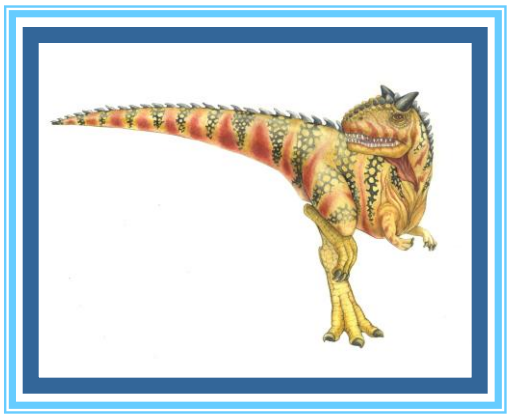
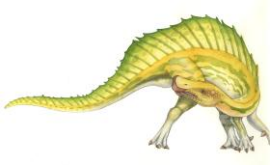
Chapter 14: File System Implementation





Outline

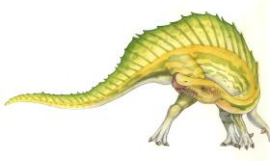
- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Example: WAFL File System





Objectives

- Describe the details of implementing local file systems and directory 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 a concrete example

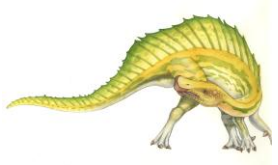


Troy University Resources



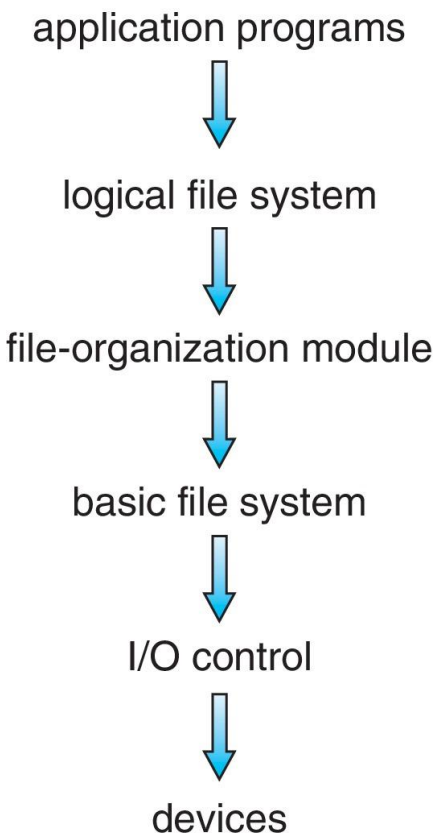
File-System Structure

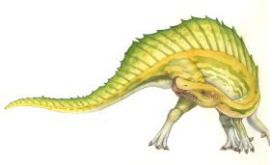
- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers use **blocks** of **sectors** (commonly 512 or 4096 bytes per sector)
- **File control block** (**FCB**) – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers





Layered File System

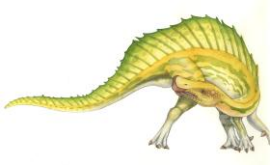






File System Layers

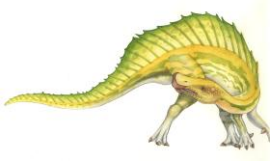
- **Device drivers** manage I/O devices at the I/O control layer
  - Given commands like
    - read drive1, cylinder 72, track 2, sector 10, into memory location 1060
  - Outputs low-level hardware specific commands to hardware controller
- **Basic file system** given command like "retrieve block 123" translates to device driver
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
- Manages free space, disk allocation





File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer

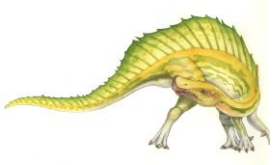


Troy University Resources



File System Layers (Cont.)

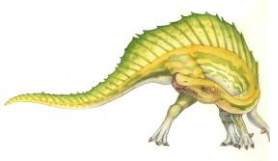
- Many file systems, sometimes many within an operating system
  - Each with its own format:
  - CD-ROM is ISO 9660;
  - Unix has **UFS**, FFS;
  - Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray,
  - Linux has more than 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - Other designs include ZFS, GFS and Oracle ASM. FUSE supports user-space file systems





File-System Operations

- 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
  - Needed if volume contains OS, usually first block of volume
- **Volume control block** (**superblock**) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table





File Control Block (FCB)

- OS maintains **FCB** per file, which contains many details about the file
  - Typically, inode number, permissions, size, dates
  - Example

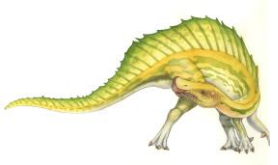
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks





In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info

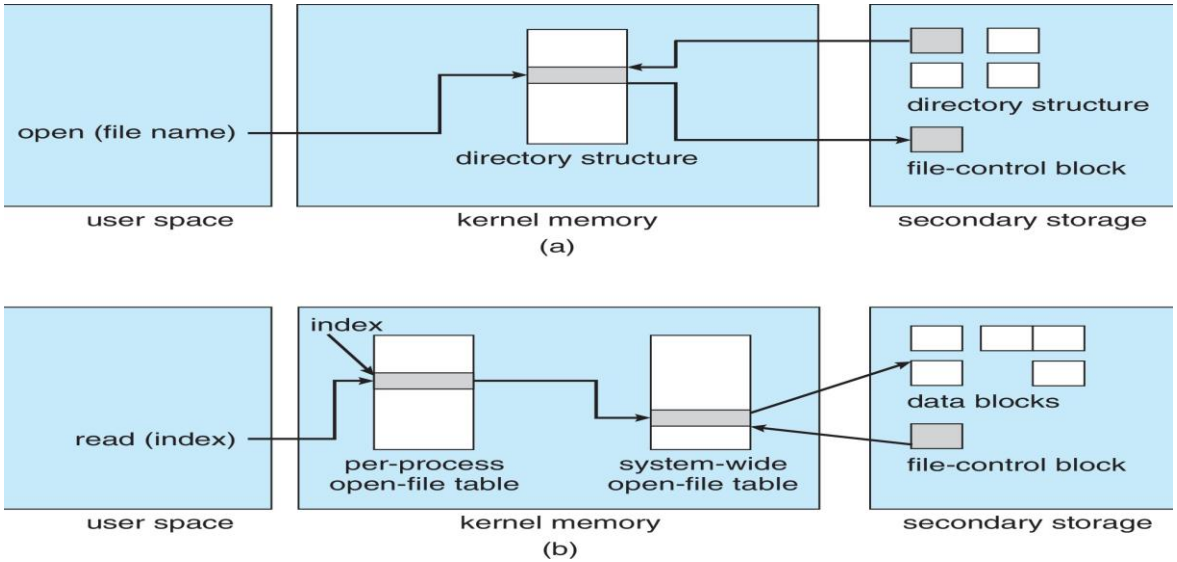


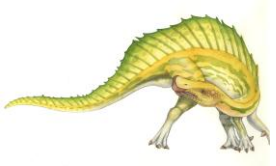
Troy University Resources



In-Memory File System Structures (Cont.)

- Part (a) of the figure shows opening a file
- Part (b) of the figure shows reading a file







Directory Implementation

- **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

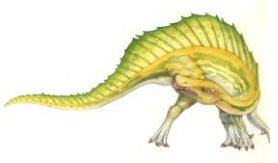


Troy University Resources



Allocation Method

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - Indexed (FAT is a variant of linked allocation)





Contiguous Allocation Method

- An allocation method refers to how disk blocks are allocated for files:
- 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 on the disk for a file,
    - Knowing file size,
    - External fragmentation, need for **compaction off-line** (**downtime**) or **on-line**

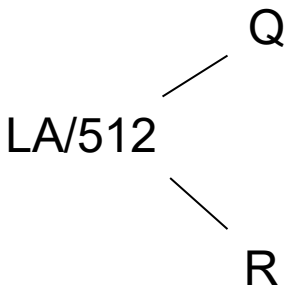


Troy University Resources

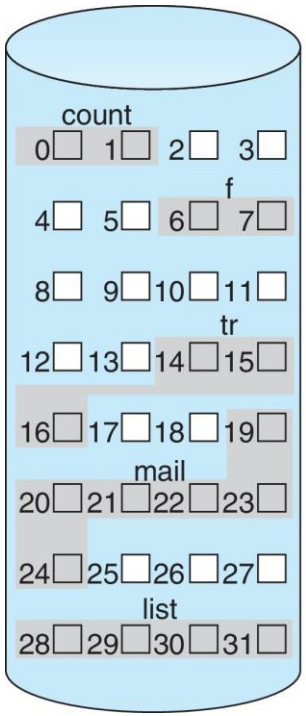


Contiguous Allocation (Cont.)

- Mapping from logical to physical (block size =512 bytes)



- Block to be accessed = starting address + Q
- 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) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An **extent** is a contiguous run of disk blocks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

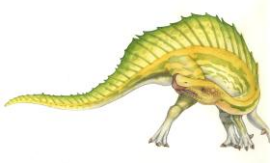


Troy University Resources



Linked Allocation

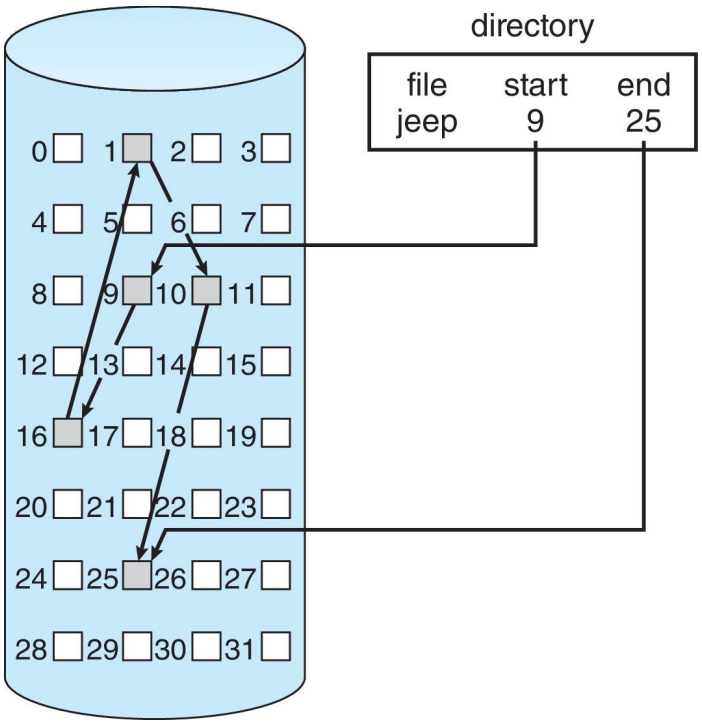
- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks





Linked Allocation Example

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



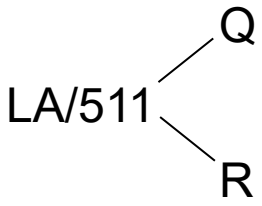


Troy University Resources



Linked Allocation (Cont.)

- Mapping (512-word blocks, one-word link, LA measured in words)



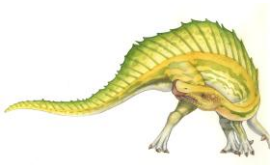
- Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file.
- Displacement into block = R + 1





FAT Allocation Method

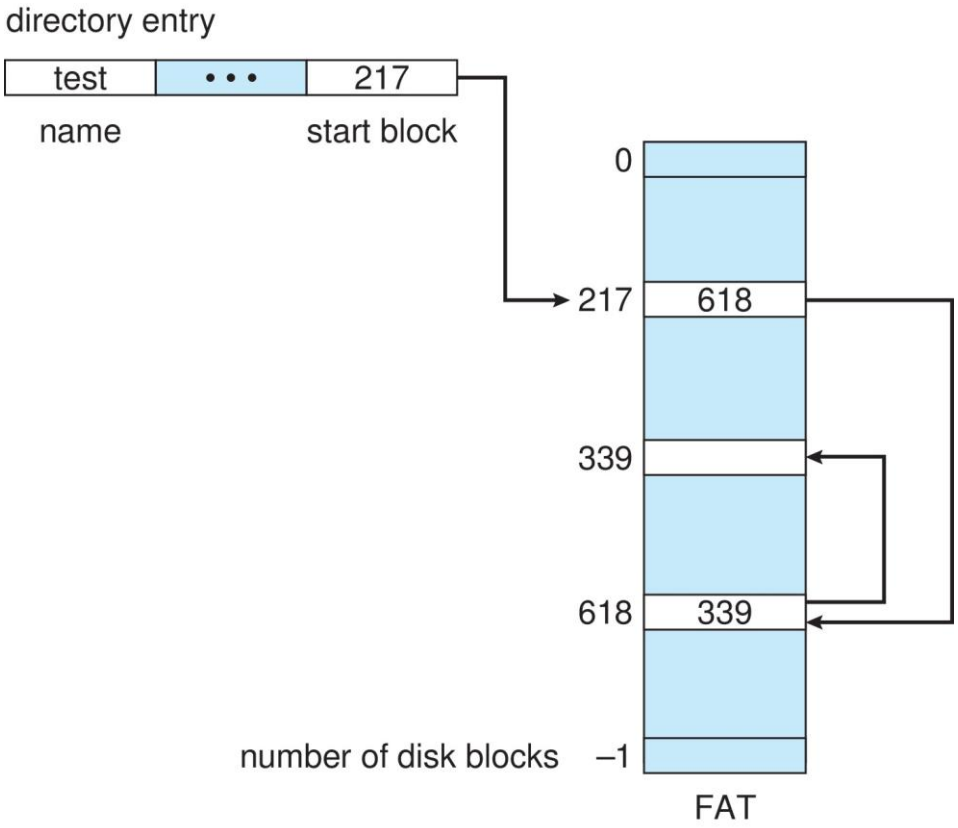
- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

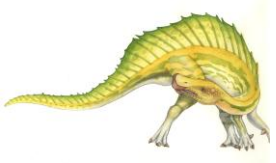


Troy University Resources



File-Allocation Table

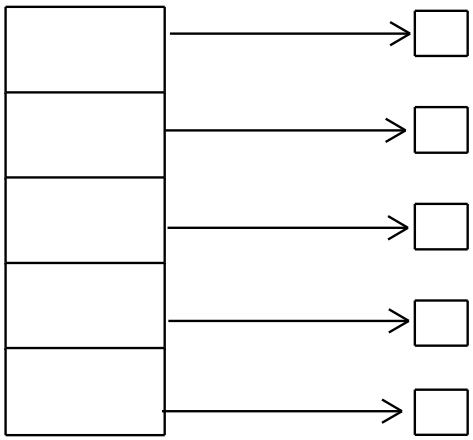


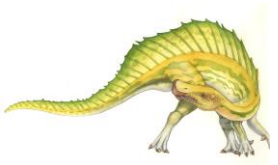




Indexed Allocation Method

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

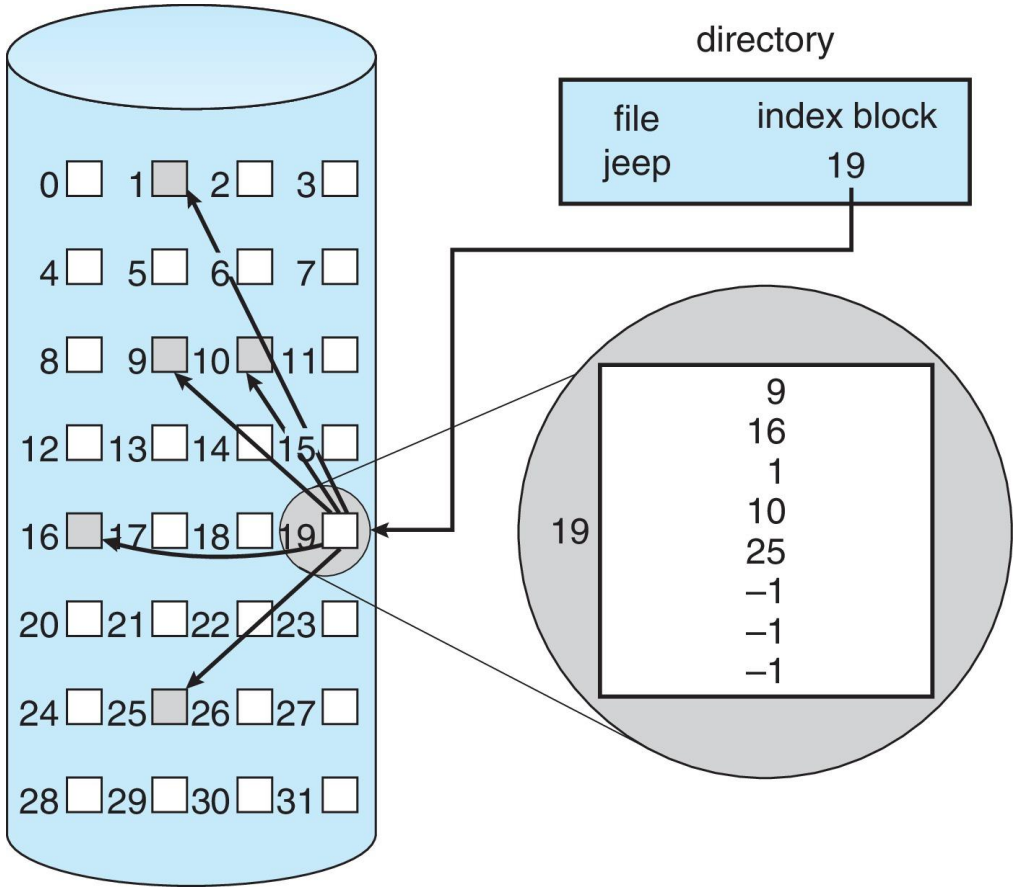




Troy University Resources



Example of Indexed Allocation

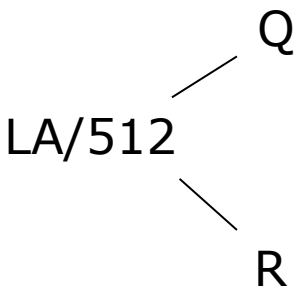






Indexed Allocation – Small Files

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Toy model: 512-word blocks and one-word block pointers. One 512-entry index addresses 256K words. LA is a word offset



- Calculation:
  - Q = displacement into index table
  - R = displacement into block

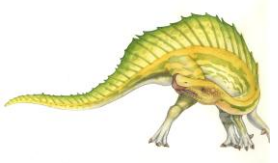


Troy University Resources



Indexed Allocation – Large Files

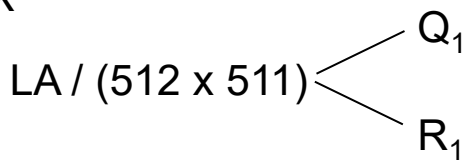
- Mapping with 512-word blocks and one-word pointers. LA is measured in words
  - Linked scheme – add index blocks as the file grows
  - Multi-level indexing



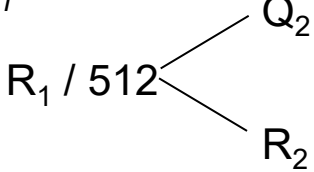


Indexed Allocation – Linked Scheme

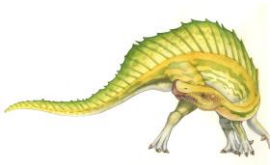
- 512-word blocks, one-word pointers. Reserve one index entry for the next link



- Outer-level mapping scheme
  - Q_1 = block of index table
  - R_1 is used as follows



  - Inner-level mapping scheme
    - Q_2 = displacement into block of index table
    - R_2 displacement into block of file

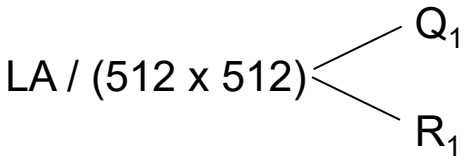


Troy University Resources

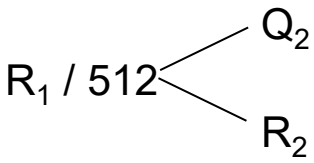


Indexed Allocation – Two-level Scheme

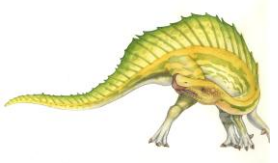
- 4 KiB blocks with 4-byte pointers address 1,048,576 data blocks (4 GiB). Mapping below uses 512-word blocks and one-word pointers



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

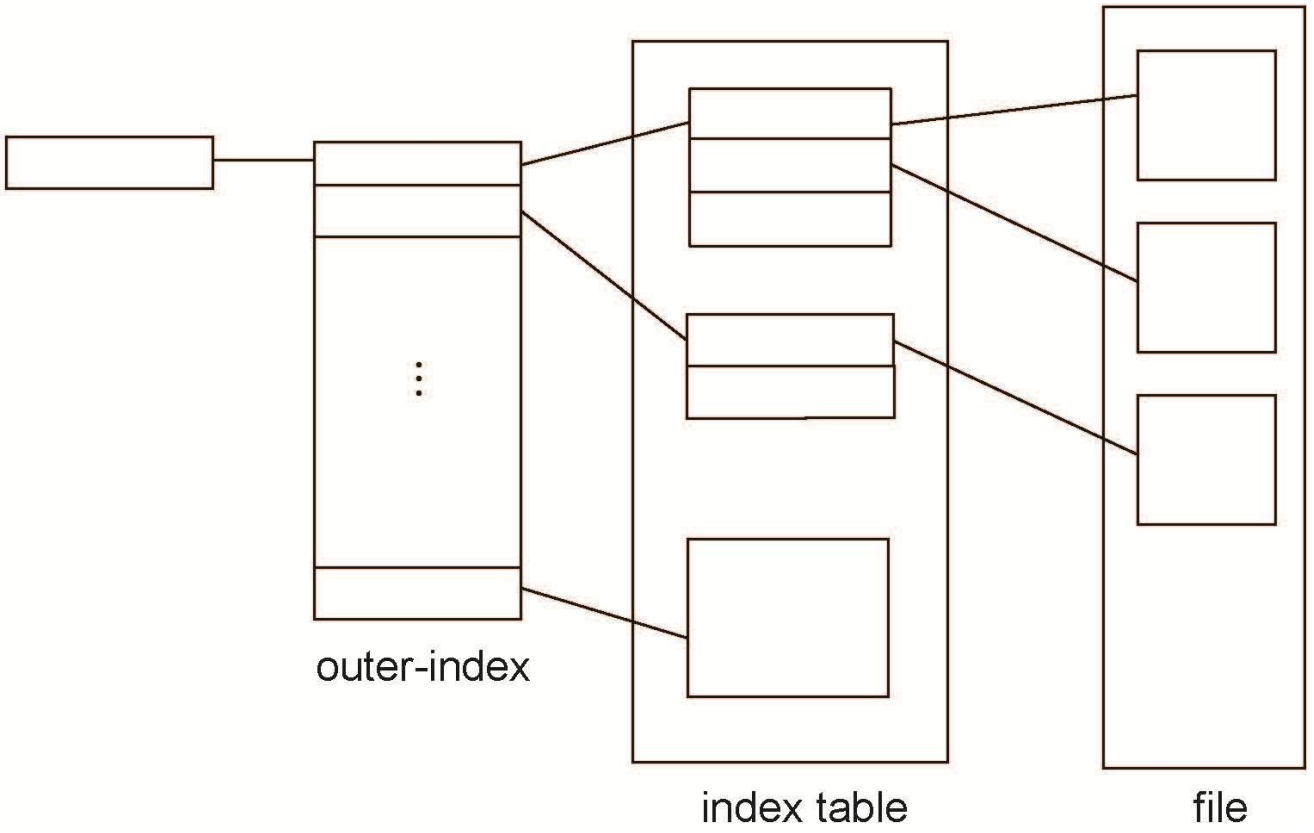


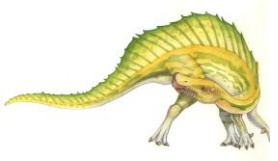
  - Mapping scheme for index level:
    - Q_2 = displacement into block of index table
    - R_2 displacement into block of file





Indexed Allocation – Two-Level Scheme

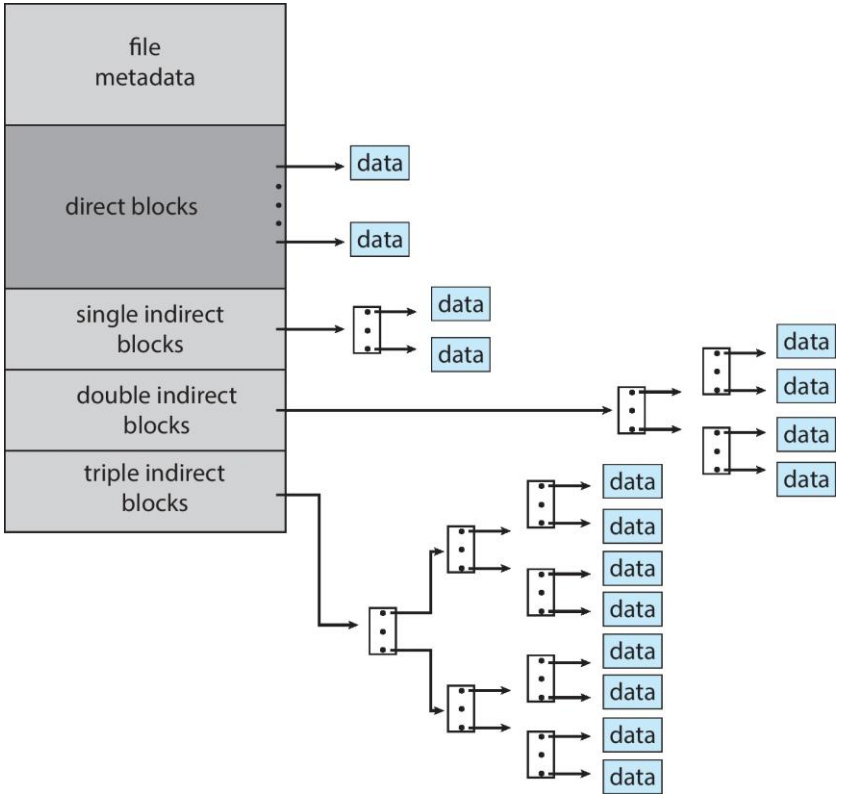




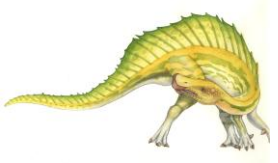


Combined Scheme : UNIX UFS

- 4K bytes per block, 32-bit addresses



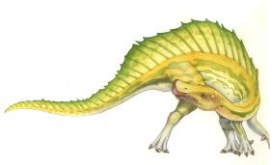
- More index blocks than can be addressed with 32-bit file pointer





Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
  - Select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead
- For NVM, no disk head so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - Goal is to reduce CPU cycles and overall path needed for I/O

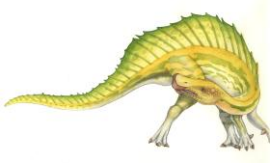


Troy University Resources



Performance (Cont.)

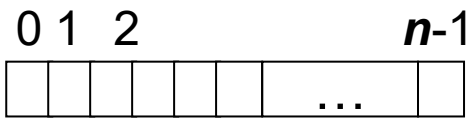
- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 636 million instructions 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

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

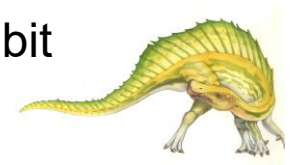


$$bit[i] = \begin{cases} 1 \Rightarrow block[i] \text{ free} \\ 0 \Rightarrow block[i] \text{ occupied} \end{cases}$$

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

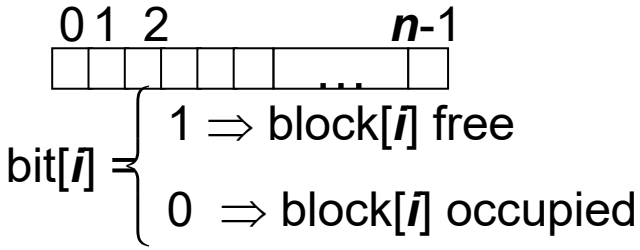


Troy University Resources



Free-Space Management

- File system maintains **free-space list** to track available blocks
- **Bit vector** or **bit map** (***n*** blocks)



- Bit map requires extra space
  - Example:

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

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

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

    if clusters of 4 blocks -> 8 MiB of memory

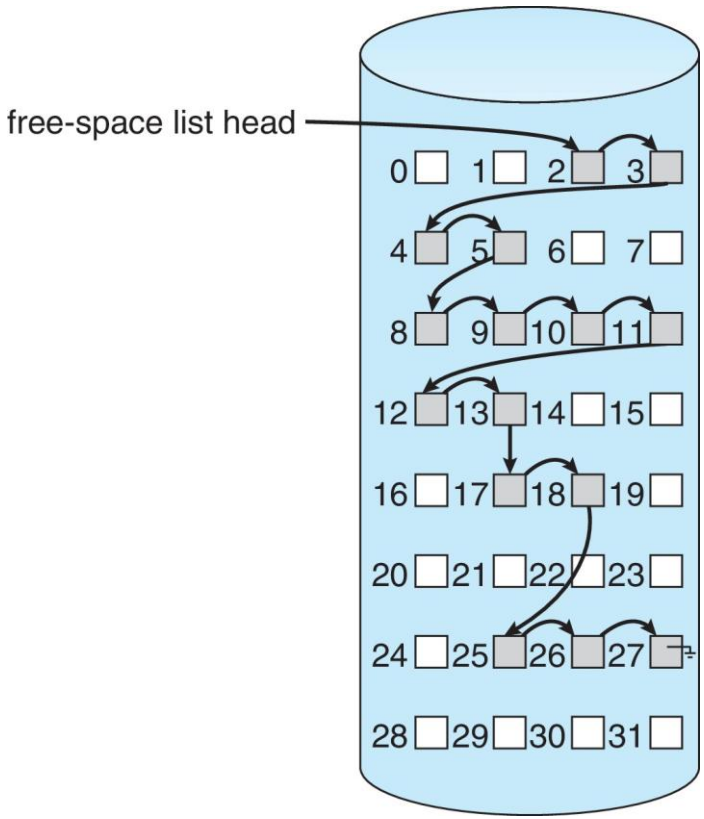
- Easy to get contiguous files

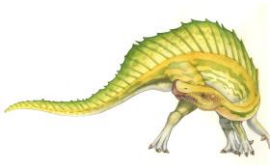




Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - Free blocks themselves store the list pointers
  - No need to traverse the entire list (if # free blocks recorded)



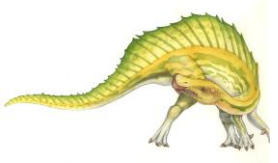


Troy University Resources



Free-Space Management (Cont.)

- 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 (Cont.)

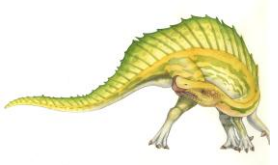
- Space Maps
  - Used in **ZFS**
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps cannot fit in memory ➔ thousands of I/Os
  - Divides device space into **metaslab** units and manages metaslabs
    - Given volume can contain hundreds of metaslabs
  - Each 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





TRIMing Unused Blocks

- HDDS overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- NAND flash SSDs cannot overwrite programmed pages in place
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM informs an SSD that specified logical blocks are no longer in use
    - Can be garbage collected or if block is free, now block can be erased





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



Troy University Resources



Efficiency and Performance (Cont.)

- 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
    - Caching is possible, but completion requires the requested durability
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Cached writes may return before device I/O completes





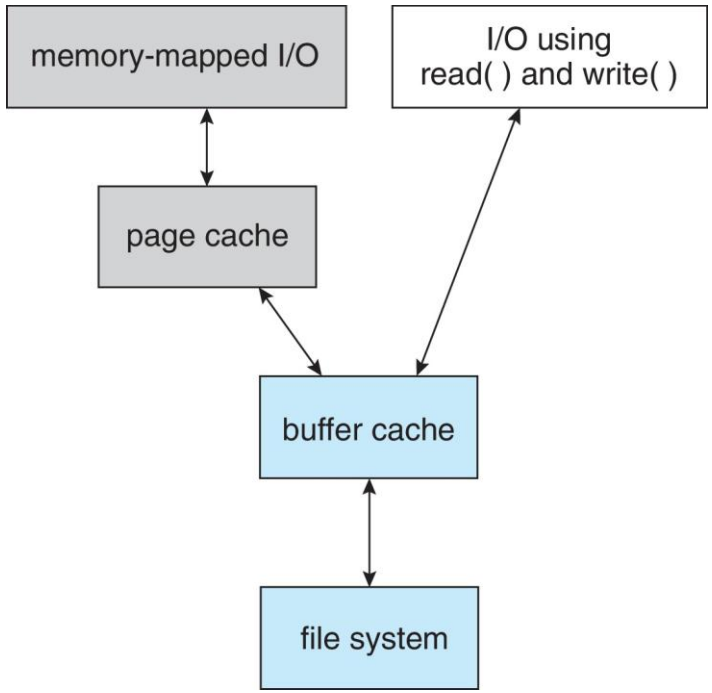
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped file access uses the page cache
- Without a unified cache, ordinary file I/O uses a separate buffer cache
- This leads to the following figure



Troy University Resources

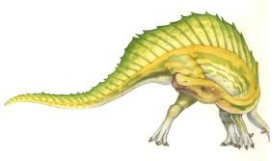
I/O Without a Unified Buffer Cache





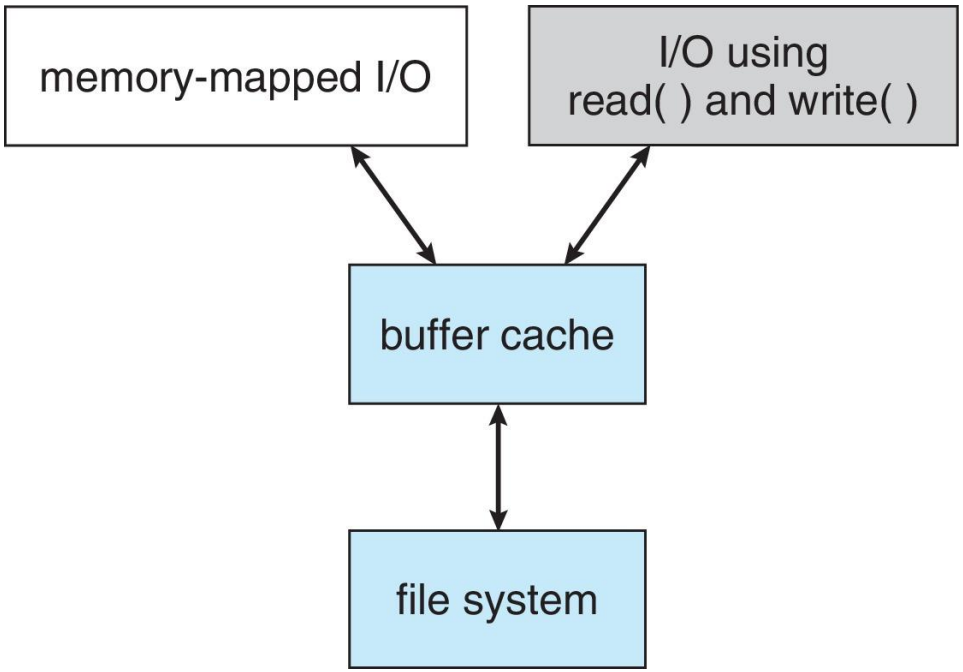
Unified Buffer Cache

- 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?



Troy University Resources

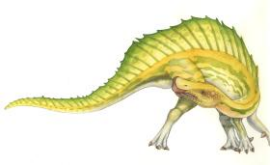
I/O Using a Unified Buffer Cache





Recovery

- **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

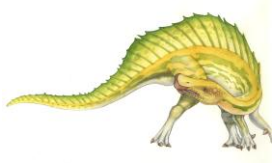


Troy University Resources



Journaling File Systems

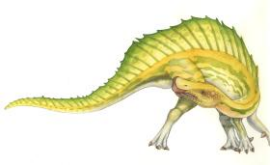
- **Journaling** file systems group metadata updates into **transactions recorded in a log**
- All transactions are written to a log
  - A transaction commits once its log records and commit record are durable
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- Committed changes are later checkpointed to their home file-system locations
  - After checkpointing completes, the corresponding log space can be reused
- After a crash, replay committed transactions and ignore incomplete transactions
- Faster crash recovery restores metadata consistency under the journal's failure model





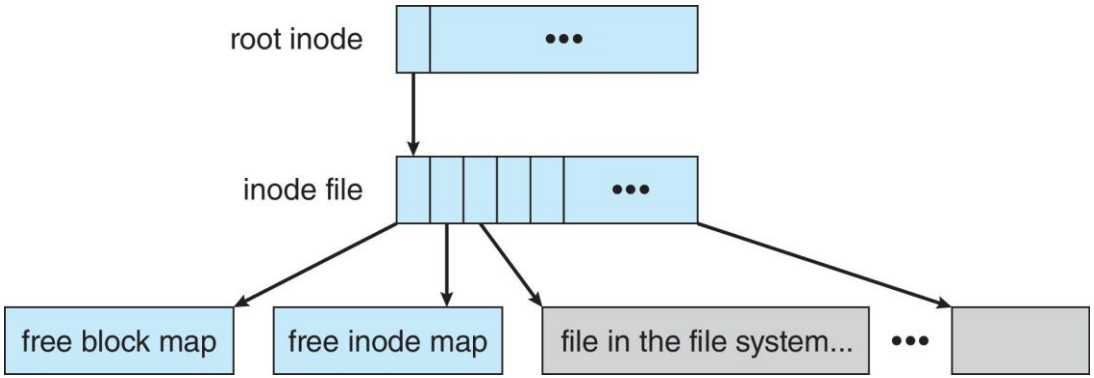
Example: WAFL File System

- 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



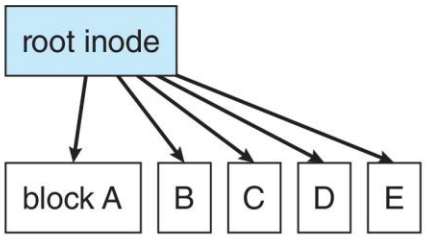
Troy University Resources

The WAFL File Layout

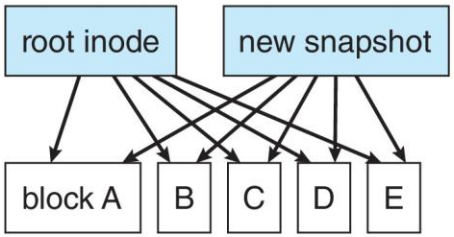




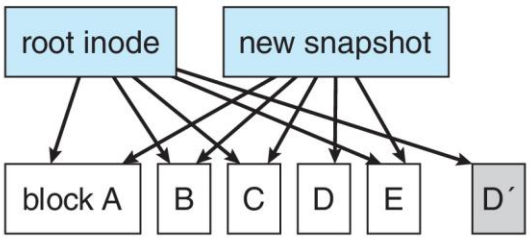
Snapshots in WAFL



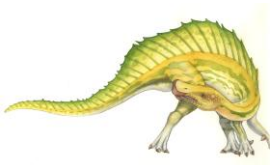
(a) Before a snapshot.



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



(c) After block D has changed to D´.



Troy University Resources



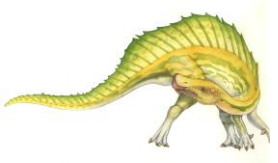
The Apple File System

In 2017, Apple, Inc., released a new file system to replace its 30-year-old HFS+ file system. HFS+ had been stretched to add many new features, but as usual, this process added complexity, along with lines of code, and made adding more features more difficult. Starting from scratch on a blank page allows a design to start with current technologies and methodologies and provide the exact set of features needed.

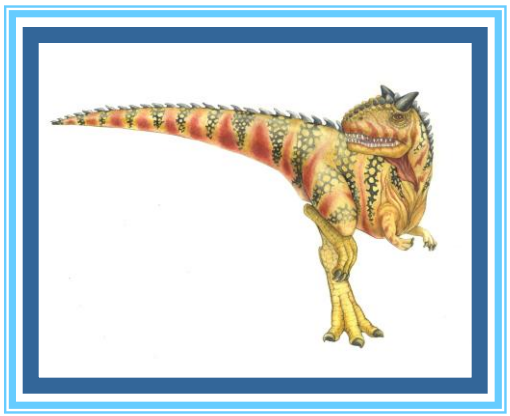
Apple File System (**APFS**) is a good example of such a design. Its goal is to run on all current Apple devices, from the Apple Watch through the iPhone to the Mac computers. Creating a file system that works in watchOS, I/Os, tvOS, and macOS is certainly a challenge. APFS is feature-rich, including 64-bit pointers, clones for files and directories, snapshots, space sharing, fast directory sizing, atomic safe-save primitives, copy-on-write design, encryption (single- and multi-key), and I/O coalescing. It understands NVM as well as HDD storage.

Most of these features we've discussed, but there are a few new concepts worth exploring. **Space sharing** is a ZFS-like feature in which storage is available as one or more large free spaces (**containers**) from which file systems can draw allocations (allowing APFS-formatted volumes to grow and shrink). **Fast directory sizing** provides quick used-space calculation and updating. **Atomic safe-save** is a primitive (available via API, not via file-system commands) that performs renames of files, bundles of files, and directories as single atomic operations. I/O coalescing is an optimization for NVM devices in which several small writes are gathered together into a large write to optimize write performance.

Apple chose not to implement RAID as part of the new APFS, instead depending on the existing Apple RAID volume mechanism for software RAID. APFS is also compatible with HFS+, allowing easy conversion for existing deployments.

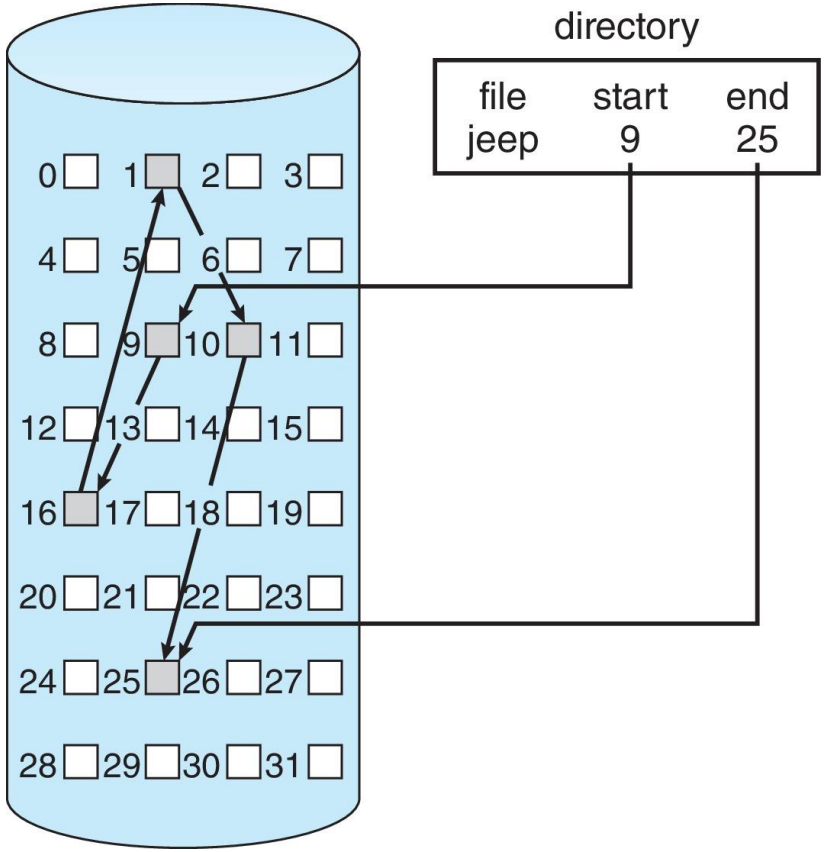


End of Chapter 14





Linked Allocation

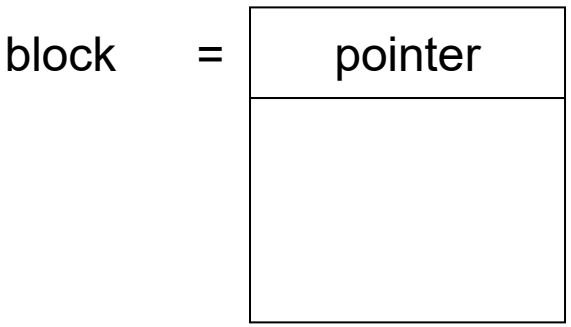




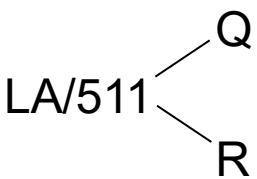


Linked Allocation

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

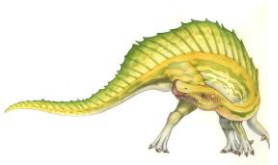


- Mapping (512-word blocks, one-word link)



Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 1

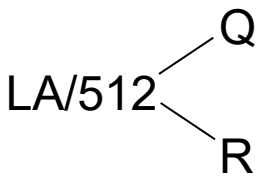


Troy University Resources

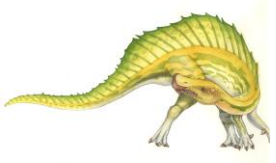


Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Toy model: 512-word blocks and one-word block pointers. One 512-entry index addresses 256K words. LA is a word offset



Q = displacement into index table
R = displacement into block

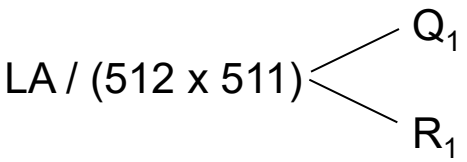




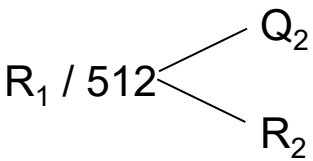
Indexed Allocation – Linked Scheme

- Mapping with 512-word blocks and one-word pointers. LA is measured in words

- Linked scheme – add index blocks as the file grows



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

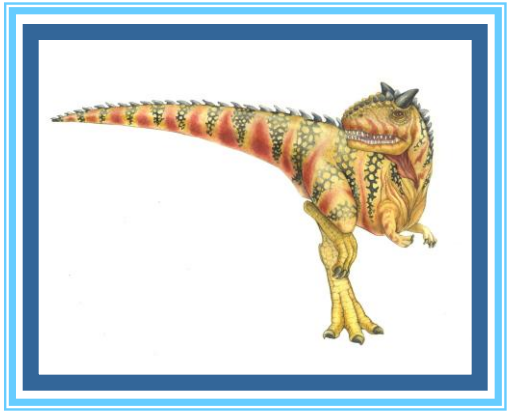


Q_2 = displacement into block of index table
R_2 displacement into block of file:



Troy University Resources

End of Chapter 14



Silberschatz, Galvin and Gagne ©2018



In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info
- The following figure illustrates the necessary file system structures provided by the operating systems
- Part (a) of the figure shows opening a file
- Part (b) of the figure shows reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

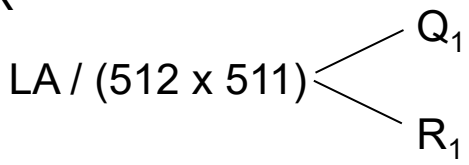


Troy University Resources

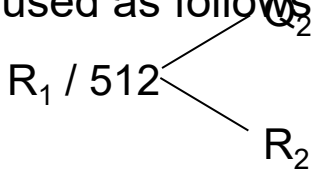


Indexed Allocation – Linked Scheme

- 512-word blocks, one-word pointers. Reserve one index entry for the next link



- Outer-level mapping scheme
  - Q_1 = block of index table
  - R_1 is used as follows
- R_1 is used as follows



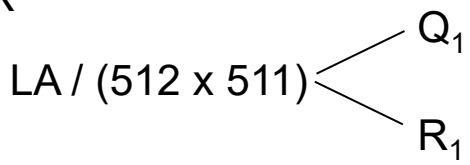
- Inner-level mapping scheme
  - Q_2 = displacement into block of index table
  - R_2 displacement into block of file



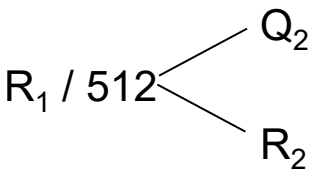


Indexed Allocation – Linked Scheme

- 512-word blocks, one-word pointers. Reserve one index entry for the next link



- Outer-level Mapping scheme
  - Q_1 = block of index table
  - R_1 is used as follows
- Inner-level Mapping scheme



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

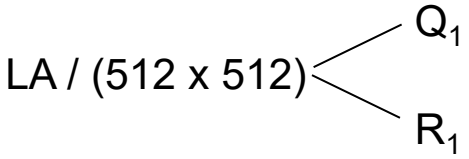


Troy University Resources



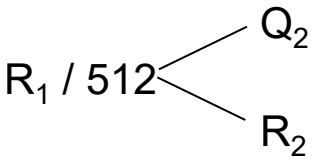
Indexed Allocation – Two-level Scheme

- 4 KiB blocks with 4-byte pointers address 1,048,576 data blocks (4 GiB). Mapping below uses 512-word blocks and one-word pointers



- Mapping scheme:

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:

