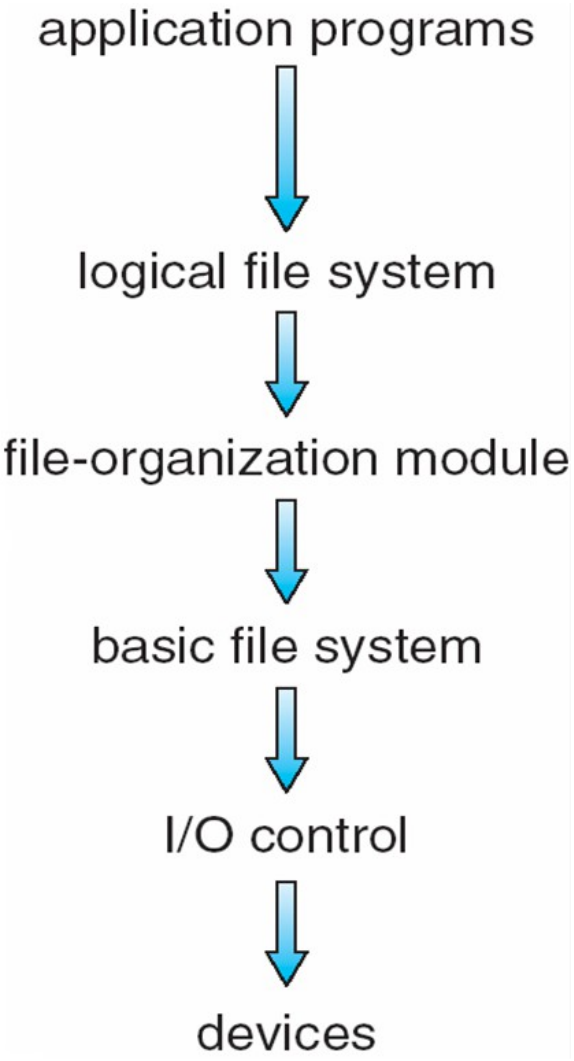
File System Implementation

ICS332

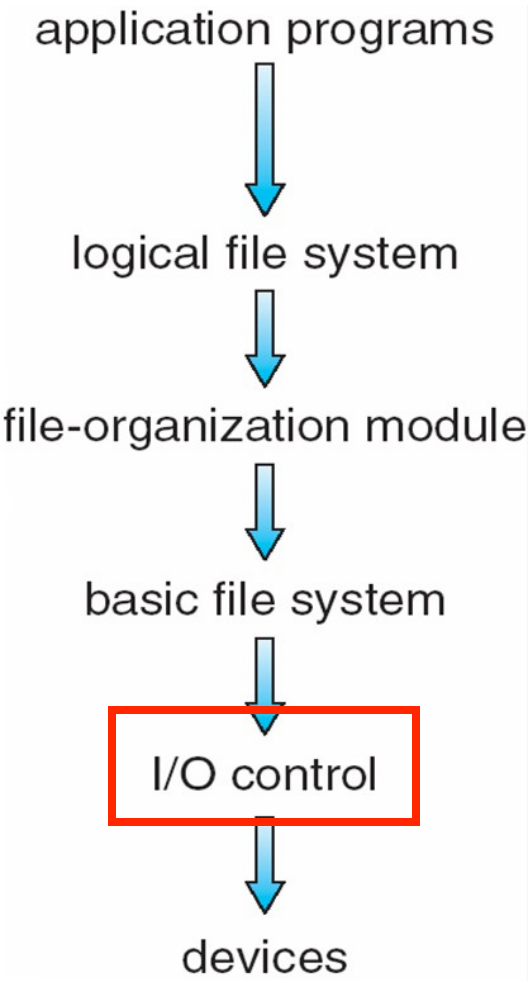
Operating Systems

File System Implementation

- The file system should provide an efficient implementation of the interface it defines
  - storing, locating, retrieving data
- The problem: define data structures and algorithms to map the logical FS onto the disk
  - some data structures live on disk
  - some data structures live (temporarily) in memory
- Typical layer organization:
  - Good for modularity and code re-use
  - Bad for overhead
- Some layers are hardware, some are software, some are a combination

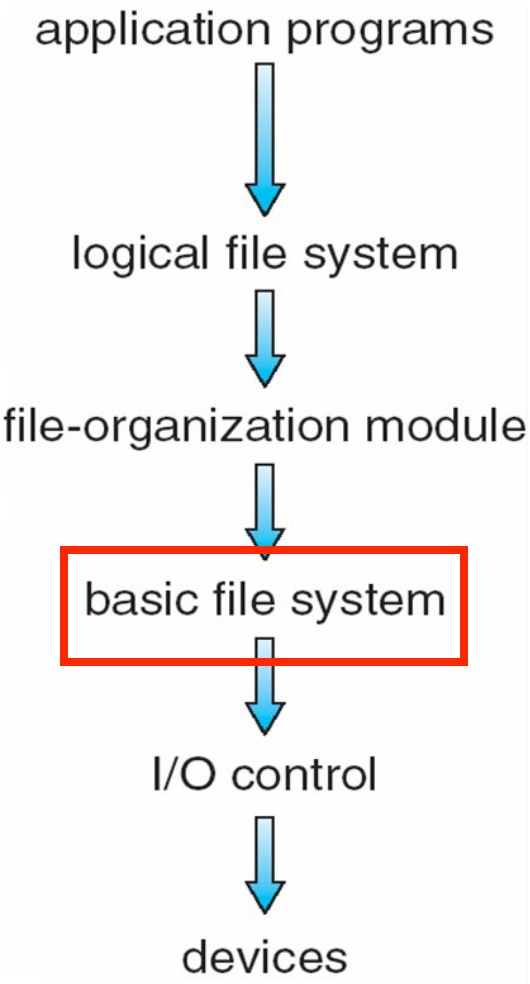


File System Implementation



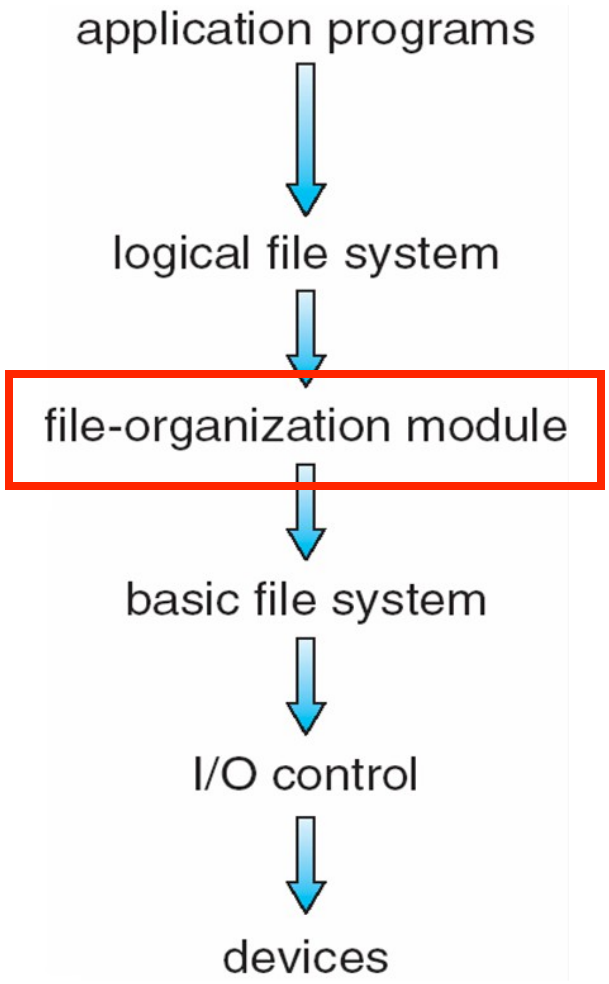
- I/O Control
  - Device drivers and interrupt handlers
  - Input from above:
    - Read **physical** block #43
    - Write **physical** block #124
  - Output below:
    - Writes into device controller's memory to enact disk reads and writes
    - React to relevant interrupts

File System Implementation



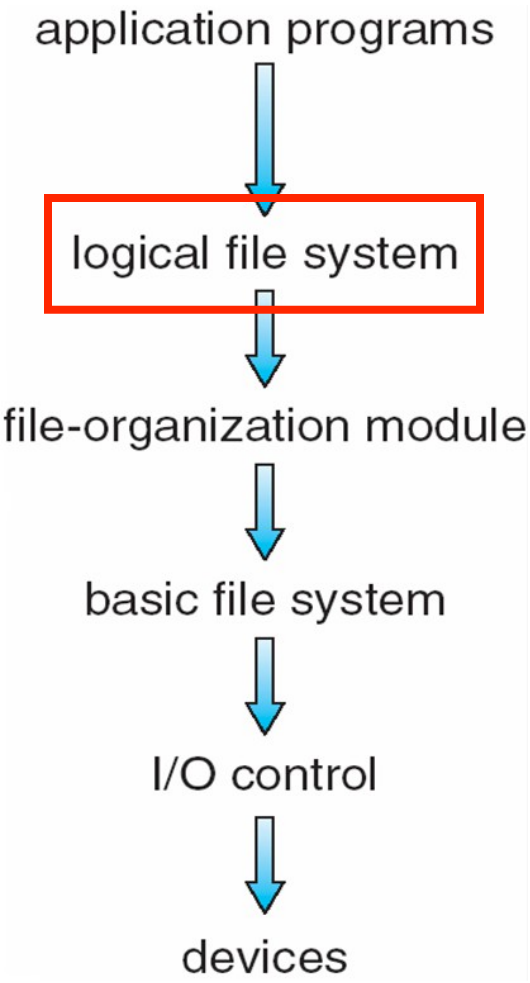
- Basic file system
  - Allocates/maintains various buffers that contain file-system, directory, and data blocks
  - These buffers are caches and are used for enhancing performance
  - Input from above:
    - Read **physical** block #43
    - Write **physical** block #421
  - Output below:
    - Read **physical** block #43
    - Write **physical** block #421

File System Implementation



- File-organization module
  - Knows about logical file blocks (from 0 to N) and corresponding physical file blocks: it performs translation
  - It also manages free space
  - Input from above:
    - Read **logical** block 3
    - Write **logical** block 17
  - Output below:
    - Read **physical** block 43
    - Write **physical** block 421

File System Implementation



- Logical file system
  - Keep all the meta-data necessary for the file system
    - i.e., everything but file content
  - It stores the directory structure
  - It stores a data structure that stores the file description (File Control Block - FCB)
    - Name, ownership, permissions
    - Reference count, time stamps, pointers to other FBCs
    - Pointers to data blocks on disk
  - Input from above:
    - Open/Read/Write filepath
  - Output to below:
    - Read/Write logical blocks

File Systems

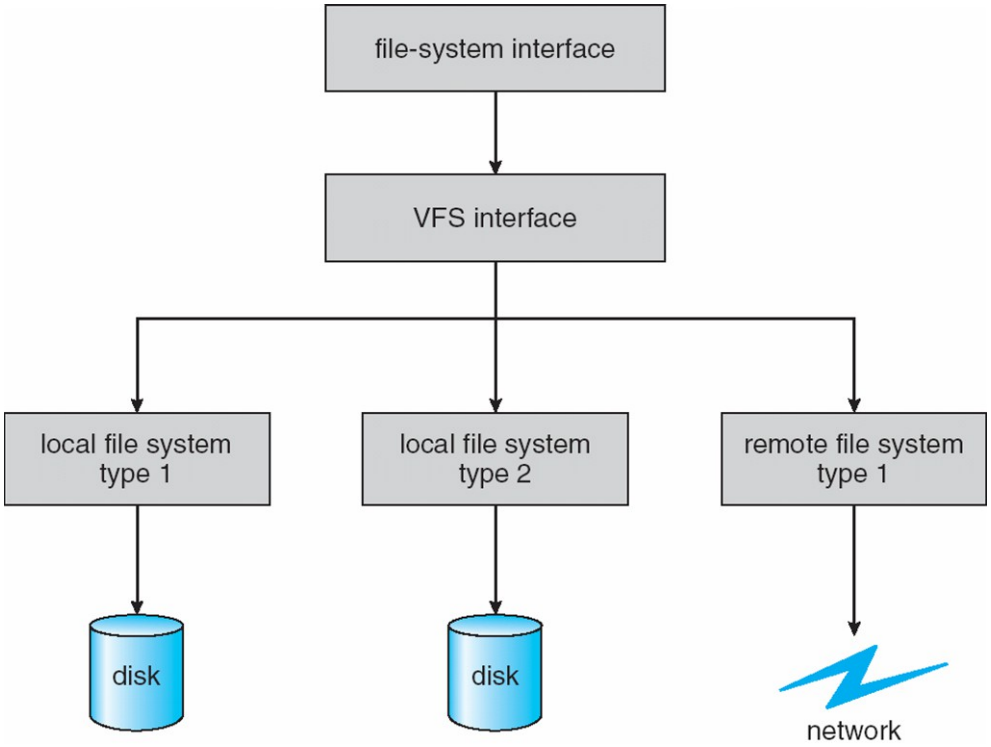
- Most OSes support many file systems
  - e.g., the ISO 9660 file system standard for CD-ROM
- UNIX:
  - UFS (UNIX FS), based on BFFS (Berkeley Fast FS)
- Windows:
  - FAT, FAT32, and NTFS
- Basic Linux supports 40+ file systems
  - Standard: ext2 and ext3 (Extended FS)
- An active area of research and development
  - Distributed File Systems
    - Not new, but still a lot of activity
  - High-Performance File Systems
  - The Google File System

File System Data Structures

- The file system comprises data structures
- **On-disk structures:**
  - An optional boot control block
    - First block of a volume that stores an OS
    - boot block in UFS, partition boot sector in NTFS
  - A volume control block
    - Contains the number of blocks in the volume, block size, free-block count, free-block pointers, free-FCB count, FCB-pointers
    - superblock in UFS, master file table in NTFS
  - A directory
    - File names associated with an ID, FCB pointers
  - A per-file FCB
    - In NTFS, the FCB is a row in a relational database
- **In-memory structures:**
  - A mount table with one entry per mounted volume
  - A directory cache for fast path translation (performance)
  - A global open-file table
  - A per-process open-file table
  - Various buffers holding disk blocks “in transit” (performance)

Virtual File System

- You'll hear of VFS (Virtual File System)
- This is simply about software engineering (modularity and code-reuse)
  - To support multiple types of FS, the OS expects a specific interface, the VFS
  - Each FS implementation must expose the VFS interface



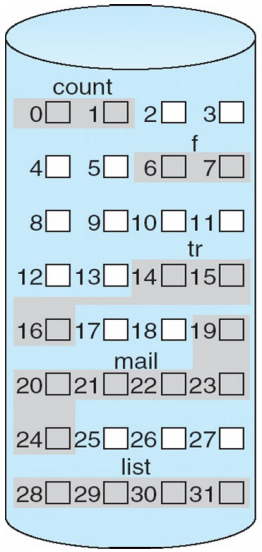
Directory Implementation

- Linear List
  - Simply maintain an on-disk doubly-linked list of names and pointers to FCB structures
  - The list can be kept sorted according to file name
  - Upon deletion of a file, the corresponding entry can be recycled (marked unused or moved to a list of free entries)
  - Problem: linear search is slow
- Hash Table
  - A bit more complex to maintain
  - Faster searches

Allocation Methods

- Question: How do we allocate disk blocks to files?

- The simplest: Contiguous Allocation
  - Each file is in a set of contiguous blocks
    - Good because sequential access causes little disk head movement, and thus short seek times
  - The directory keeps track of each file as the address of its first block and of its length in blocks



directory

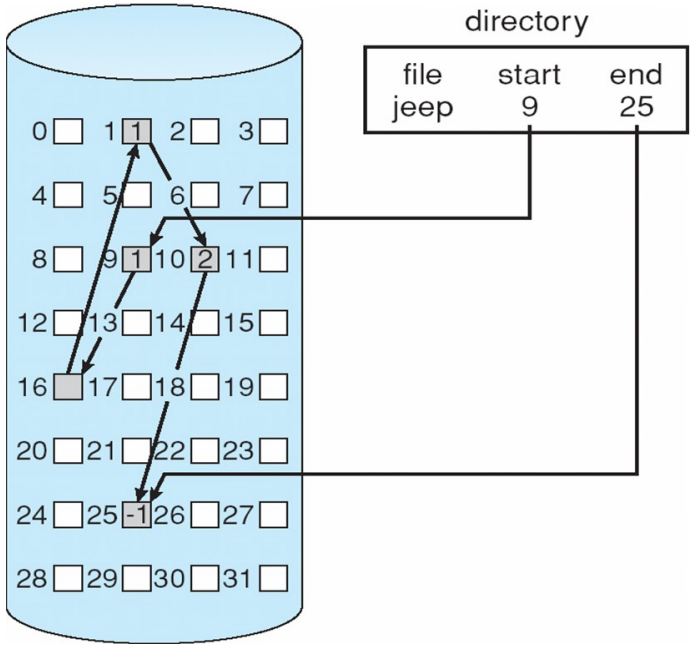
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation Problems

- Can be difficult to find free space
  - Best Fit, First Fit, etc.
- External fragmentation
  - With a big disk, perhaps we don't care
  - Compaction/defrag
    - Expensive but doable
- Difficult to have files grow
  - Copies to bigger holes under the hood?
    - High overhead
  - Ask users to specify maximum file sizes?
    - Inconvenient
    - High internal fragmentation
  - Create a linked list of file chunks
    - Called an extent

Linked Allocation

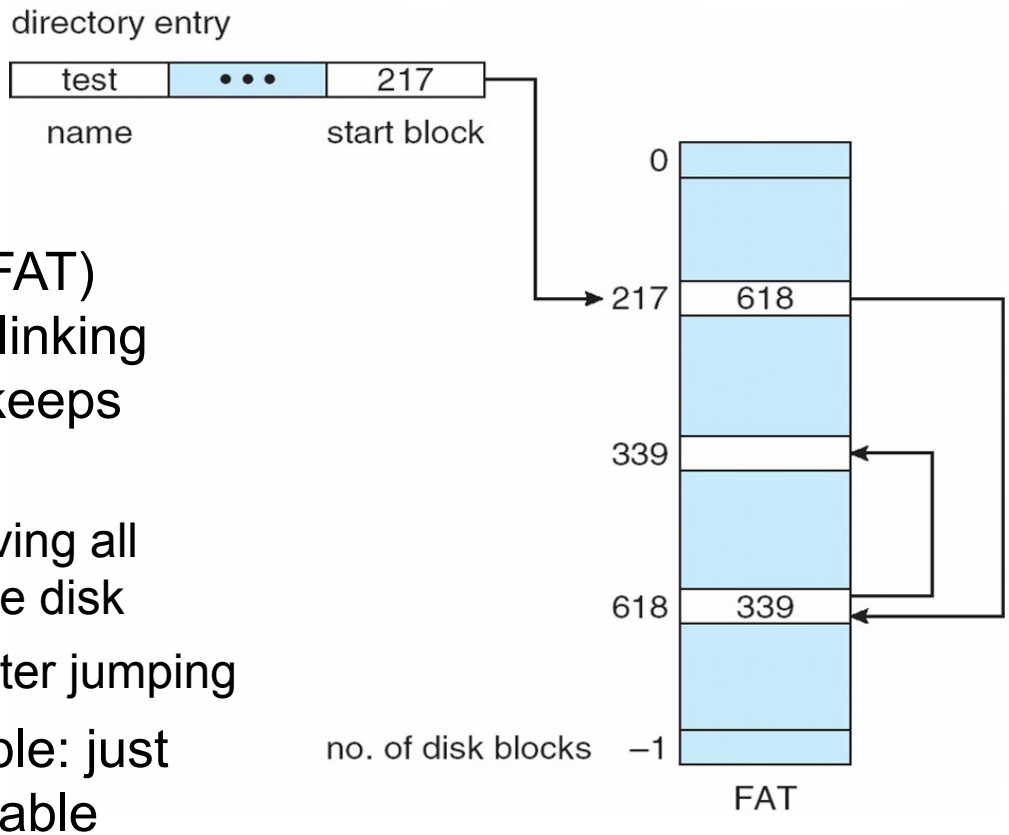
- A file is a linked-list of disk blocks
  - Blocks can be anywhere
- The directory points to the first and last block of the files
  - Pointer between internal blocks are kept on disk and “hidden”
- Solves all the problems of contiguous allocation
  - no fragmentation
  - files can grow



Linked Allocation Problems

- Great for sequential access but not so much for direct access
  - A direct access requires quite a bit of pointer jumping
    - meaning disk seeks (remember: data structure is on disk!)
- Wastes space
  - Say each pointer is 5 bytes, and each block is 512 bytes, then 0.78% of the disk stores pointers instead of data
  - Easy to solve by coalescing blocks together, i.e., allowing for bigger blocks
    - But at the cost of larger internal fragmentation
- Poor reliability
  - If a pointer is lost or damaged, then the file is unrecoverable
  - Can be fixed with a double-linked list, but increases overhead

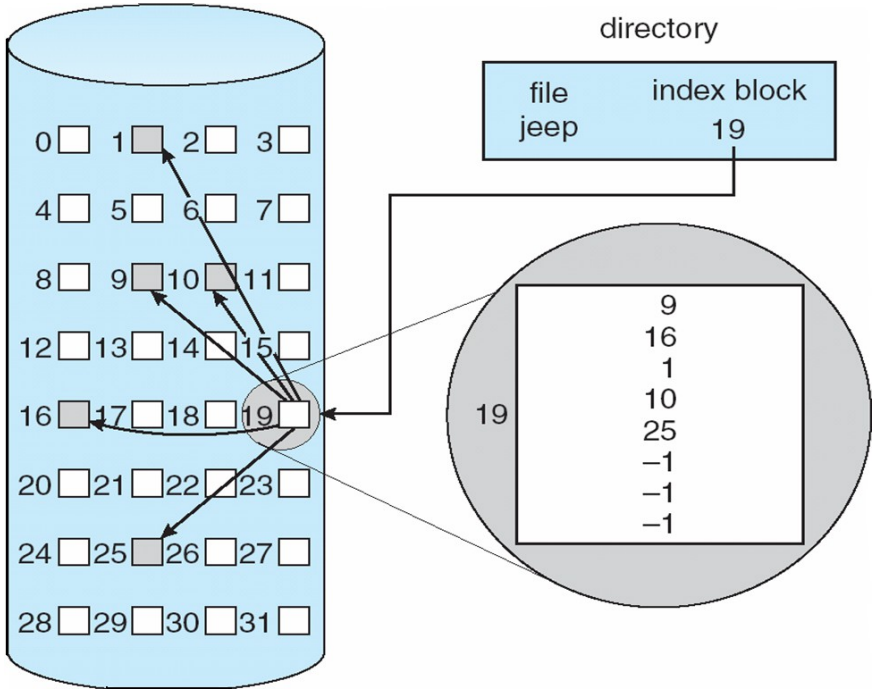
The FAT System



- The File-Allocation Table (FAT) scheme implements block linking with a separate table that keeps track of all links
  - Solves the problem of having all pointers scattered over the disk
  - Hence much quicker pointer jumping
- Finding a free block is simple: just find the first 0 entry in the table
- The FAT can be cached in memory to avoid disk seeks altogether

Indexed Allocation

- All block pointers are brought to a single location: the index block
- There is one index block per file
  - The i-th entry points to the i-th block
- The directory contains the address of the index block
- Very similar to paging, and same advantage: easy direct access
- Same advantage, but same problem: how big is the index block (page table)?
  - Not good to use all our space for storing index information!
  - Especially if many entries are nil
  - What if the index is bigger than a block? (remember page table pages)
- We had one page table per process, and now one index block per file!!
- There are several solutions

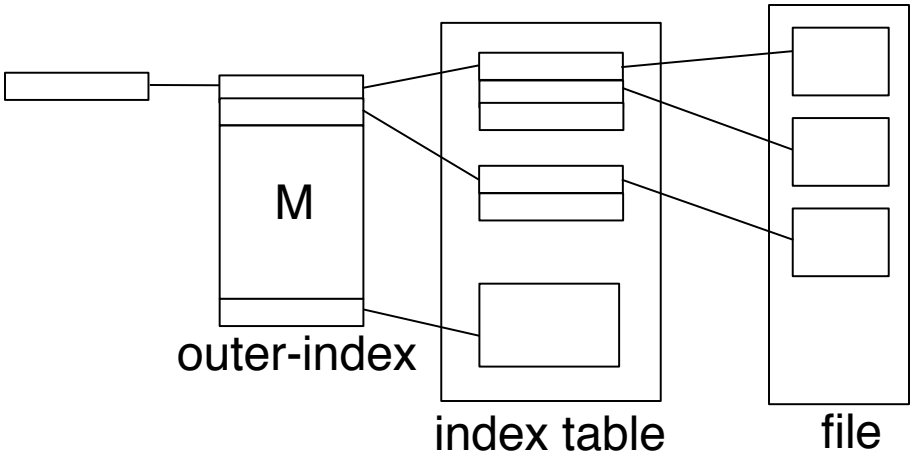


Indexed Allocation

- Linked index:
  - To allow for an index block to span multiple disk blocks, we just create a linked list of disk blocks that contain pieces of the full index
    - e.g., the last word in the first disk block of the index block is the address of the disk block that contains the next piece of the index block (easy, right?)
  - This adds complexity, but can accommodate any file size
  - Remember that disk space is not as costly as RAM space, and that the disk is very slow
  - Therefore, trading off space for performance and allowing for large indices is likely a much better trade off than it would be for RAM

Indexed Allocation

- Multilevel index:
  - Just like a hierarchical page table
  - If we have 512-byte blocks, and 4-byte pointers, then we could store 128 entries in a block
  - A 3-level scheme can then allow for 1GB files
  - A 4-level scheme can then allow for 128GB files

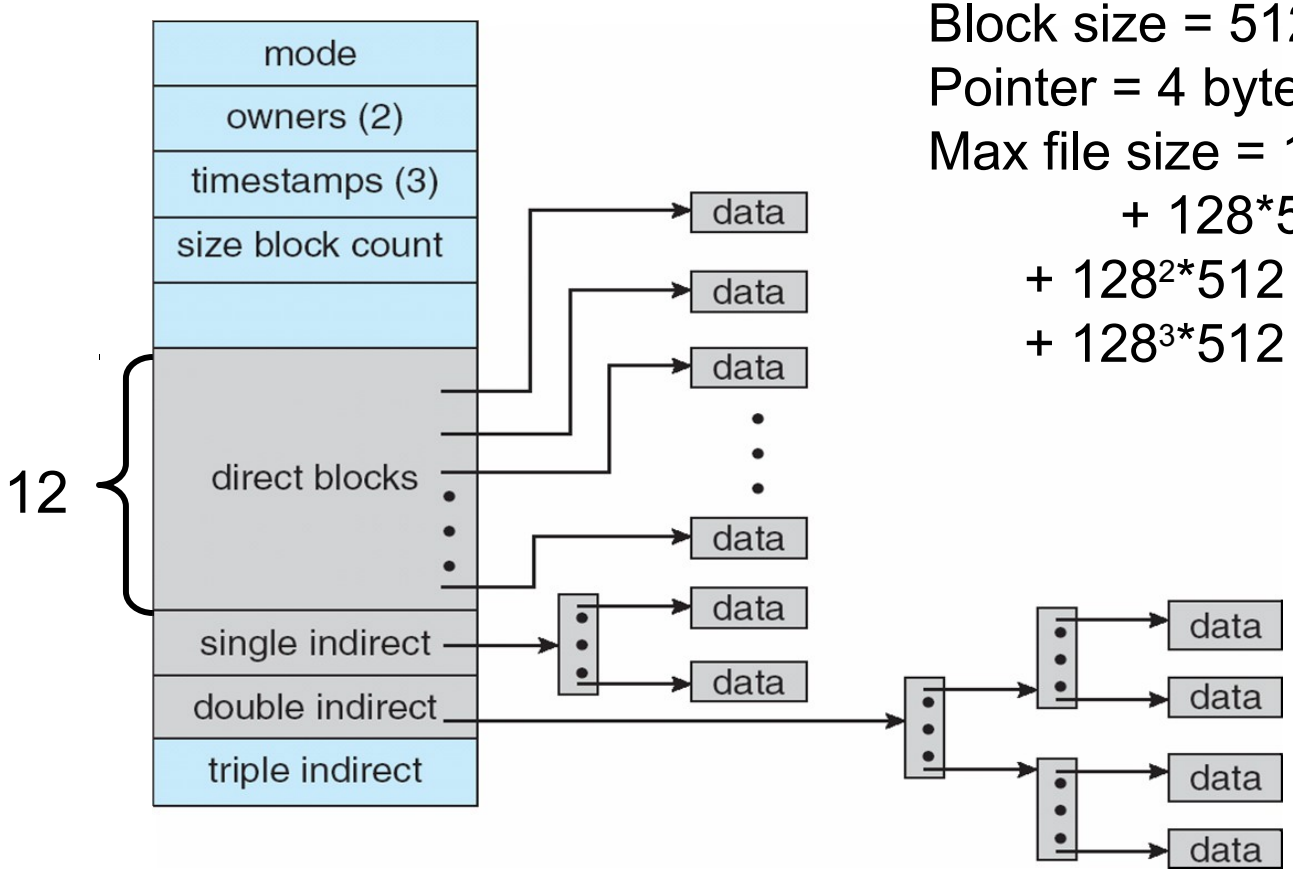


Indexed Allocation

- Combined Index:
  - For a small file it seems a waste to keep a large index
  - For a medium-sized file it seems a waste to keep multi-level indices
  - How about keeping all options open:
    - A few pointers to actual disk blocks
    - A pointer to a single-level index
    - A pointer to a two-level index
    - A pointer to a three-level index
  - That way small files don't even use an index

Indexed Allocation

- The UNIX FCB: the inode



Block size = 512 bytes
Pointer = 4 bytes
Max file size = 12*512
+ 128*512
+ 128^2*512
+ 128^3*512 bytes

In-Class Exercise

- Disk blocks are 8KiB, a block pointer is 4 bytes
- What is the maximum file size with the i-node structure?

  (give answer as a sum of terms)

In-Class Exercise

- Disk blocks are 8KiB, a block pointer is 4 bytes
- What is the maximum file size with the i-node structure?

- Direct indirect: 12 * 8KiB
- Single indirect: (8KiB / 4) * 8KiB
- Double indirect: (8KiB / 4) * (8KiB / 4) * 8KiB

- Total: $12 * 2^{13} + 2^{11} * 2^{13} + 2^{11} * 2^{11} * 2^{13}$

(which is about 32GiB)

Inodes and Directories

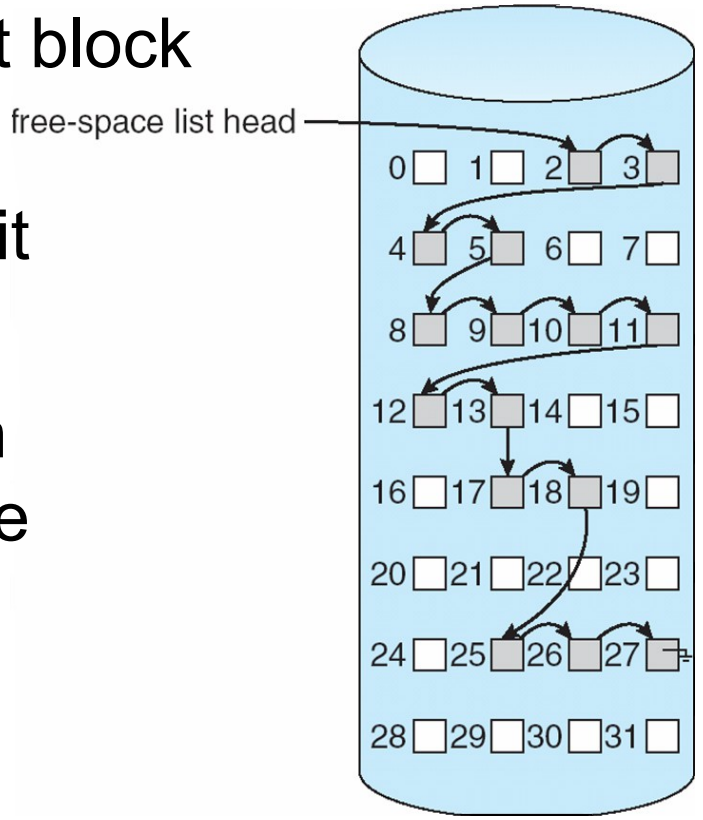
- An inode can describe a file or a directory
  - A bit says whether it's one or the other
- An inode for a directory also points to data blocks
- But these data blocks happen to contain <name, pointer to inode> pairs
- These data blocks are searched for names when doing pathname resolution
- The system keeps an in-memory cache of recent pathname resolutions

Free Space Management

- Question: How do we keep track of free blocks?
- Simple option: Bitmap
  - Keep an array of bits, one bit per disk block
  - 1 means free, 0 means not free
  - Good:
    - Simple
    - Easy to find a free block (we love bitwise instructions)
      - e.g., find the first non-zero word
  - Bad:
    - The bitmap can get huge
    - So it may not be fully cachable in memory
      - At this point the number of times we said “but we could perhaps cache it in memory” should bring home the point that RAM space is really a premium
  - This is what NTFS does

Free Space Management

- Another option: Linked List
  - Maintain a chain of free blocks, keeping a pointer to the first block
- Traversing the list could take time, but we rarely need to do it
- Remember that FAT deals with free blocks in the data structure that keeps the “linked-list” of non-free blocks



Free Space Management

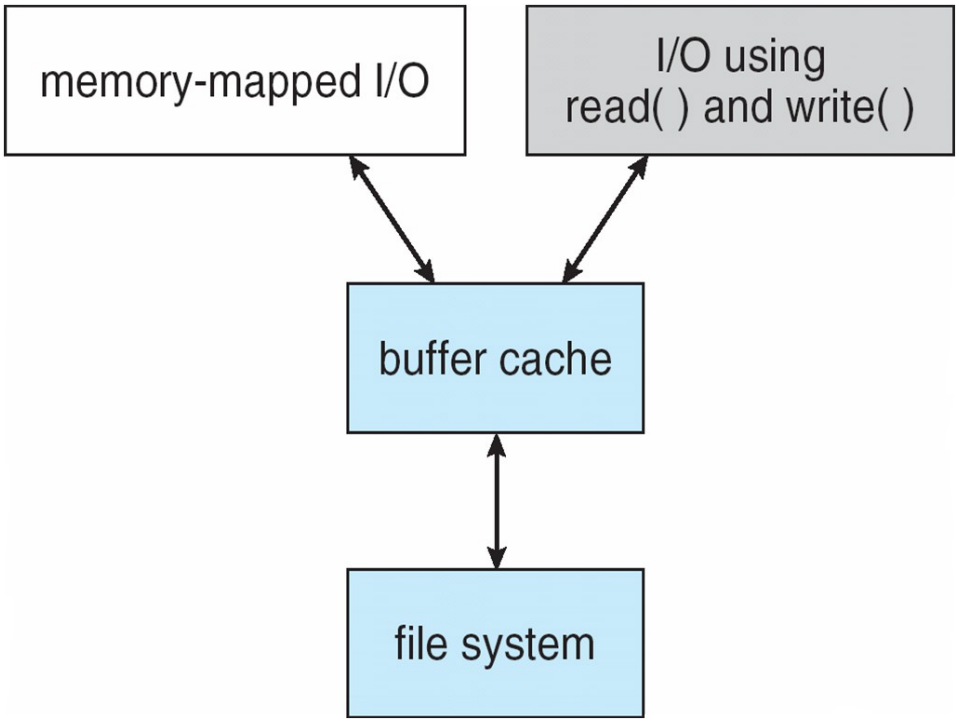
- Another option: Counting
  - Simply keep the address of a free block and the number of free blocks immediately after it
  - Saves space
    - Entries are longer
    - But we have fewer of them
  - These entries can be stored in an efficient data structure so that chunks of contiguous free space can be identified
    - Although we may do non-contiguous space allocation, it's always better to have disk spatial locality for performance

Efficiency and Performance

- There are many efficiency and performance issues for file systems (Section 11.6)
- Each aspect of the design impacts performance, hence many clever implementation tricks
- One well-known example: inode allocation
  - inodes are pre-allocated
    - When creating a new file, fields can just be filled in
  - inodes are spread all over the disk
    - So that a file's data can be close to its inode, if at all possible (minimizing seeks)

Efficiency and Performance

- Caching of disk blocks to take advantage of temporal locality



- Asynchronous writes whenever possible
- LRU
- Free-behind
- Read-ahead

- Competes with virtual memory for disk space!

Consistency Checking

- The File System shouldn't lose data or become inconsistent
  - It's a fragile affair, with data structure pointers all over the place, with parts of it cached in memory
- An abrupt shutdown can leave an inconsistent state
  - The system was in the middle of updating some pointers
  - Part of the cached metadata was never written back to disk
- Consistency can be checked by scanning all the metadata
  - Takes a long time, occurs upon reboot if necessary
  - A "necessary" bit is kept up-to-date by the system
- Unix: fsck, Windows: chkdsk
- Bottom line: We allow the system to be corrupted, and we later attempt repair

Journaling

- Problems with consistency checking:
  - Some data structure damaged may not be repairable
  - Human intervention is needed to repair the data structure
  - Checking a large file system takes forever
- Other option: Log-based transaction-oriented FS (Journaling)
- Log-based recovery:
  - Whenever the file system metadata needs to be modified, the sequence of actions to perform is written to a circular log and all actions are marked as “pending”
  - Then the system proceeds with the actions asynchronously
    - Marking them as completed along the way
  - Once all actions in a transaction are completed, the transaction is “committed”
  - If the system crashes, we know all the pending actions in all non-committed transactions, so we can undo all committed actions
    - And there are not too many of them
  - Writing to the log is overhead, but it’s sequential writing to the log file

Conclusion

- File Systems can be seen as part of or outside the OS
- File Systems are a complex and active topic
  - File System research papers get published all the time
  - Tons of File System development in R&D
  - A lot of discussion of what a file system really is
    - Can we weaken the semantic and make it easier to implement?
  - Distributed File Systems
    - e.g., file systems over p2p networks?