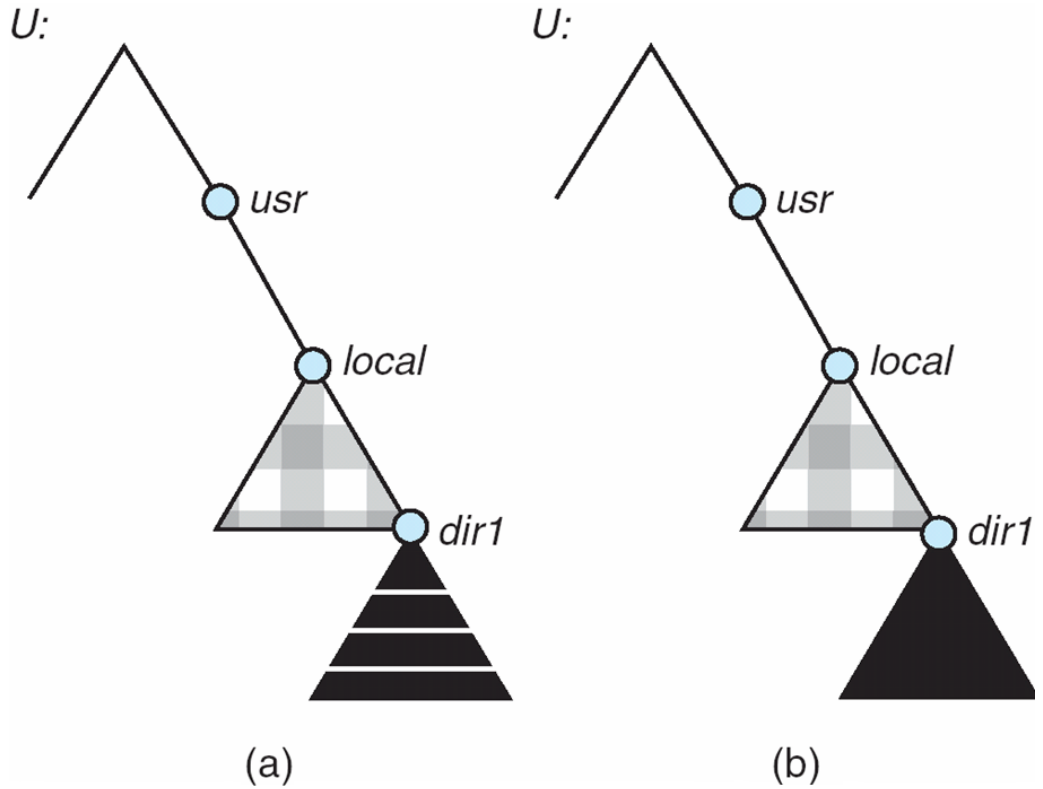
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces

- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services

Three Independent File Systems



Mounting in NFS



Mounts

Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

NFS Mount Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

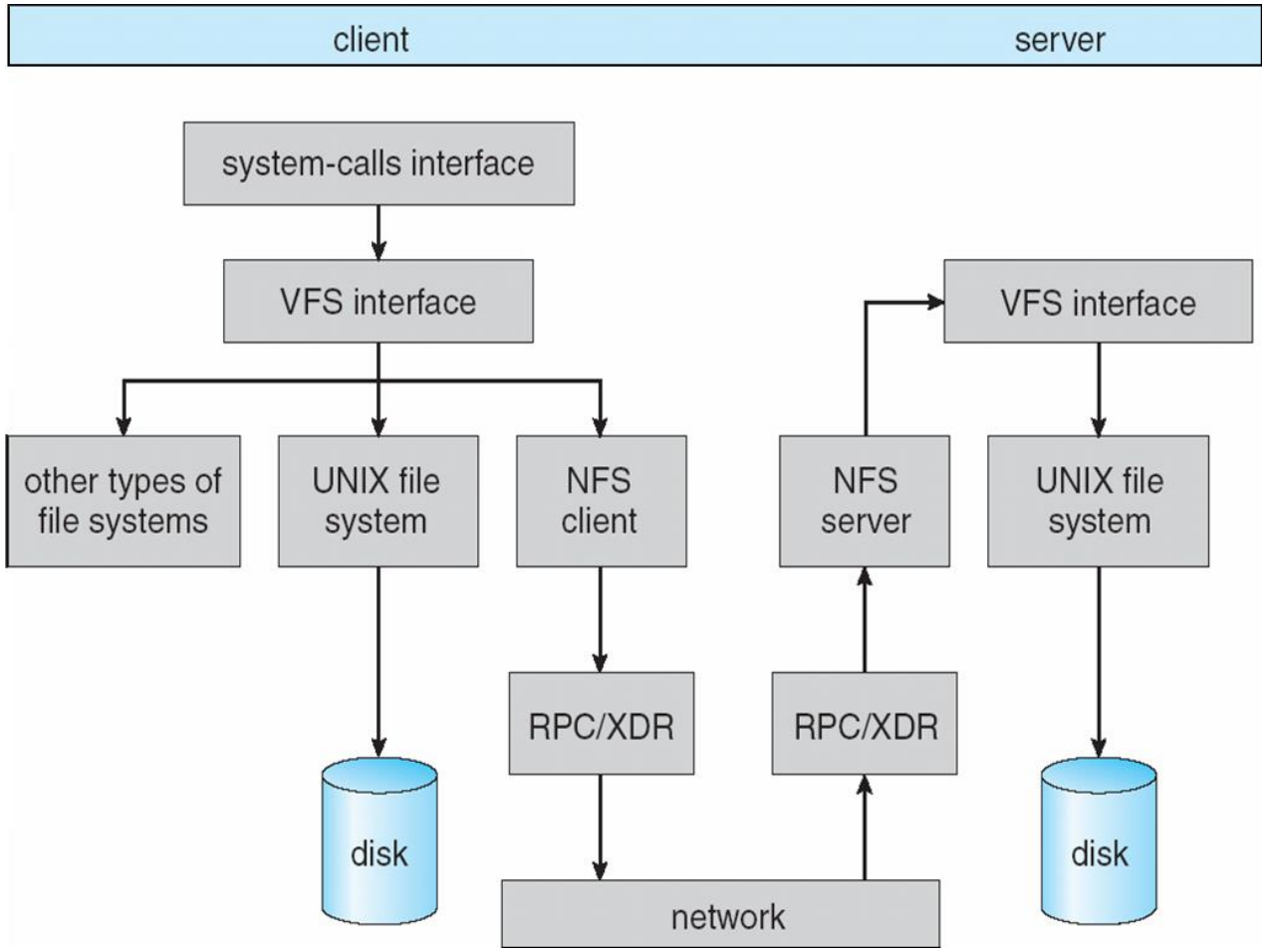
Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open, read, write**, and **close** calls, and **file descriptors**)

- Virtual File System (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests

- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture



NFS Path-Name Translation

- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode

- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names

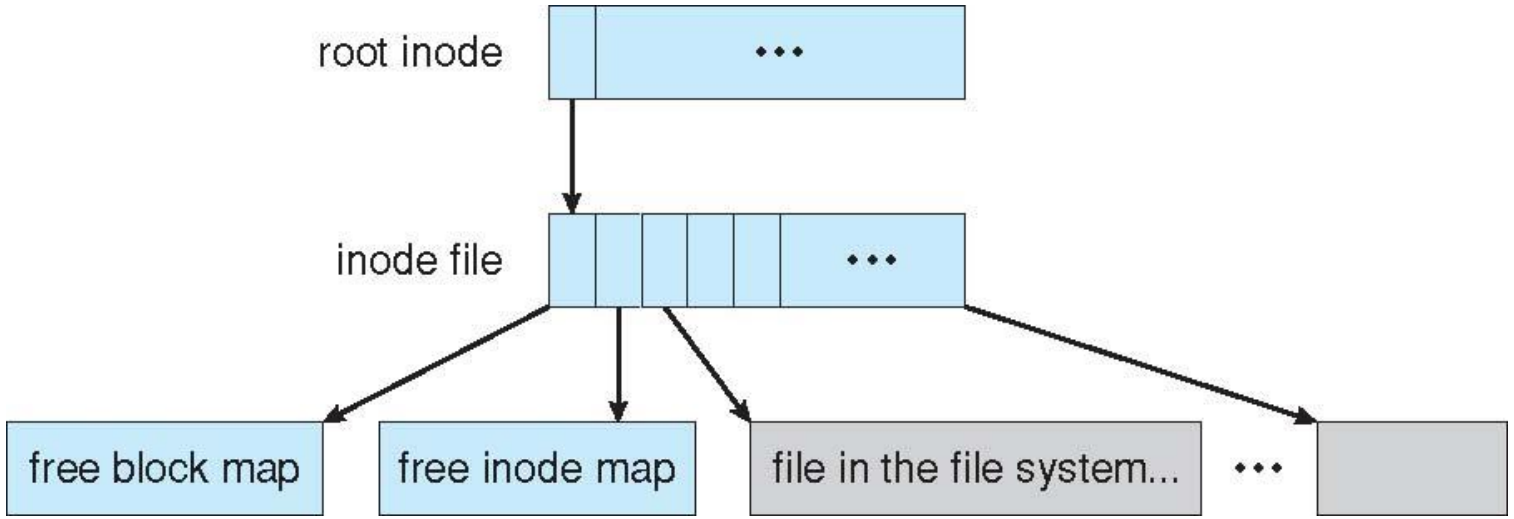
NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

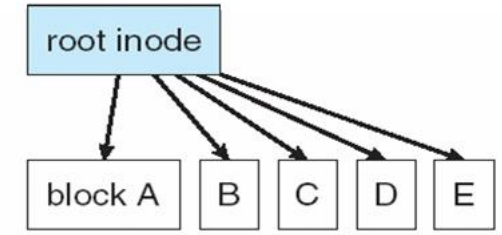
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

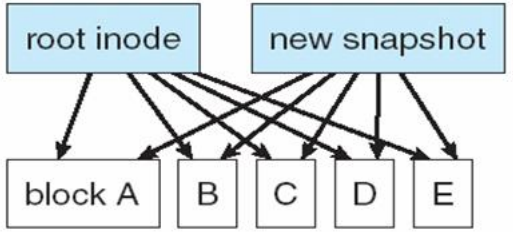
The WAFL File Layout



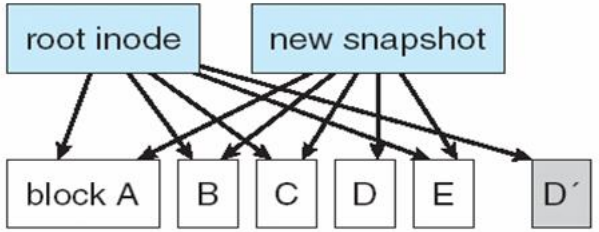
Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

Thanks

Politecnico di Torino

Department of Control and Computer Engineering

Questions?

sarah.azimi@polito.it