File Systems

Unix File System

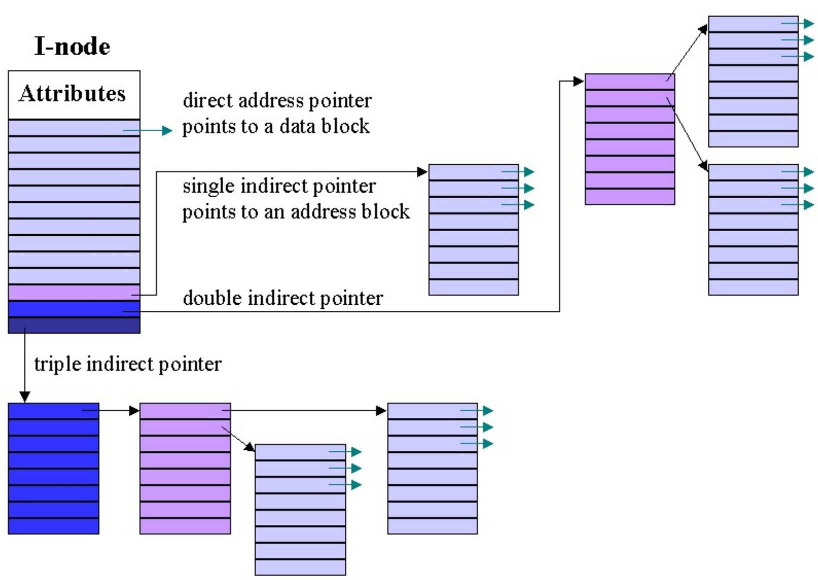
- File system structure
  - Super block: keep track of the file system parameters
    - Block size, # I-node blocks, # file blocks, etc.
  - I-node map, data block map
    - In the summary block
    - Bit maps specifying which blocks are actually occupied (or empty)
  - I-node blocks
    - All i-nodes are placed in this region (metadata blocks)
  - File blocks
    - The actual content of the file

Super Block	I-node Map	File Map	I-node Block	…	I-node Block	File Block	…	File Block

Unix File System: I-node

- I-node
  - The low level file descriptor + index
    - Like the physical name versus the logical path name
  - Each I-node contains
    - File name, file type, other attributes
      - Date modified, ownership, access privileges, etc.
    - Pointers to data blocks
      - Direct pointers: 10 pointers
      - Single indirect pointers: 1 pointer
      - Double indirect pointers : 1 pointer
      - Triple indirect pointers : 1 pointer
  - Kernel maintains I-nodes of all active files in memory

Unix File System: I-node



Unix File System: I-node

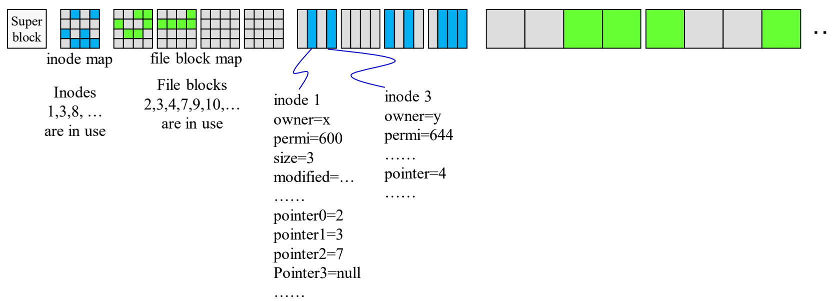
- How big a file can the I-node address?
    - 10 direct pointers, 1 single indirect pointer, 1 double indirect pointer, 1 triple indirect pointer
  - Assume: each file block is 4KB, each address is 4B
  - Direct pointers only
    - Total can point to 10 file blocks
    - Max file size: 10 * 4KB ⇒ 40KB
  - With single indirect pointer
    - One address block = 4KB = 1K addresses
    - Each address from the address block points to one block
    - Total can point to 1K file blocks ⇒ 1K * 4KB ⇒ 4MB
    - Max file size: 4MB + 40KB

Unix File System

- How big a file can the I-node address?
  - With double indirect pointer
    - One address block = 4KB = 1K addresses
    - 1K addresses, each point to an address block ⇒ 1M address
    - Total can point to 1M file blocks ⇒ 1M * 4KB ⇒ 4GB
    - Max file size: 4GB + 4MB + 40KB
  - With triple indirect pointer
    - Indirectly can point to 1G file blocks ⇒ 1G * 4KB ⇒ 4TB
    - Max file size: 4TB + 4GB + 4MB + 40KB
    - Can cover a huge file size
    - Suitable for different file sizes
      - Less access overhead for smaller files
      - Less overhead for accessing the initial blocks of files

Unix File System

❖ A file system example



- How to know where the i-node for a file is in the i-node block ⇒ Directory

Unix File System

❖ Directory

- Directory essentially stores (file name, i-node number)
  - Plus some duplicates of the information in i-node
  - Such as owner, access privilege
- i-node number is the index to the i-node block
  - Each i-node is of a fixed size
    - o Old: 128B, some new ones: 256B
  - Assume that each i-node is of size 128B, if i-node number for a file F is 10, then the i-node for the file can be found at: (i-node block base address) + 128 * 10
- Start from root directory
  - But where is the root inode number? It has no parent directory to record its inode number
  - Always fixed (in unix, inode number for root is always 2)

0: reserved for NULL value
1: pointers to bad blocks

Unix Fast File System

- Access efficiency issues
  - Problems with indexed allocation
    - File blocks may be scattered around
    - i-node may be far away from the file blocks
    - Access speed will be very slow even for sequential accesses
      - Sequential access is probably the most common form for file accesses
  - File allocation should keep disk access efficiency in mind
- ⇒ BSD Unix FFS
  - Increase file block size, make it independent of sector size
  - Cylinder groups

Unix Fast File System

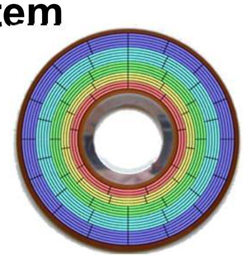
- Cylinder groups
  - Disk-aware file allocation solution provided in Unix FFS
  - Disk is divided into many cylinder groups, each includes several consecutive cylinders, generally several MB
    - Common settings: 16 ⇒ 64M
    - Each has a copy of the super block, its own inode map, data map, inode blocks and data blocks
  - Allocate a file in one group so that data blocks and inode blocks are all nearby

Super Block	I-node Map	File Map	I-node Block	…	I-node Block	File Block	…	File Block
Super Block	I-node Map	File Map	I-node Block	…	I-node Block	File Block	…	File Block

Unix Fast File System

- Cylinder groups
  - Locality for multiple files
    - User frequently access a few files together or skipping among them
    - Develop programs, accessing several source files
    - Put files in a directory together in one cylinder group
  - Policy for large files
    - A very large file may occupy all the blocks in a cylinder group
    - Lose the chance to consider locality of multiple files
    - Solution: put large files across multiple cylinder groups (sequential access will get quite a few blocks of the file and then jump to another)

Unix Fast File System

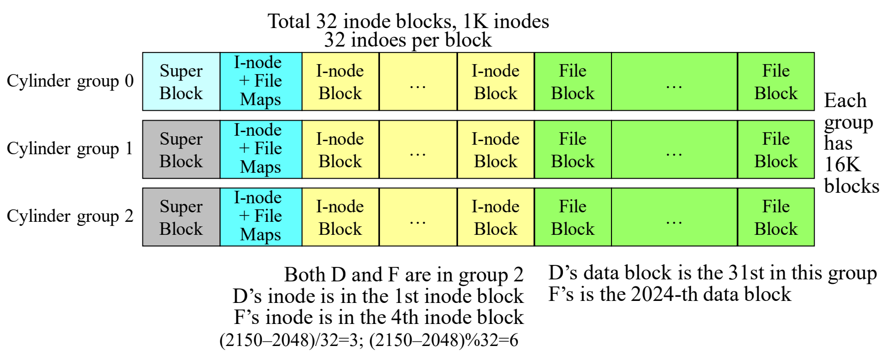


- Example for Unix FFS
  - Disk for the file system
    - 8K sectors per track, each sector is 512B
  - Each file block is of size 4KB
    - 1K file blocks per track
  - Each group is 16 tracks $\Rightarrow$ 16K blocks, total size 64MB
    - Each group intends to host 1K files (average file size: 64KB)
    - 1 super block, 1 map block
      - One map block can host both inode map and data block map
      - 16K blocks $\Rightarrow$ map = 16K bits = 2KB $\Rightarrow$ within one 4KB block
    - Each group needs 1K inodes $\Rightarrow$ 32 inode blocks $\Rightarrow$ 32 bit map
      - Commonly, each inode is 128B $\Rightarrow$ each file block can store 32 inodes
    - The remainder are data blocks, 16K–34 data blocks

Unix Fast File System

- ❖ Example
  - ➢ Read a file F in a directory D
    - ▪ D’s inum is 2050, data block index is 30
    - ▪ F’s inum is 2150, data block index is 2023

Where are D and F on disk?
D and F are in group 2
D’s inode: block 2, inode 2
D’s data: block (34+30)
F’s inode: block 5, inode 6
F’s data: block (34+2023)
Each track has 1K blocks



Unix Fast File System

- ❖ Example
  - ➢ Read a file F in a directory D
    - ▪ D’s inum is 2050, data block index is 30
    - ▪ F’s inum is 2150, data block index is 2023
      - – Both are single blocked
  - ➢ Read D’s data block
    - ▪ D’s inode address: block #2 ⇒ track #32, starting sector #16
    - ▪ D’s data address: block #64, track #32, starting sector #512
  - ➢ Read F’s data block
    - ▪ F’s inode address: block #5 ⇒ track #32, sector #40
      - – No need to read, once it is opened, the inode is cached in file table
    - ▪ F’s data address: block #2057 ⇒ track #34, sector #72
      - – F is the 2058-th block in group 2 = 3rd track, 10th block (block 9)

Where are D and F on disk?
D and F are in group 2
D’s inode: block 2, inode 2
D’s data: block (34+30)
F’s inode: block 5, inode 6
F’s data: block (34+2023)
Each track has 1K blocks

Unix Fast File System

- Example
  - Add a new file block in F
    - D's inum is 2050, data block index is 30
    - F's inum is 2150, first data block index is 2023, new one is 2123
  - Write the new data block for F
    - Block #(34+2123=2157) in cyc-group 2
    - Block #109 on the track #2 (3rd track)
    - At track #34, starting sector #872
      - Start at sector #872 (109*8)
  - Update F's inode block
      - From previous page: track #32, sector #40
  - Write to data block map
    - Block #1 in group 2 = track #32, sector #8

Where are D and F on disk?
D and F are in group 2
D's inode: block 2, inode 2
D's data: block (34+30)
F's inode: block 5, inode 6
F's data: block (34+2023)
Each track has 1K blocks

Update Atomicity

- What if system crashed during updating
  - Update to a disk sector is atomic, guaranteed by the disk, which stores sufficient power to ensure this
  - IEEE standards requires a system to define the atomic disk write size: PIPE_BUF
    - Defined in limits.h in Linux
    - Generally from 4KB to 64KB
    - File data block size is generally less than PIPE_BUF
    - Page size is also confined by PIPE_BUF to assure atomic swap space write

Update Atomicity

- What if system crashed during updating
  - Update to a block involves more than data block update
    - Need to update the data block, the inode, the maps
  - Just the data block is written to disk
    - inode that points to the block and bitmap shows it is not allocated
    - As if the write never occurred ⇒ Not a problem at all
  - Just updated inode, data block is not updated
    - If we trust the inode pointer, garbage data will be read
    - The bitmap shows that data block has not been allocated ⇒ The file system data structures is inconsistent
  - Just updated bitmap, not the rest
    - bitmap indicates that data block 5 is used, but no inode points to it
    - Result in a space leakage: block 5 can no longer be used

Update Atomicity

- What if system crashed during updating
  - Updated the maps and inode, but not the data
    - System is consistent, but data is garbage
  - Updated the inode and the data but not the maps
  - Updated the data and the maps but not the inode
    - Inconsistent file system state
  - Updated the directory, but the file info is not updated
    - Inconsistent file system state
  - Updated the file info, but directory info is not updated
    - Inconsistent file system state
- Any of the scenarios can happen
  - Disk scheduling ⇒ The order of the activities is unknown

Update Atomicity

- Cope with crash during updating
  - Solution 1: check the file system consistency
    - E.g., unix fsck, etc
    - Check the inodes in the directories to rebuild the bit maps
      - Trust inodes more than bit map
    - But cannot find some of the failure scenarios
      - Garbage data
        - Just updated inode, data block is not updated
        - Updated the maps and inode, but not the data
      - Data loss
        - Just the data block is written to disk

Update Atomicity

- Cope with crash during updating
  - Solution 2: Journaling
    - Write each transaction, including data, to a journal first, then perform the update
      - Include the begin transaction and end transaction marks to validate the transaction
    - But a journal may not be written in order either and the journal itself may be invalid
      - E.g., journal for the inodes and the maps are written, transaction begin/end are written, but not the data $\Rightarrow$ Data is garbage
      - Fix: next page
    - Fix: add an integrity coding in the transaction end block
      - Disk system actually guarantees atomic write to each block
      - End-transaction + checksum are small enough to fit one block

Update Atomicity

- Cope with crash during updating
  - Solution 2: Journaling
    - Where to write the journal
      - Has to be on disk, has to be in a known location
      - Choose to set aside a space after the super block
    - Journal space is limited
      - Treat journal space as a circular buffer
    - Need to assure that journal is always written before the actual transaction
      - Get confirmation from disk for journal writing, before performing the actual transaction $\Rightarrow$ Higher latency

Update Atomicity

- Cope with crash during updating
  - Journal example: For a transaction that create a new file with a new data block
    - First block of the journal contains
      - Begin transaction mark, transaction id, operation type
      - File name, file inum, file inode content
      - Directory name, directory file inum, directory file inode content
    - Second and third blocks include
      - Content of the new data block for the file
      - Content of the new data block for the directory, if needed
    - Fourth block includes
      - End transaction mark, transaction id, checksum of this journal
        - Transaction id should be there to identify the Begin/End pair
    - Journal generally is designed to write sequentially

In all designs, this journal data becomes the real data block
Otherwise, waste time and space

File Systems

- Issues for OS
  - Organize files
    - Directories structure
  - File types based on different accesses
    - Sequential, indexed sequential, indexed
  - File allocation (on disks)
  - Support various file operations
    - create, delete, open, close, append
    - read, write, seek
  - Access control

File Types based on Different Accesses

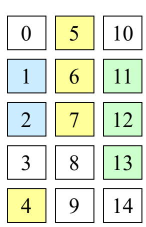
- Sequential File
  - a file has to be accessed sequentially
- Indexed Sequential File
  - Can have sequential and random accesses
  - Use index to locate a specific record in the file
    - Record size has to be fixed
    - System computes the offset from the index
    - E.g., record size = 16B, index = 5 (starting from 0)
    - $\Rightarrow$ offset = 5*16 = 40B
    - $\Rightarrow$ move the head to the correct offset

File Types based on Different Accesses

- Indexed File
  - A table is maintained to allow primary key and/or secondary key searches
    - The table is generally duplicated in memory
    - After locating the key, the record can be located
- Systems
    - All systems supports sequential accesses
    - Unix: support "lseek"
      - Similar to indexed sequential file, but user computes offset
    - VMS: support indexed sequential file
    - OS rarely supports indexed files
  - All these are the conceptual file types, not file allocation

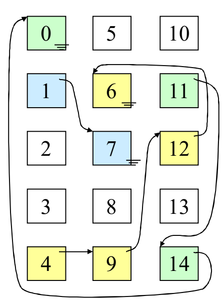
File Allocation

Contiguous Allocation



File id	start	len
File A	4	4
File B	1	2
File C	11	3

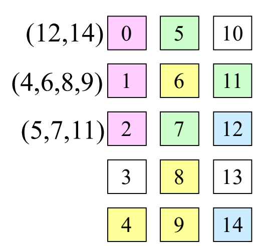
Chained Allocation



Start block

File A	4
File B	1
File C	11

Indexed Allocation



Index block

File A	1
File B	0
File C	2

File Allocation

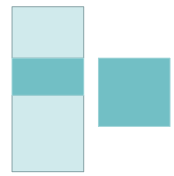
- Contiguous Allocation
  - Similar to dynamic allocation in main memory
    - Can use best-fit or other strategies
  - Just need to maintain file start location and length
  - Efficient file accesses, but disk fragmentation problem
- Chained (Linked-List) Allocation
  - Allocate any free blocks (similar to paging)
  - Need to keep pointers to the next block (chain all blocks)
  - No fragmentation problem
  - Very inefficient accesses
    - Require sequential accesses to file blocks

File Allocation

- Indexed Allocation
  - No fragmentation problem either
  - Need to keep index blocks for each file
  - The index block can be copied to memory
    - Can still have efficient access
  - More commonly used
    - Relatively efficient file accesses and efficient in space usage
- Disk access principle
  - Fastest is to access contiguous blocks

File Updates

- File allocation on updates
  - Contiguous allocation
    - Good for read, but very poor for updates
  - Indexed allocation
    - Choose a proper block size and update the entire block
    - But: small block size ⇒ High metadata and management costs + lose more on sequential writes in indexed allocation
    - Large block size ⇒ fragmentation + update issue
      - What if only updated a small portion of the block?
      - May also need to update inode, inode map, data map
  - ⇒ Log design
    - Updates are logged in memory



File Updates

- File allocation on updates
  - ⇒ Log design
    - Updates are logged in memory
    - Write the log out when it is full
    - Read: first find out whether updates exist in the log
    - Compaction
      - Merge the original and update log to generate new blocks
    - How to record the log
      - For many key-value storage system ⇒ Use key as the log index
      - For large text files ⇒ Use line as a unit and line number as the key
      - Other file types ⇒ Block as a unit and block # as the key

Log-Structured File System

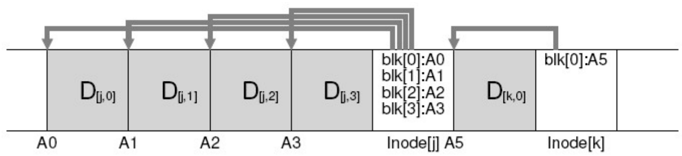
- Being disk-aware to an extreme
  - Best is to access contiguous blocks
  - For read: Try best to write consecutive blocks in a file to consecutive disk locations to improve sequential read
    - But sequential as a principle, not a requirement
    - Nothing else can be done for random read
  - For write: Write all updates to consecutive blocks on disk
    - No matter which blocks they are and which files they belong to
    - This is the basic concept of log structured file system (LFS)
    - But how to achieve this?
  - LFS is used in many file systems, especially the major cloud file systems (not Unix)

Log-Structured File System

- Write all updates to consecutive blocks
  - Buffer all the writes in the memory, called a segment, and write the entire segment out in one write request
    - In not, then each write will be issued separately, and reads from other processes may come in between the writes
      $\Rightarrow$ making writing contiguously impossible
    - The size of the segment is fixed, configurable for each file system, generally in MB
  - Wait till write the entire segment may not be feasible
      - If fsync or flush command is issued
      - If a sufficient time has passed
    - Write variant length of partial segment

Log-Structured File System

- Write all updates to consecutive blocks
  - But writing all the data together is not good enough
    - ⇒ Need to update inodes also
    - ⇒ Write data blocks with new inode blocks
      - But inode number will no longer work (no longer valid) example



Log-Structured File System

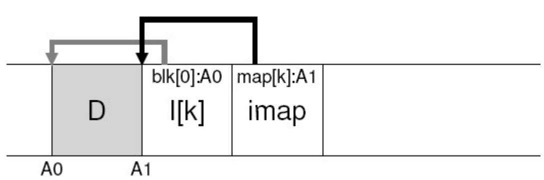
- Write all updates to consecutive blocks
  - If we change inode location (by writing an inode in a new location), the inode number in the directory needs to be changed also
  - ⇒ Update the directory and write the new directory in the log with the data
  - ⇒ Then, the parent directory needs to be updated, do the same as above?
  - ⇒ This will happen recursively ⇒ rewrite till the root
  - ⇒ Not good, can we do better?

Log-Structured File System

- Write all updates to consecutive blocks
  - How to locate the inode (inode number won't work now)
  - ⇒ Use indirect pointer, imap
    - Directory still keep the inode pointer
    - File system keeps an inode map (imap) to map an inode number to the most recent inode location
      - imap is like an array, can be addressed by inode number
      - Each imap entry only needs to be 4 bytes, to address the disk location for the inode
  - Where to store the imap?
    - Fixed region on disk (like original inode)
      - Need to write in the log area, and then switch to fixed imap area ⇒ defeat the purpose of LFS

Log-Structured File System

- Write all updates to consecutive blocks
  - Where to store the imap?
    - Fixed region on disk (like original inode)
      - Again defeat the purpose of LFS
    - Together with data and inode blocks, write in the log area



  - How to find imap in this case?
    - Finally need something in a fixed place to address other things
    - LFS choose the checkpoint region (CR) and delayed write

E.g., Write imap every 30 seconds (too long ⇒ data loss; too short ⇒ defeat the purpose)

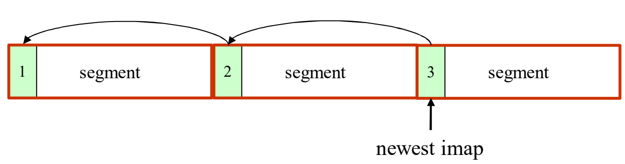
Log-Structured File System

- Write all updates to consecutive blocks
  - How to find imap?
    - LFS uses the “checkpoint region” (CR)
      - CR is in the beginning of each segment (fixed location)
  - ⇒ Write to imap breaks the sequential write
    - ⇒ Solution: delayed write (e.g., every 30 seconds)
      - Data will be lost if crash before it is written
    - ⇒ Solution: duplicate imap
      - Sequentially write imap out (gray block), but duplicate it in CR with delayed write (green block)
      - Enable recoverability, though may be slow, but better than data loss, and the inefficient recovery will be needed very infrequently

CR imap	Data	…	Data	inodes	…	inodes	imap	…

Log-Structured File System

- Write all updates to consecutive blocks
  - How to find imap?
  - imap on the log will not have all the inodes
    - Where a specific inode is? (original place or in some segment)
    - ⇒ Need to search through all segments, starting from the last one
    - E.g., inode with inum = x is in segment 3 (latest) and segment 1
  - ⇒ Solution: maintains the log tree (for imap) in memory
    - Still will result in slower read and faster write



Network File System

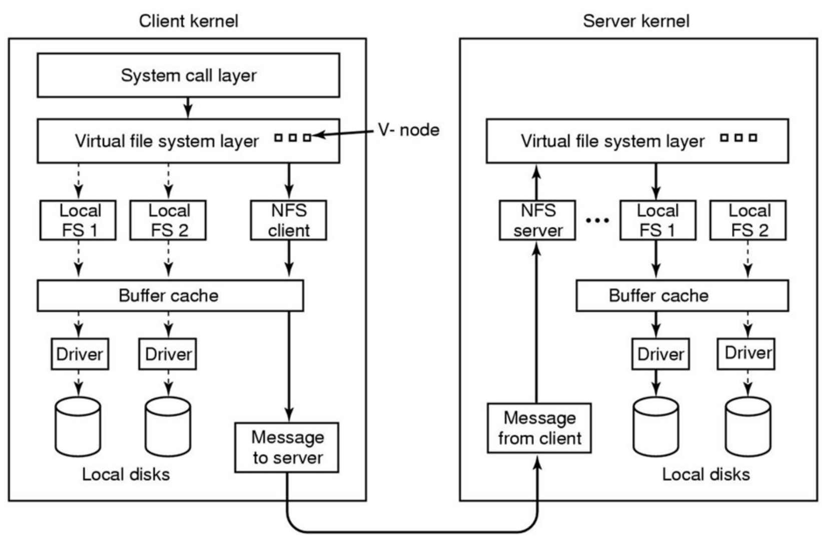
- Access files on the network
  - Remotely login to another computer to access its files
  - Mount a remote drive as though it is a local drive
    - Principle of network file system (NFS)
- NFS history
  - Originally designed by Sun in Solaris OS
  - Now it is a standard file system in all major Oss
    - Linux, Windows, …

Network File System

- Server exports directories
  - E.g., in Solaris, /etc/exports is the file specifying the list of directories to be exported
- Client mount the directory
  - Mount exported directory xxx to a local directory yyy
  - In Solaris, /etc/mnttab contains all directory mount info
  - E.g., mount cs1:/export/proj /usr/alice/proj/ nfs
  - Virtually, no difference between local and remote files
    - /usr/alice/proj is just like a local directory for Alice, used exactly the same way as a local directory
    - But the specific mount to a specific remote directory makes it non-transparent

e.g., cloud file systems are all DFS, not NFS

NFS Implementation

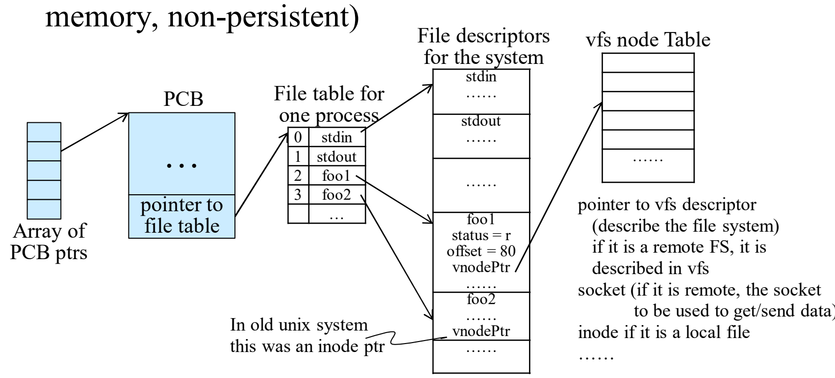


NFS Implementation

- VFS (virtual file system)
  - A file access always goes to VFS and subsequently be directed to local or NFS (or other) file systems
    - Other file systems can also be built under VFS
  - VFS Data structures
    - A v-node is used for each VFS file entry (virtual i-node)
    - It is just a pointer
      - For a local file, v-node points to an i-node
      - For a remote file, v-node points to an r-node

NFS Implementation

- VFS (virtual file system)
  - An intermediate data structure in the system to provide OS a uniform interface for different file systems (only stored in memory, non-persistent)



Readings

- Sections 12.2, 12.3, 12.4, 12.6, 12.7
- Section 16.2 last part for access control
- Papers
  - Unix fast file system
    - A fast file system for UNIX, ACM Transactions on Computer Systems, August 1984 (original)
    - ffsck: The Fast File System Checker, ACM Transactions on Storage, January 2014 (more informative)
  - Network file system
    - The Sun network filesystem: Design, implementation and experience, USENIX, 1986

Readings

- Papers
  - Journaling
    - Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea, C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, “IRON File Systems,” SOSP ’05, Brighton, England, October 2005
  - Log-structured file system
    - Mendel Rosenblumand John Ousterhout, “Design and Implementation of the Log-structured File System, ” SOSP ’91, Pacific Grove, CA, October 1991