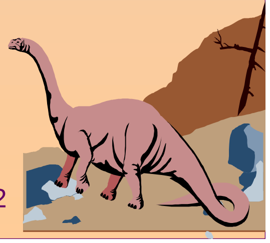
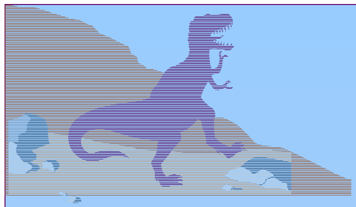
Chapter 12: File System Implementation

*This chapter is concerned with the **details** associated with file systems residing on secondary storage.*

Specific implementation issues are explored using the <u>disk</u> as the secondary storage device.

- File System Structure
- File System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS





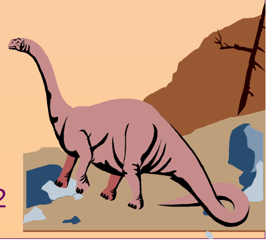
File-System Structure

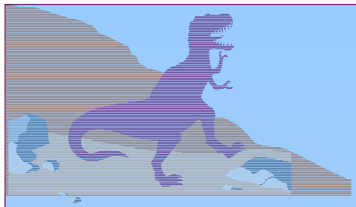
Pertinent Disk Details

- *The physical unit of transfer is a disk sector (e.g., 512 bytes).*
- *Sectors can be written in place.*
- *Any given sector can be accessed directly.*

- *The OS imposes a file system for efficient and convenient access to the disk.*
- *The file system design deals with two distinct matters:*

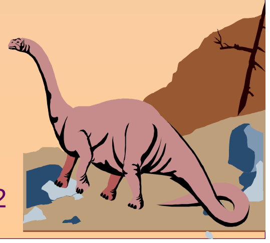
1. *How should the file system look to the user.*
2. *Creating data structures and algorithms to map the logical file system onto the physical secondary-storage device.*





Blocks and Fragments [4.2BSD]

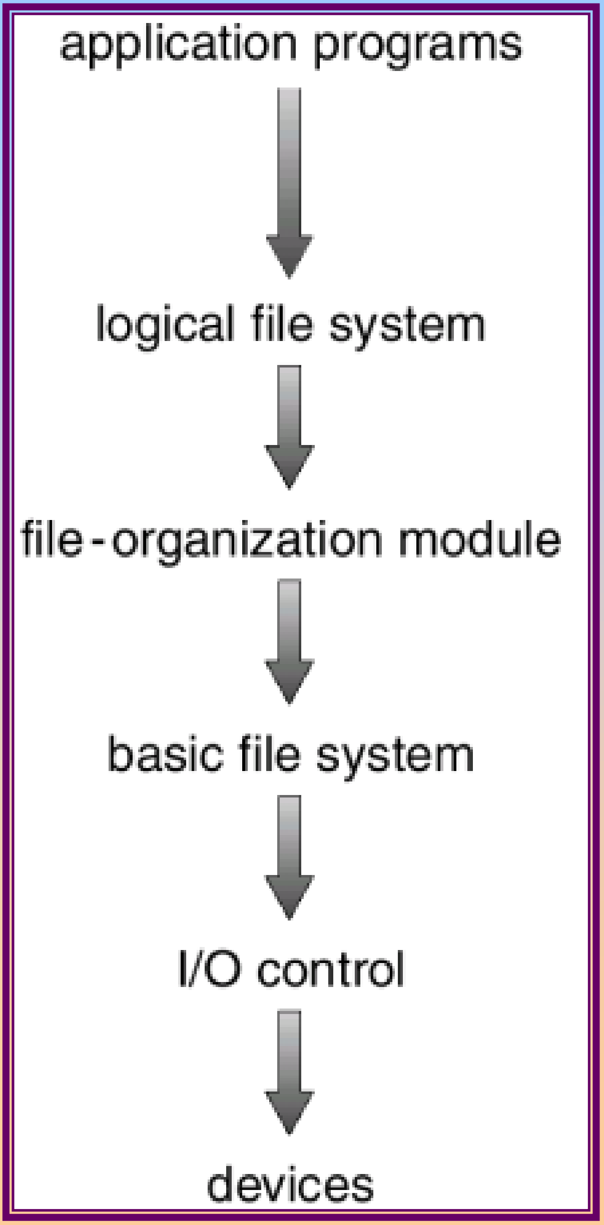
- Logical transfer of data is in *blocks.*
- *Blocks* are “chunks” or clusters of disk sectors.
- block size decision:: {a time space tradeoff}
- Uses two sizes **block** and **fragment**
  - E.g. 4KB block; 1KB fragment

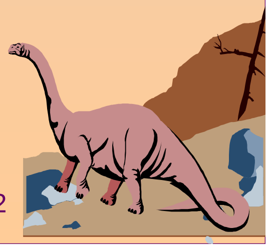


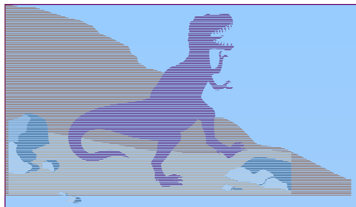
Layered File System

A layered design abstraction

- I/O control :: device drivers and interrupt service routines that perform the actual block transfers.
- Basic file system :: issues generic *low-level* commands to device drivers.
- File organization :: translates logical block addresses to physical block addresses.
- Logical file system :: handles *metadata* that includes file-system structure (e.g., directory structure and file control blocks (FCB's)).





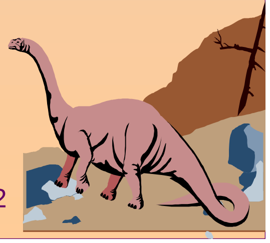


In-Memory File System Structures

On-disk and in-memory structures are needed to implement a file system:

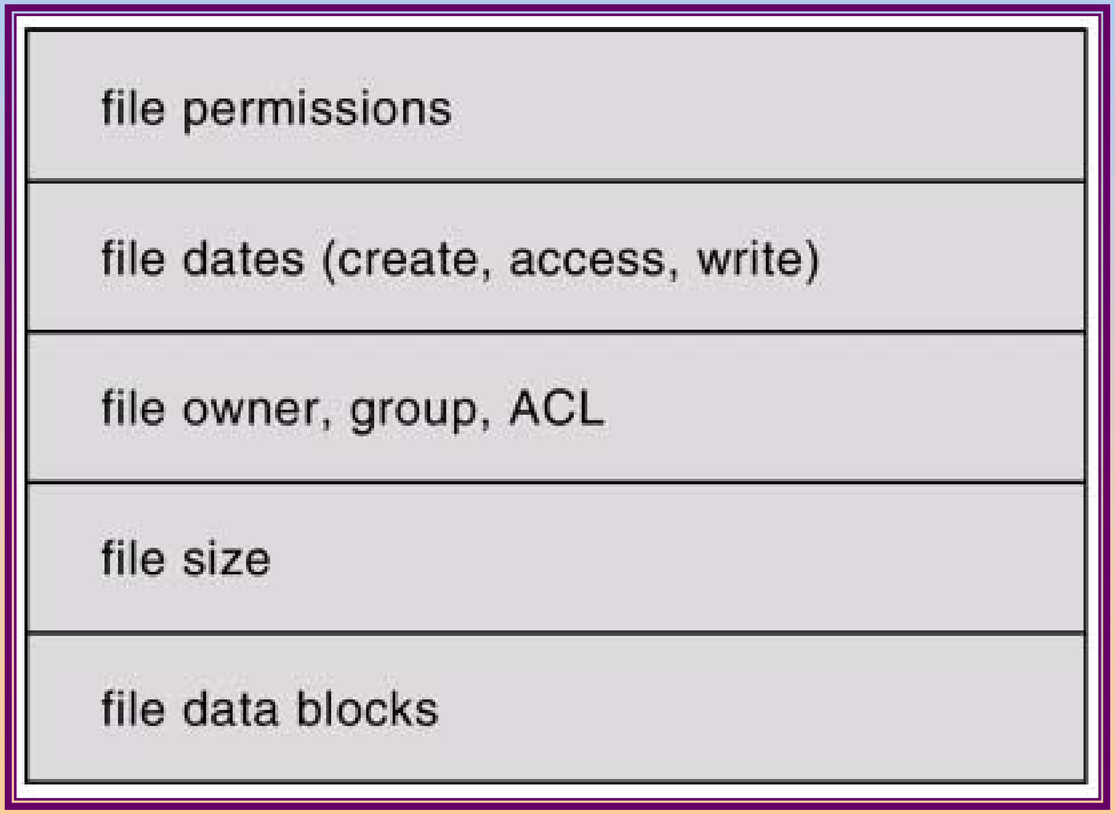
On disk::

1. Boot control block: *needed to boot OS from a disk partition.*
2. Partition control block: *holds details about partition (e.g., blocks in partition, free-block count, ...)* **`superblock [UNIX]`**, **`Master File Table [NTFS]`**
3. Directory structure: *to organize files.*
4. FCB (or inode) :



A Typical File Control Block

[Linux] inode is the term for FCB



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

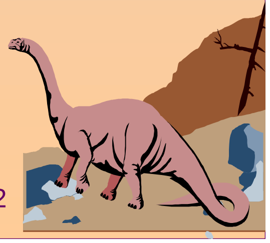
In-Memory File System Structures

On-disk and in-memory structures needed to implement a file system:

In-memory::

1. Per-process open-file table
2. System-wide open-file table
3. In-memory directory structure
4. In-memory partition table

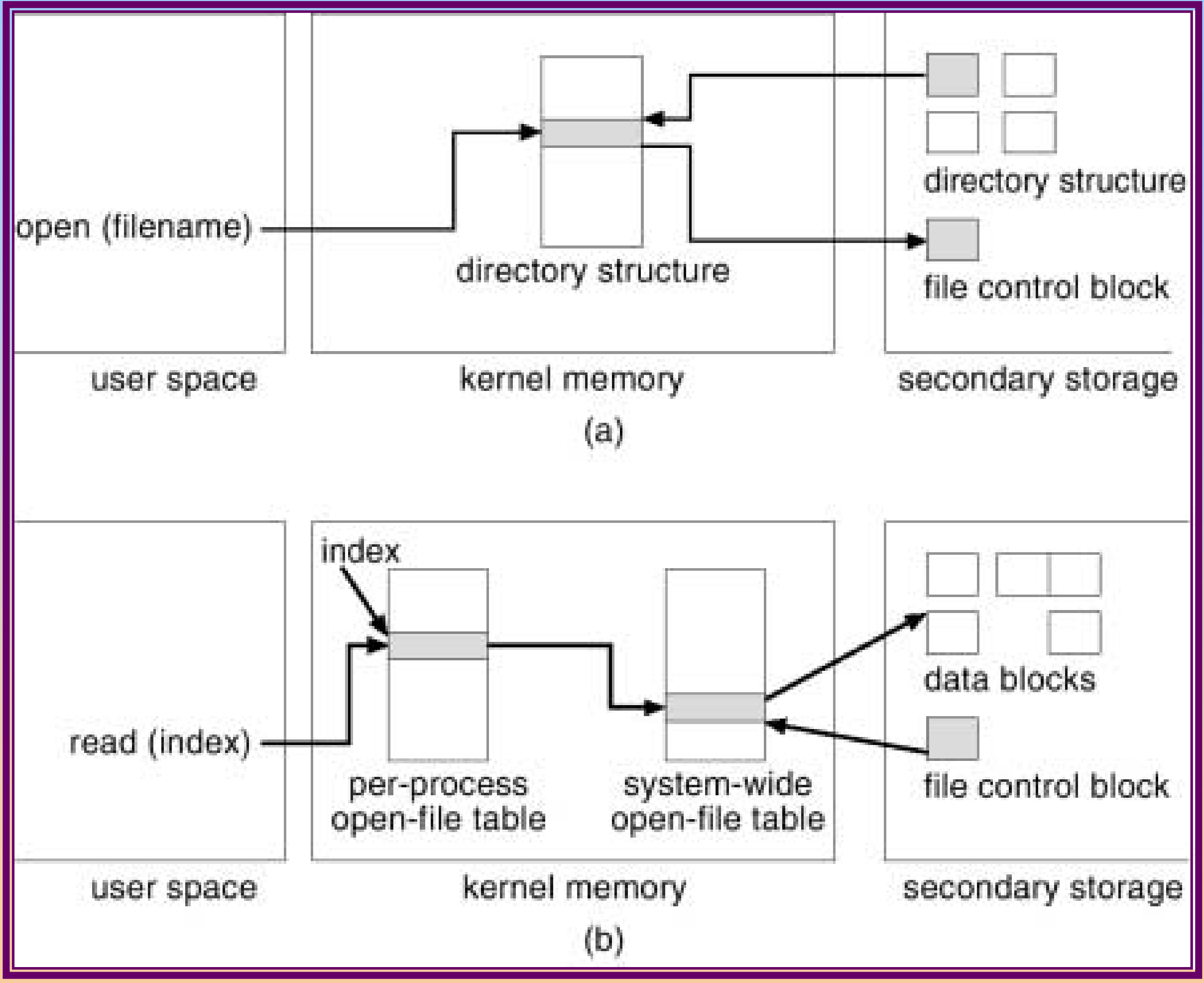
- *In some OS's file system scheme used as interface to other system aspects.* [Unix uses system-wide open table for networking - e.g.,socket descriptors.]
- *Caching used extensively – namely, all the information about an open file is in memory except actual data blocks.*



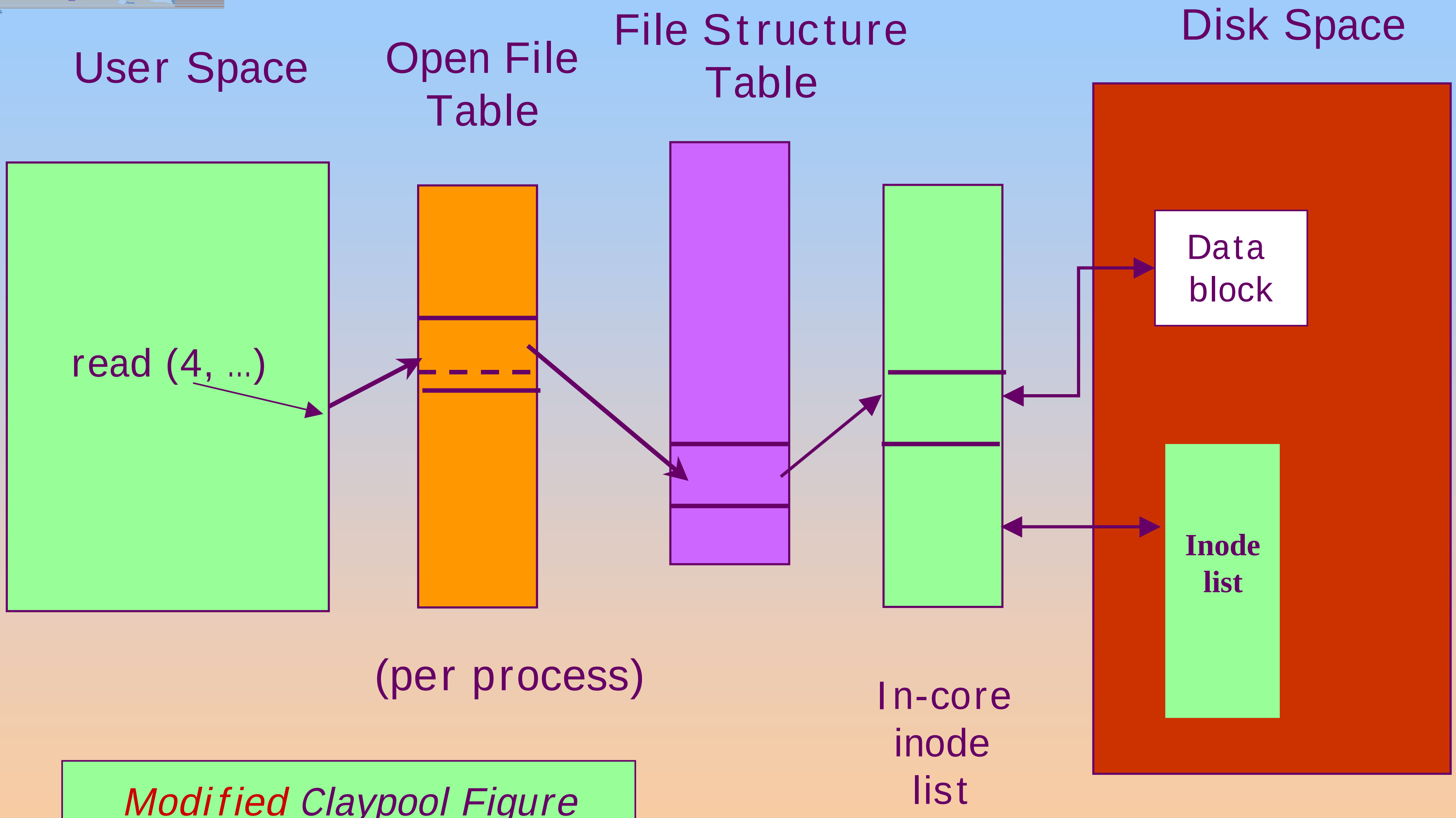
In-Memory File System Structures

(a) Open operation

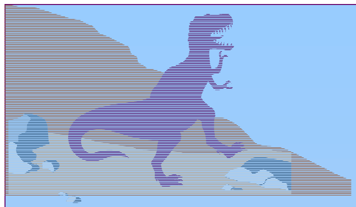
(b) Read operation



File System Implementation

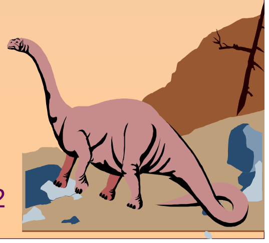


Modified Claypool Figure
[Figure A.7 page 837]

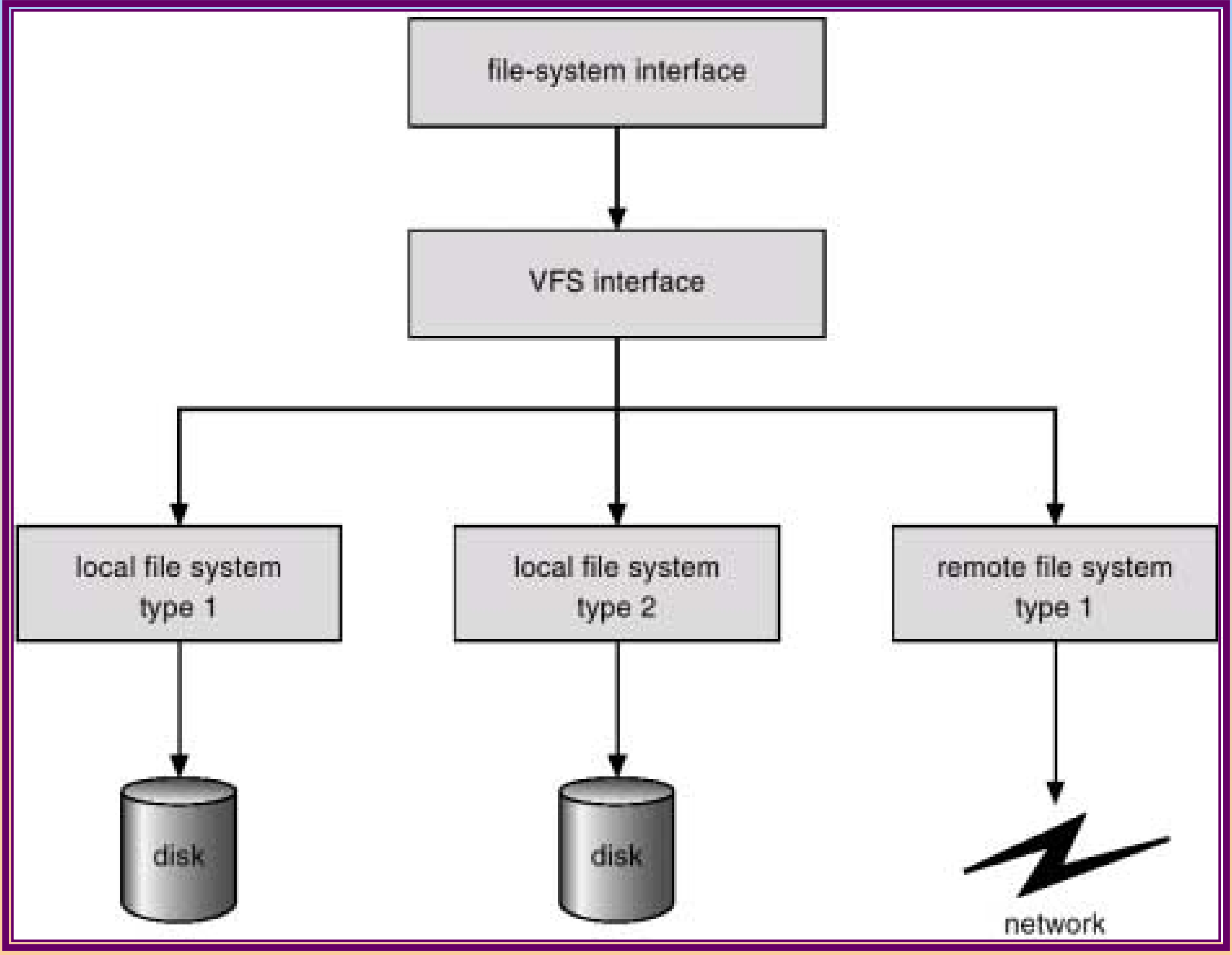


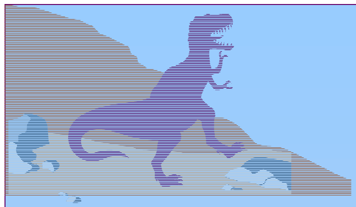
Virtual File Systems

- Virtual File Systems (VFS) 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.
- The API is to the VFS interface, rather than any specific type of file system.
- *VFS provides a mechanism for integrating NFS into the OS.*



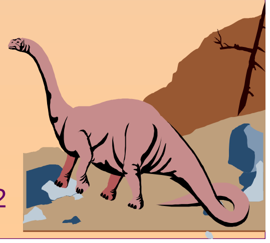
Schematic View of Virtual File System





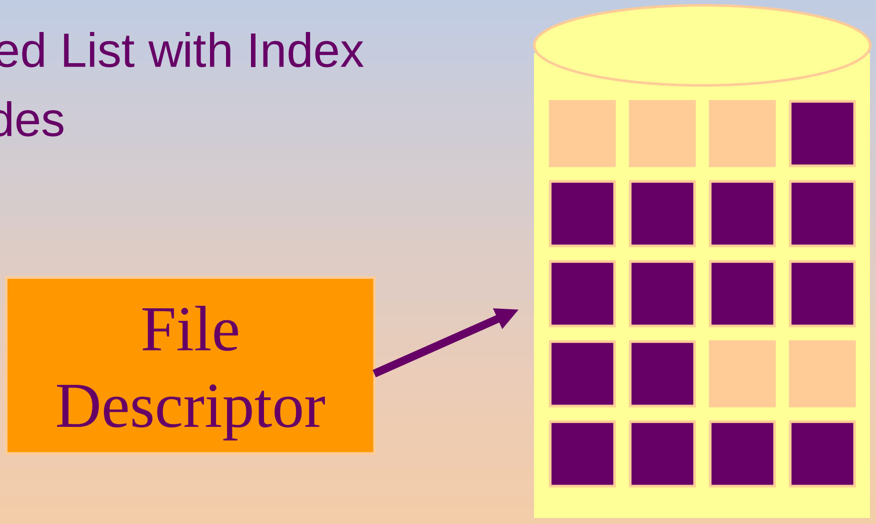
Directory Implementation

- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute ***due to linear search***
- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size ***{disadvantage: hash function is dependent on fixed size}***



File System Implementation

- Which blocks with which file?
- File descriptor implementations:
  - Contiguous
  - Linked List
  - Linked List with Index
  - I-nodes



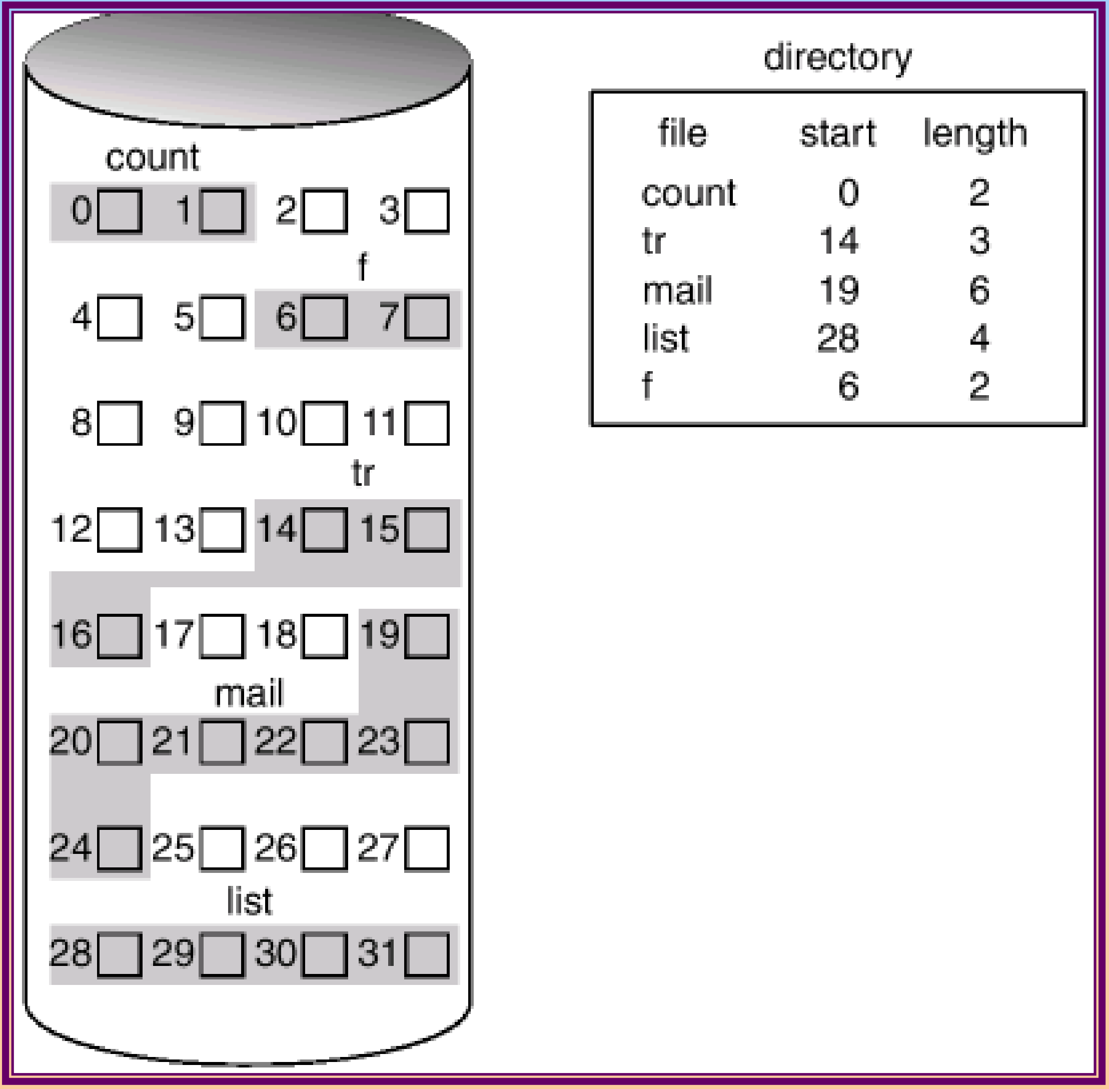
Allocation Methods

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

Contiguous Allocation

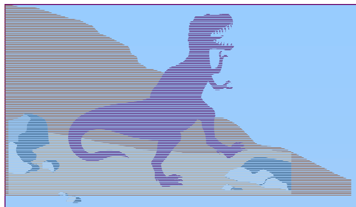
- Each file occupies a set of contiguous blocks on the disk.
- Simple – only starting location (block #) and length (number of blocks) are required.
- Random access.
- Wasteful of space (dynamic storage-allocation problem).
- Files cannot grow.

Contiguous Allocation of Disk Space



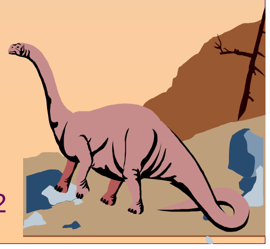
directory

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



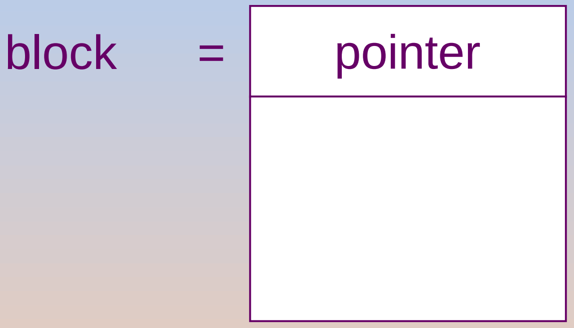
Extent-Based Systems

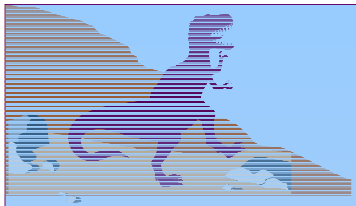
- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme.
- Extent-based file systems allocate disk blocks in **extents**.
- An **extent** is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.



Linked Allocation

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





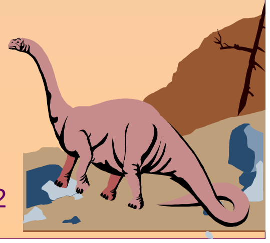
Linked Allocation (Cont.)

- Simple – need only starting address
- Free-space management system – no waste of space
- No random access
- Mapping

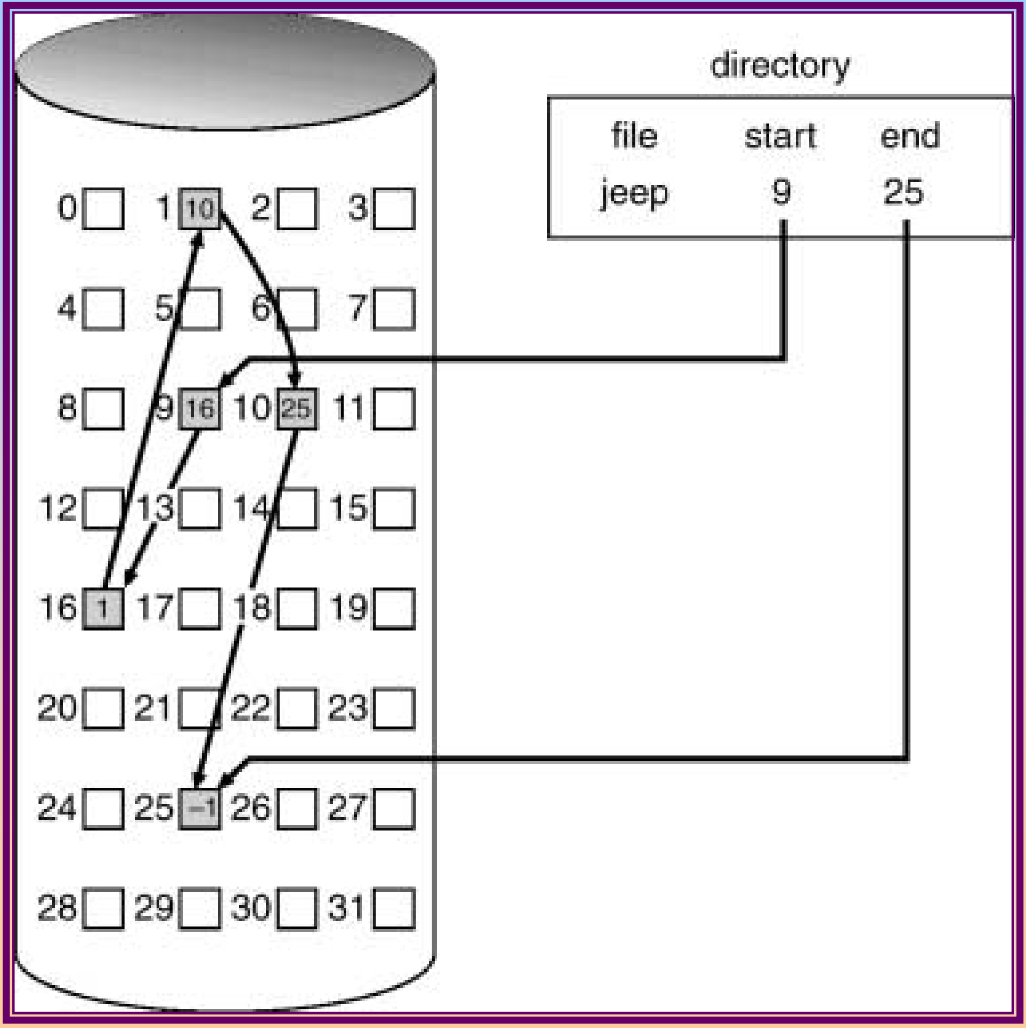
$$\begin{tikzcd}
 & Q \\
LA/511 \arrow[ur, no head] \arrow[dr, no head] & \\
 & R
\end{tikzcd}$$

Block to be accessed is the Qth block in the linked chain of blocks representing the file.
Displacement into block = R + 1

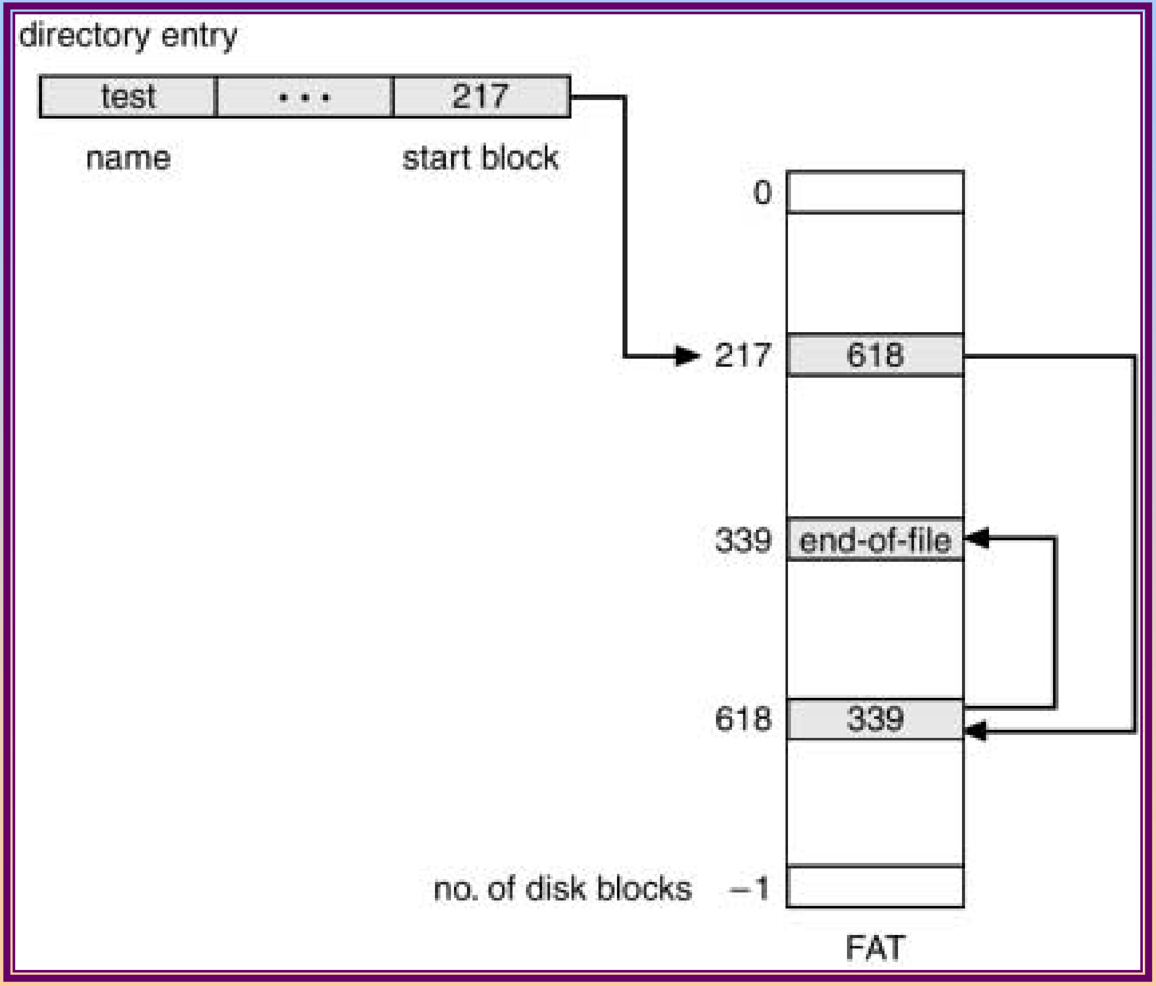
File-allocation table (FAT) – disk-space allocation used by MS-DOS and OS/2.



Linked Allocation

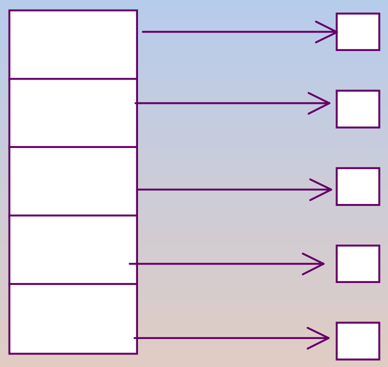


File-Allocation Table



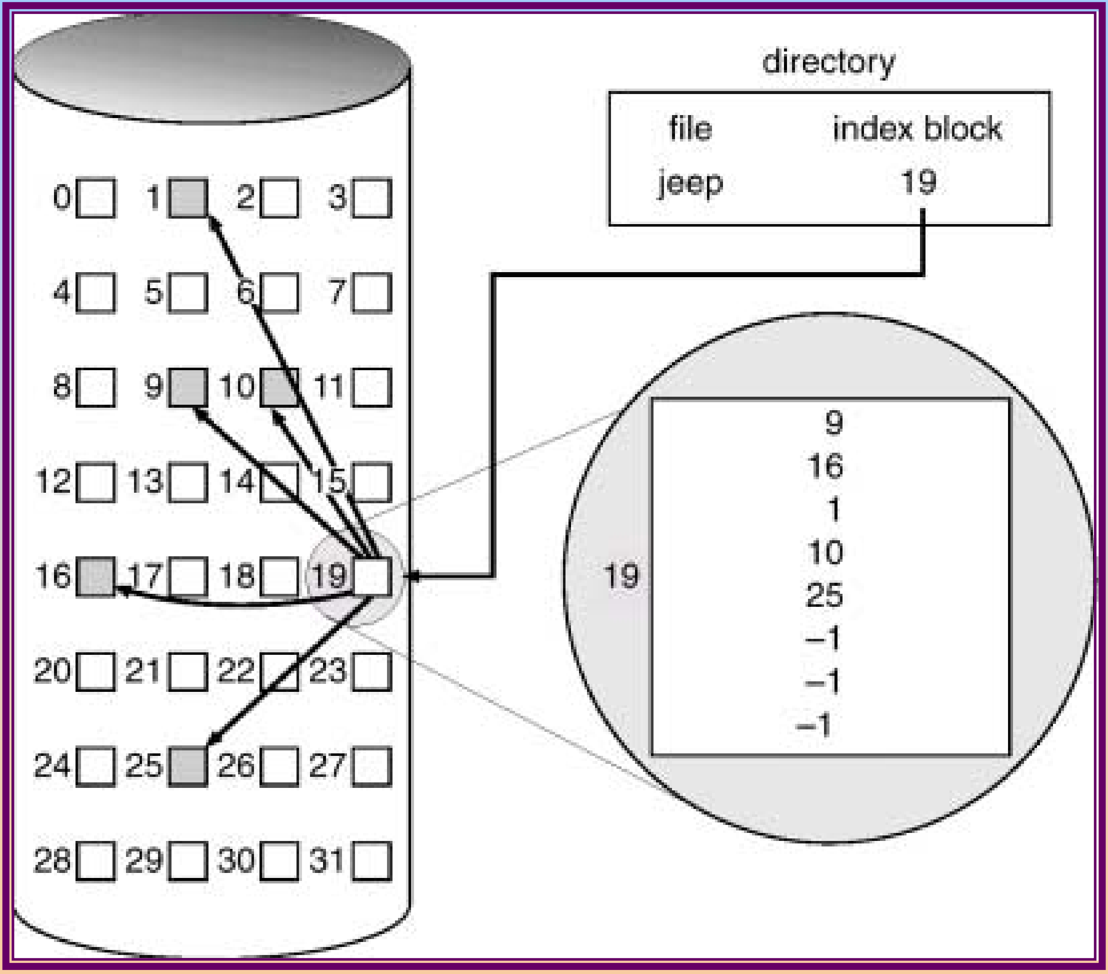
Indexed Allocation

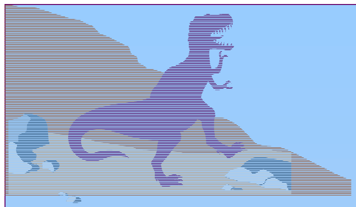
- Brings all pointers together into the *index block.*
- Logical view.



index table

Example of Indexed Allocation



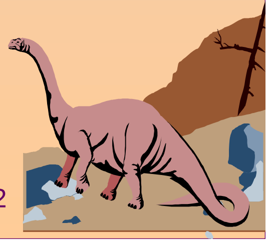


Indexed Allocation (Cont.)

- 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 words and block size of 512 words. We need only 1 block for index table.

$$\begin{tikzcd}
 & Q \\
LA/512 \arrow[ur, no head] \arrow[dr, no head] & \\
 & R
\end{tikzcd}$$

Q = displacement into index table
R = displacement into block



Indexed Allocation – Mapping (Cont.)

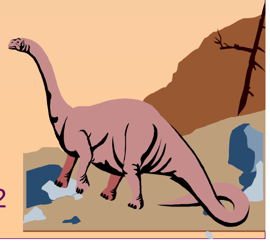
- Mapping from logical to physical in a file of unbounded length (block size of 512 words).
- Linked scheme – Link blocks of index table (no limit on size).

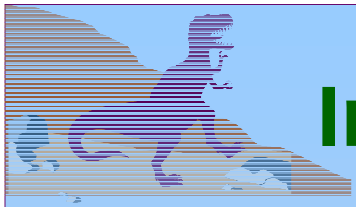
$$\begin{tikzcd}[column sep=large, row sep=small] & Q_1 \\ LA / (512 \times 511) \arrow[ur, dash] \arrow[dr, dash] & \\ & R_1 \end{tikzcd}$$

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

$$\begin{tikzcd}[column sep=large, row sep=small] & Q_2 \\ R_1 / 512 \arrow[ur, dash] \arrow[dr, dash] & \\ & R_2 \end{tikzcd}$$

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





Indexed Allocation – Mapping (Cont.)

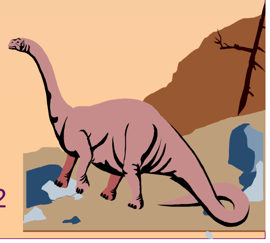
- Two-level index (maximum file size is 512^3)

$$\begin{tikzcd} & Q_1 \\ \text{LA} / (512 \times 512) \arrow[ur, dash] \arrow[dr, dash] & \\ & R_1 \end{tikzcd}$$

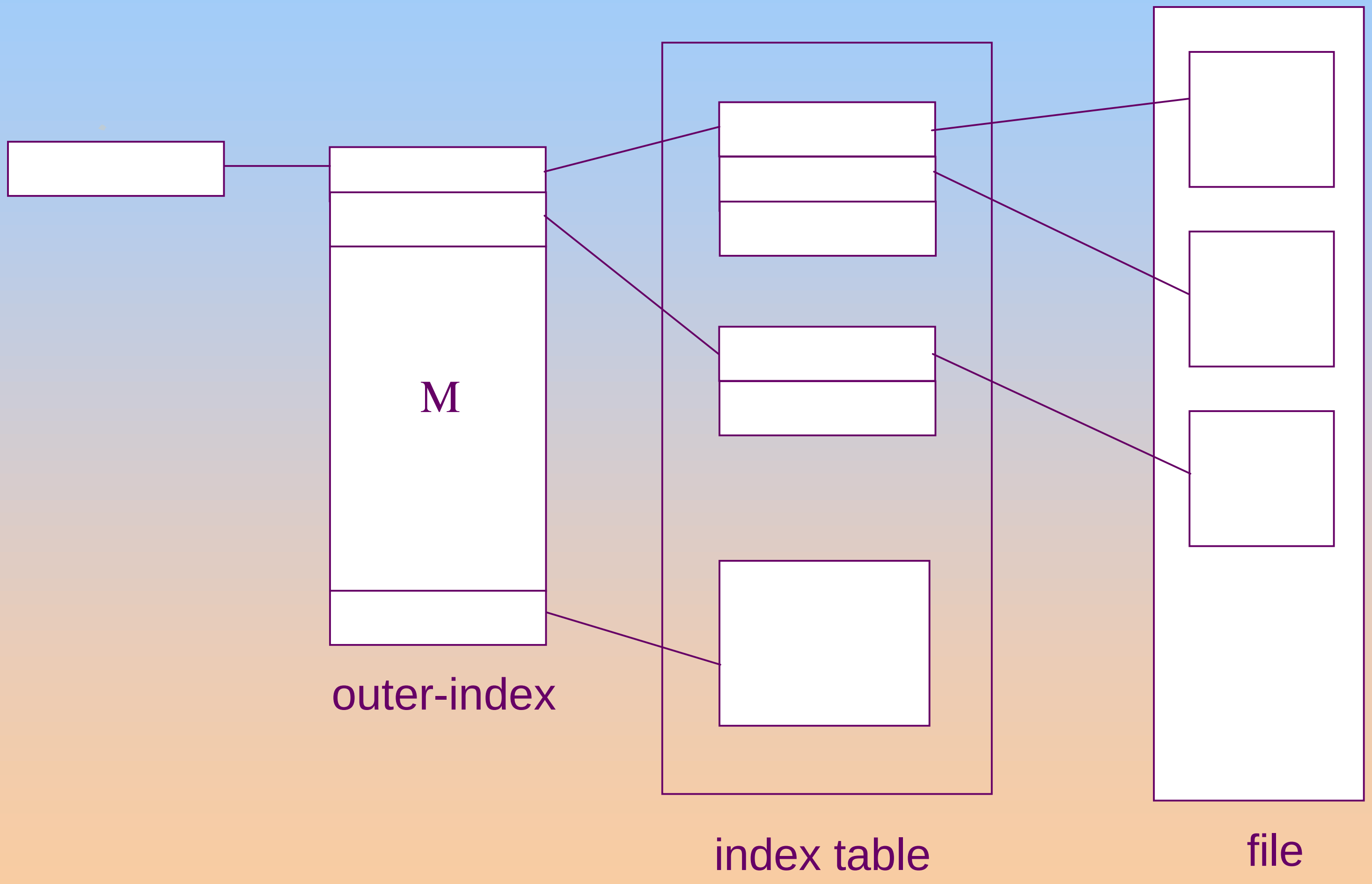
Q_1 = displacement into outer-index
R_1 is used as follows:

$$\begin{tikzcd} & Q_2 \\ R_1 / 512 \arrow[ur, dash] \arrow[dr, dash] & \\ & R_2 \end{tikzcd}$$

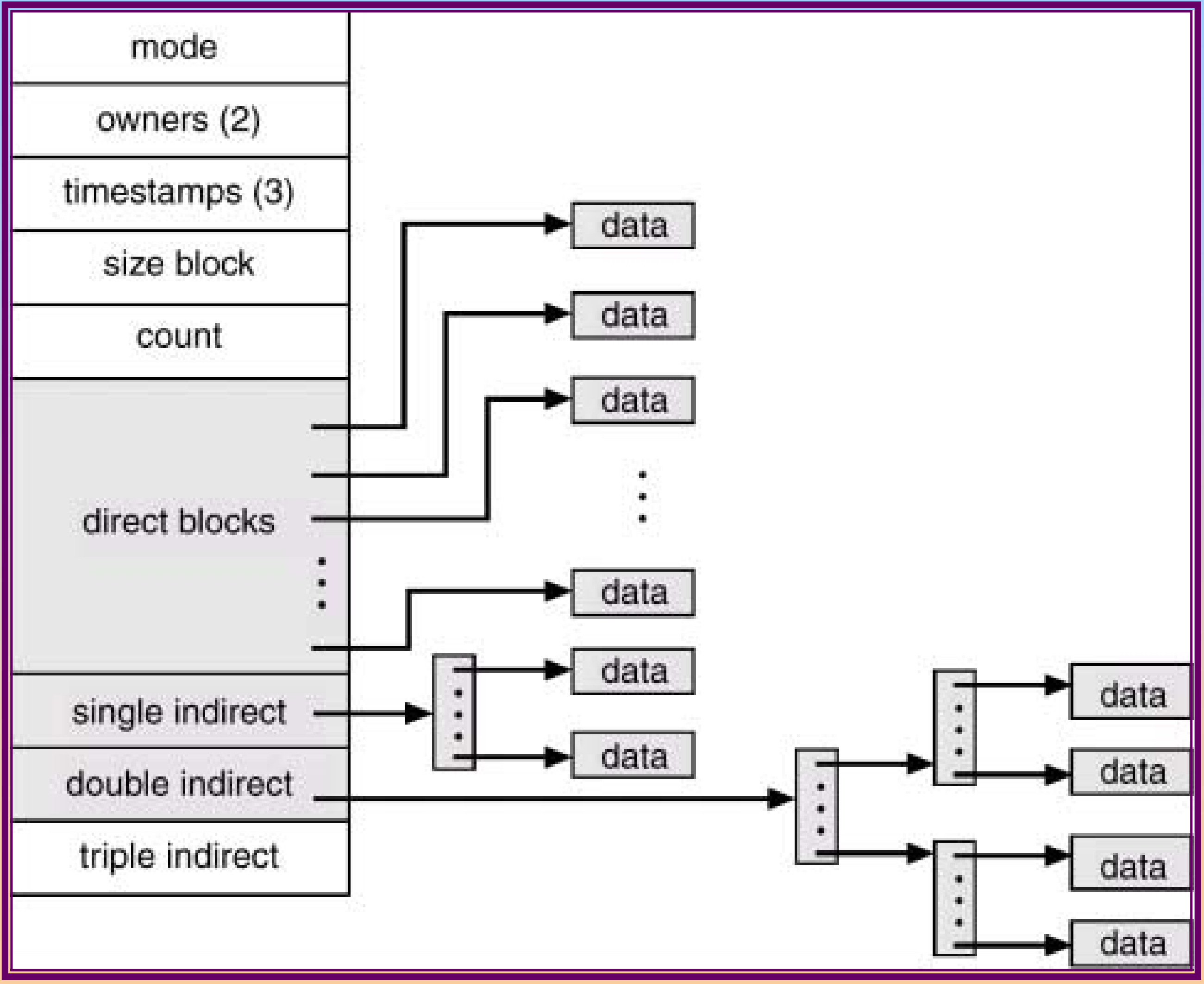
Q_2 = displacement into block of index table
R_2 displacement into block of file:

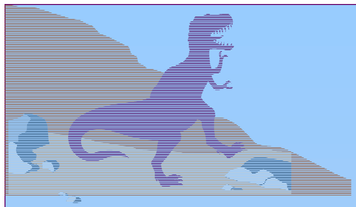


Indexed Allocation – Mapping (Cont.)



Combined Scheme: UNIX (4K bytes per block)



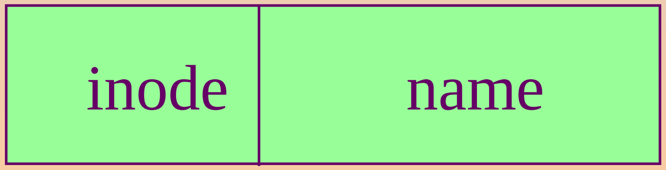


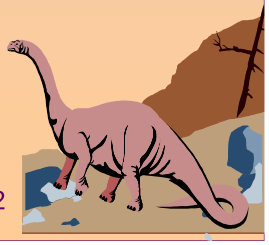
Hierarchical Directory (Unix)

Claypool Figure

- Tree
- Entry:
  - name
  - inode number (try “ls –l” or “ls –iad .”)
- example:

  `/usr/bob/mbox`





Unix Directory Example

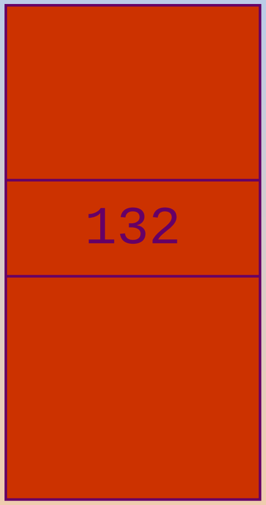
Claypool Figure

Root Directory

1	.
1	..
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

Looking up usr gives I-node 6

I-node 6



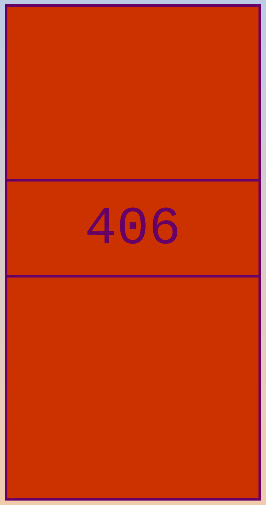
Relevant data (/usr) is in block 132

Block 132

6	.
1	..
26	bob
17	jeff
14	sue
51	sam
29	mark

Looking up bob gives I-node 26

I-node 26

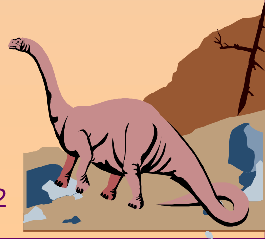


/usr/bob is in block 406

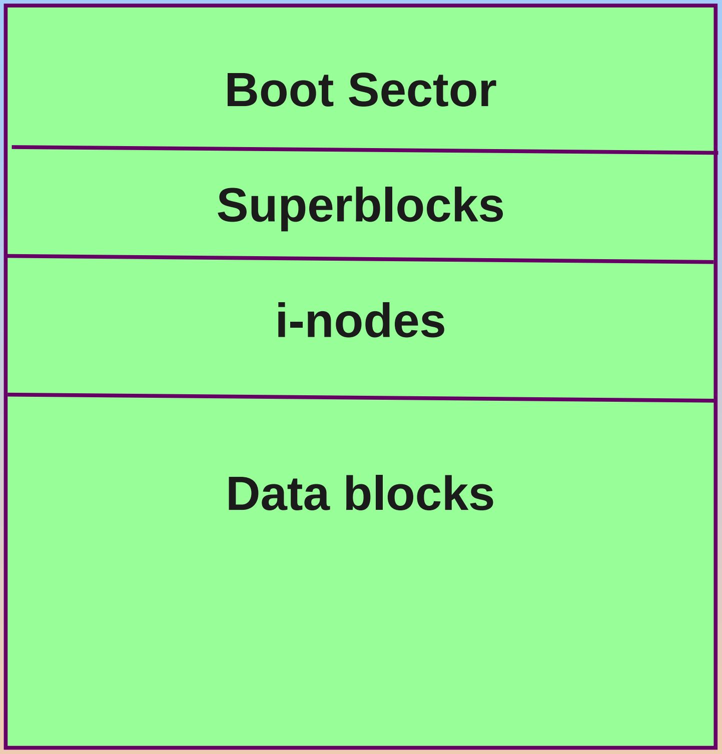
Block 406

26	.
6	..
12	grants
81	books
60	mbox
17	Linux

Aha! I-node 60 has contents of mbox

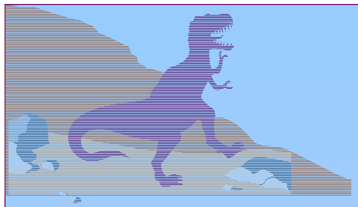


Unix Disk Layout



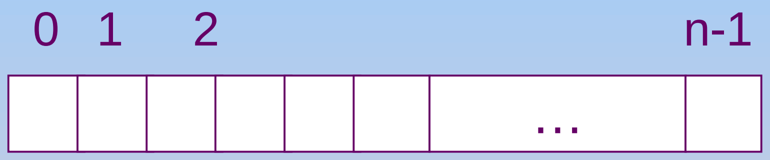
Superblock

- **Number of inodes**
- **Number of blocks**
- **Block number of I-node bit-map block**
- **first data block**
- **Max file size**
- ***Magic number –*** *identifies cpu Architecture and executable type*



Free-Space Management

- Bit vector (*n* blocks)

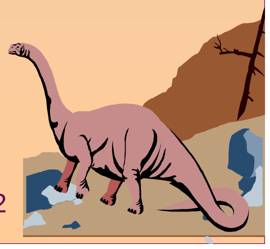


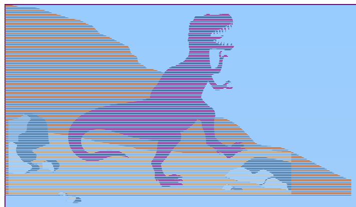
bit[*i*] = 678 0 ⇒ block[*i*] free

1 ⇒ block[*i*] occupied

Block number calculation

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





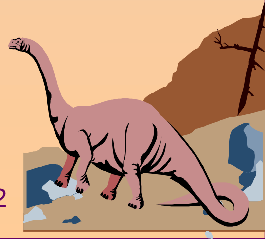
Free-Space Management (Cont.)

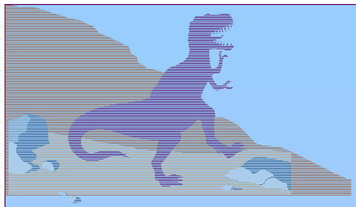
- Bit map requires extra space. Example:

  block size = 2^{12} bytes

  disk size = 2^{30} bytes (1 gigabyte)

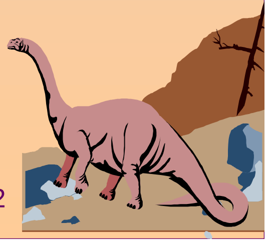
  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files
- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
- Grouping
- Counting

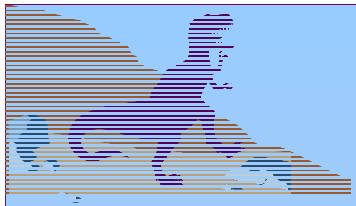




Free-Space Management (Cont.)

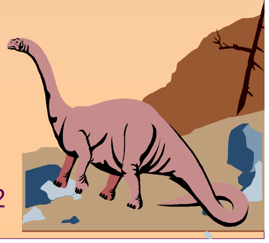
- Need to protect:
  - Pointer to free list
  - Bit map
    - ✔ Must be kept on disk
    - ✔ Copy in memory and disk may differ.
    - ✔ Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk.
  - Solution:
    - ✔ Set bit[*i*] = 1 in disk.
    - ✔ Allocate block[*i*]
    - ✔ Set bit[*i*] = 1 in memory

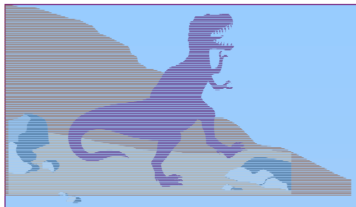




Linux File System {Chapter 20}

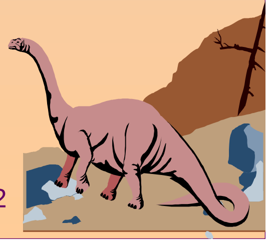
- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics.
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*.
- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - ✔ The *inode-object* and the *file-object* structures represent individual files
    - ✔ the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.



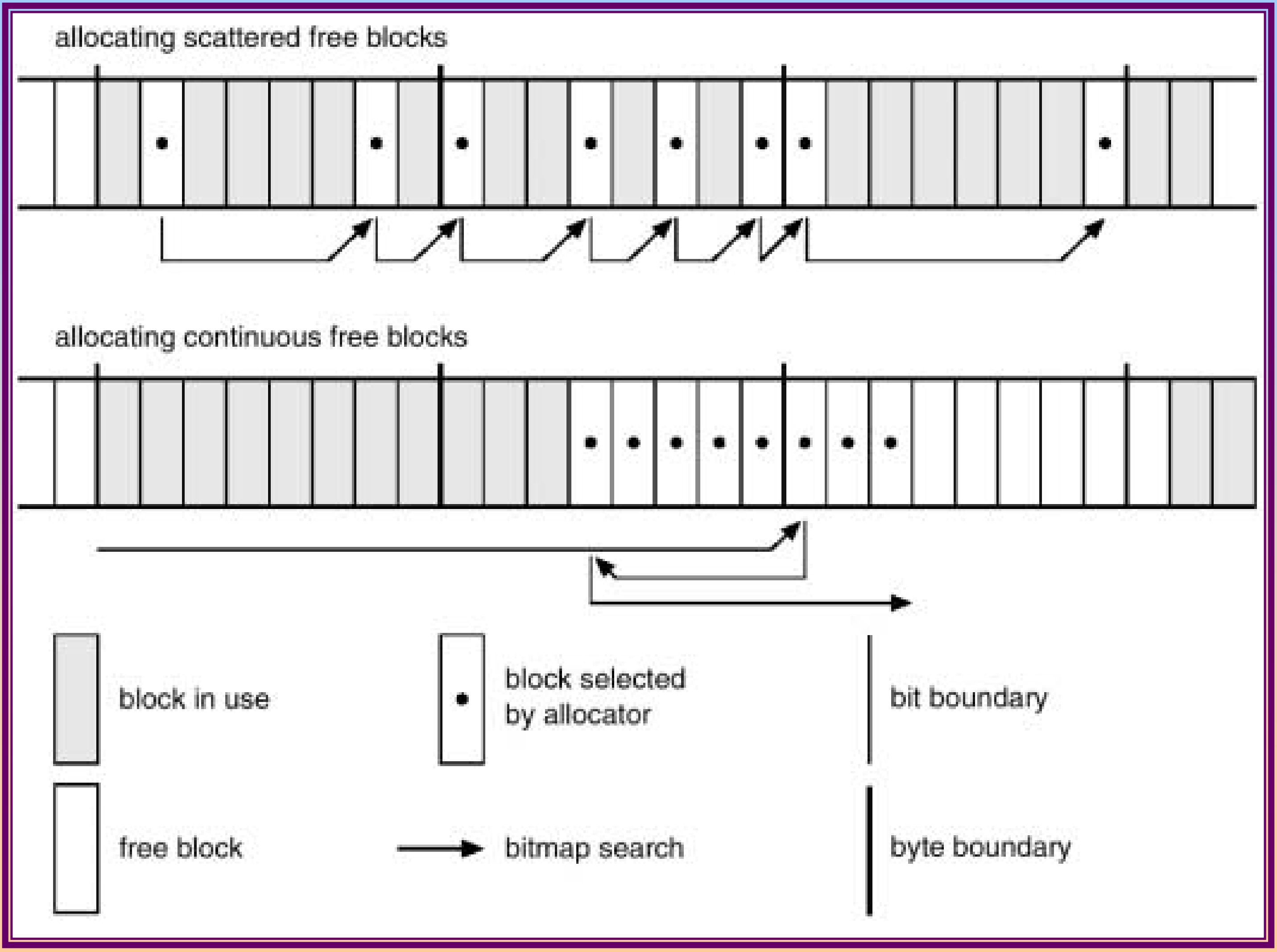


The Linux Ext2fs File System

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file.
- The main differences between ext2fs and ffs concern their disk allocation policies.
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.
  - Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

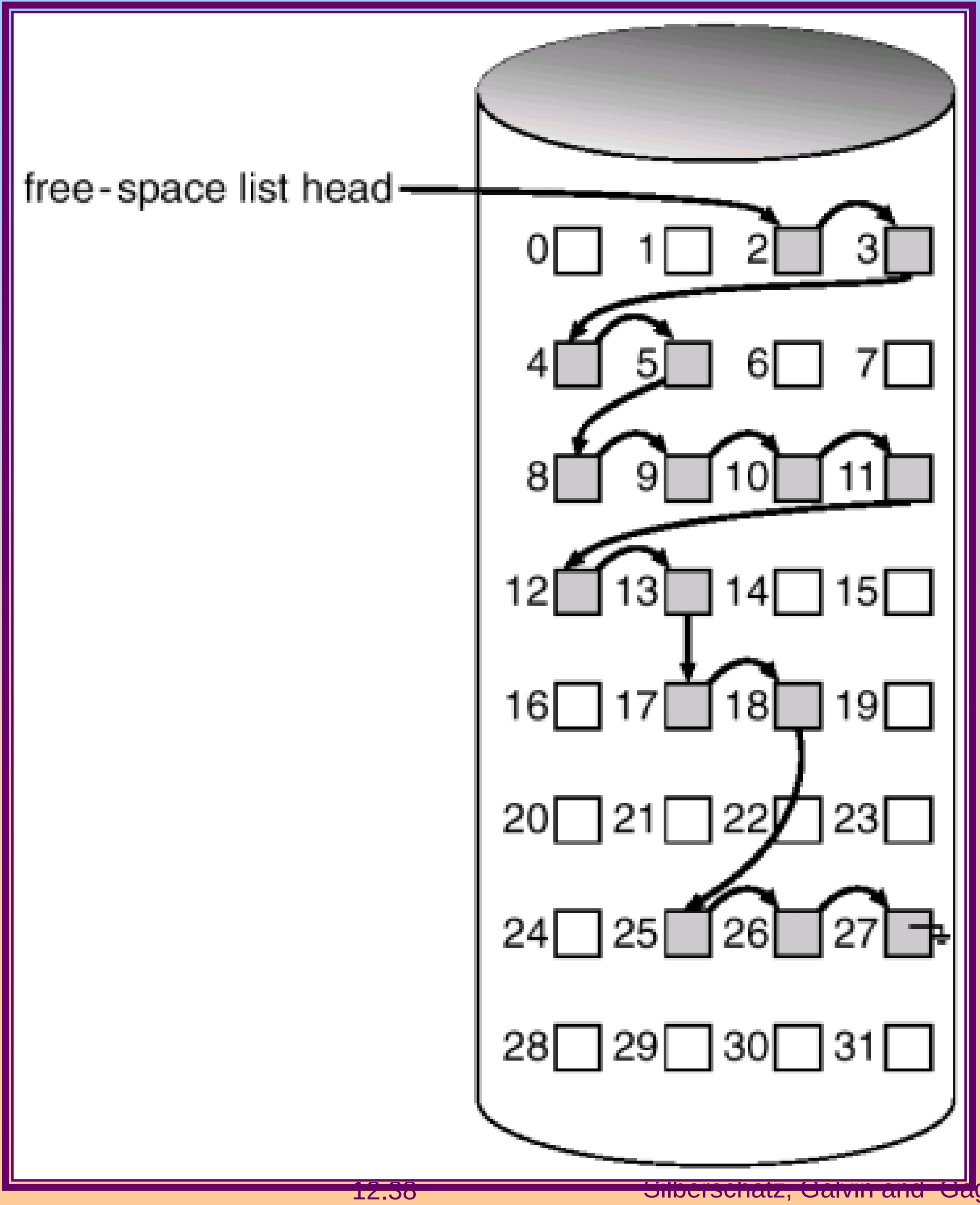


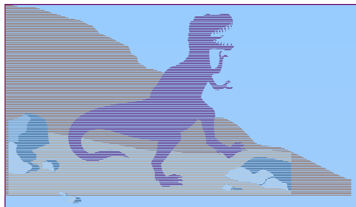
Ext2fs Block-Allocation Policies


allocating scattered free blocks
allocating continuous free blocks
block in use
block selected by allocator
bit boundary
free block
bitmap search
byte boundary

- Uses bitmap of free blocks
- Looks for free byte first
- Upon finding byte it backs up to avoid small holes.
- If no free byte, looks for bit
- It uses scattered allocation instead.

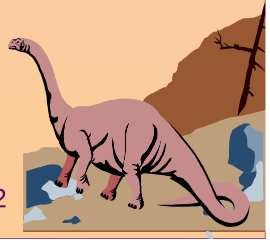
Linked Free Space List on Disk

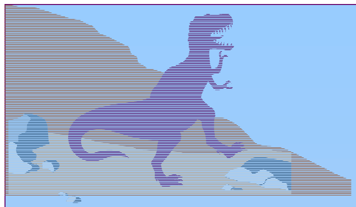




Efficiency and Performance

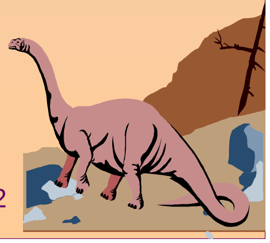
- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file’s directory entry
- Performance
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.



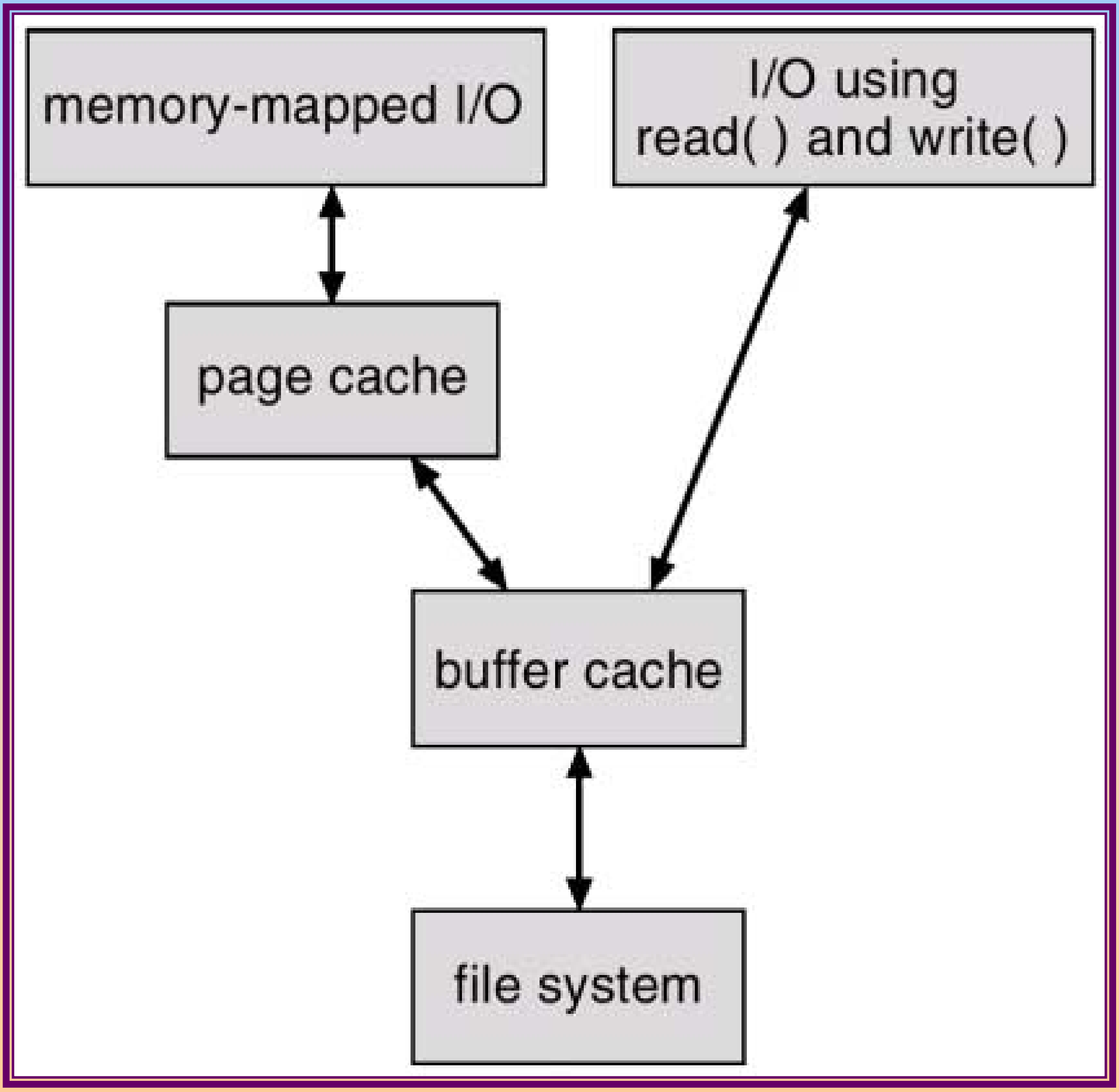


Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques.
- 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.



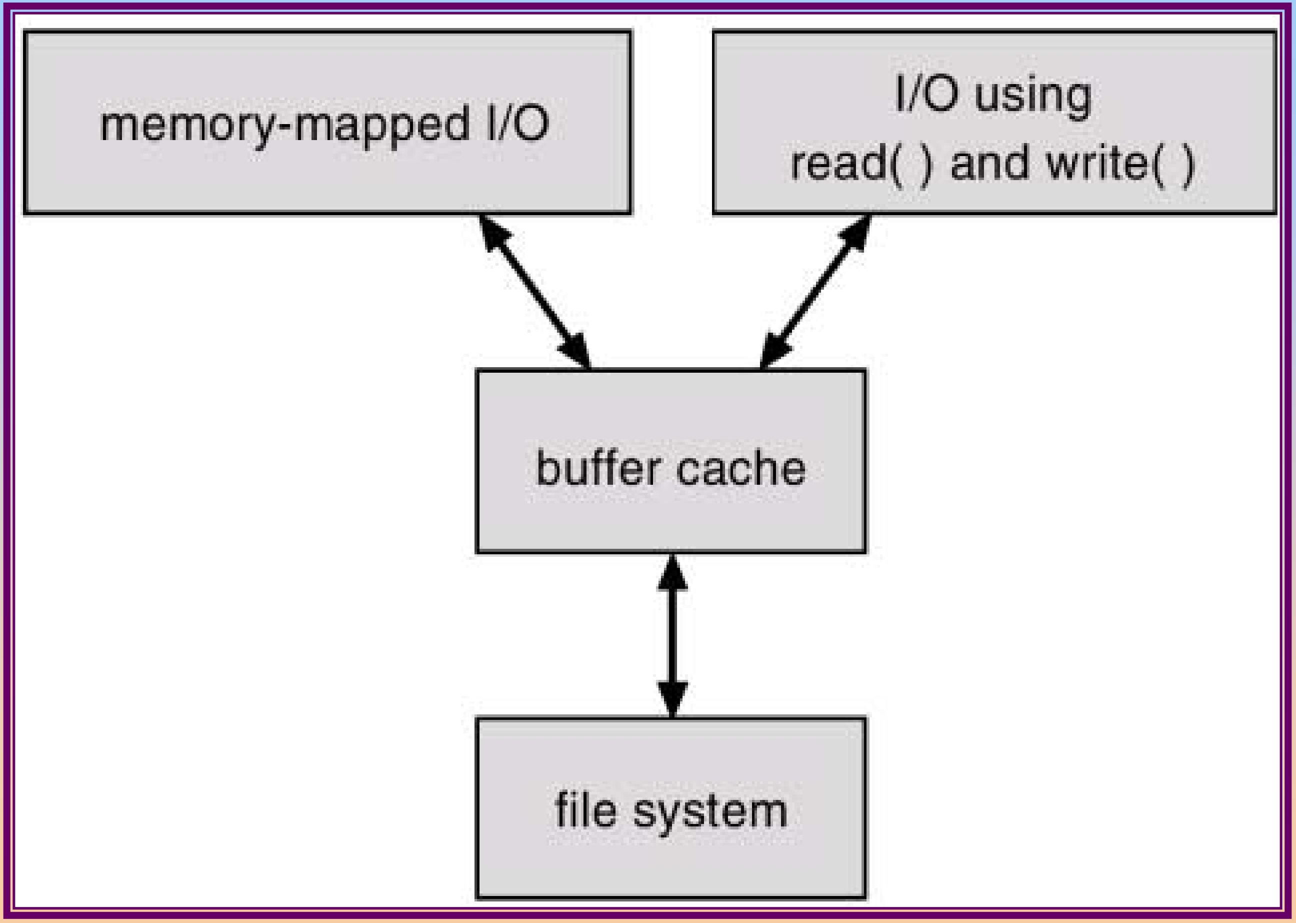
I/O Without a Unified Buffer Cache



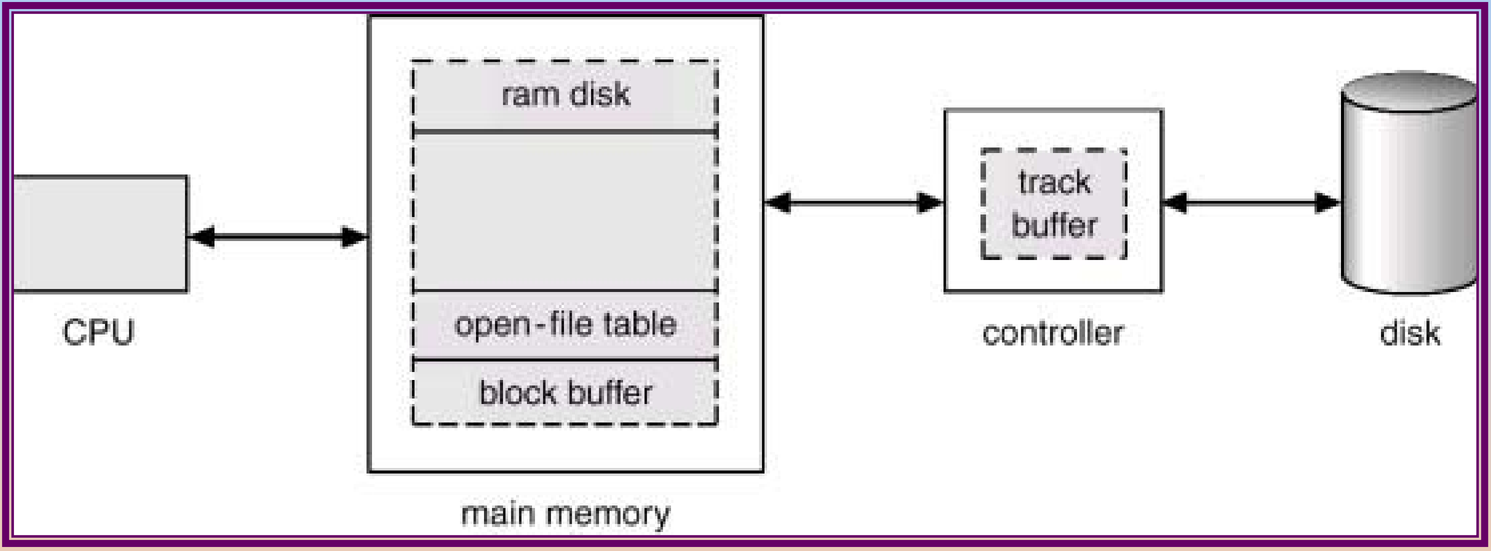
Unified Buffer Cache

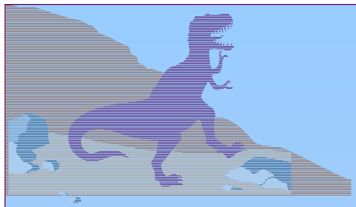
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.

I/O Using a Unified Buffer Cache



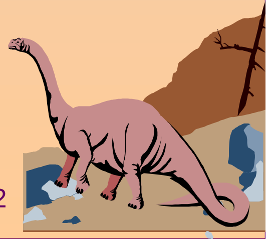
Various Disk-Caching Locations

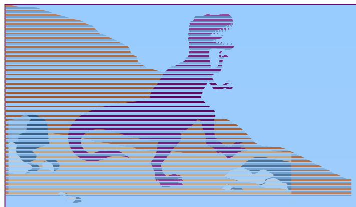




Recovery

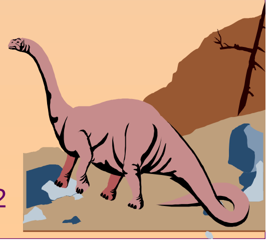
- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Use system programs to *back up* data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by *restoring* data from backup.

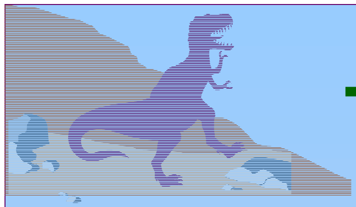




Log Structured File Systems

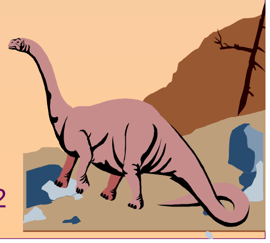
- **Log structured** (or journaling) file systems record each update to the file system as a **transaction**.
- All transactions are written to a **log**. A transaction is considered **committed** once it is written to the log. However, the file system may not yet be updated.
- The transactions in the log are asynchronously written to the file system. When the file system is modified, the transaction is removed from the log.
- If the file system crashes, all remaining transactions in the log must still be performed.

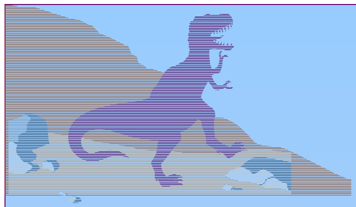




The Sun Network File System (NFS)

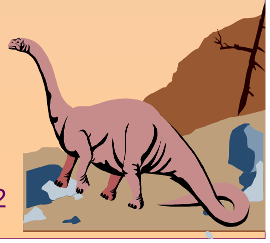
- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).
- The implementation is part of the Solaris and SunOS operating systems running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol and Ethernet.

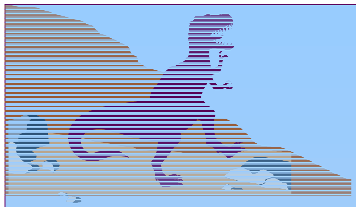




NFS (Cont.)

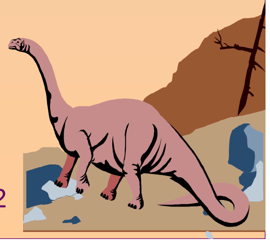
- Interconnected workstations viewed as a set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner.
  - A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the 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.



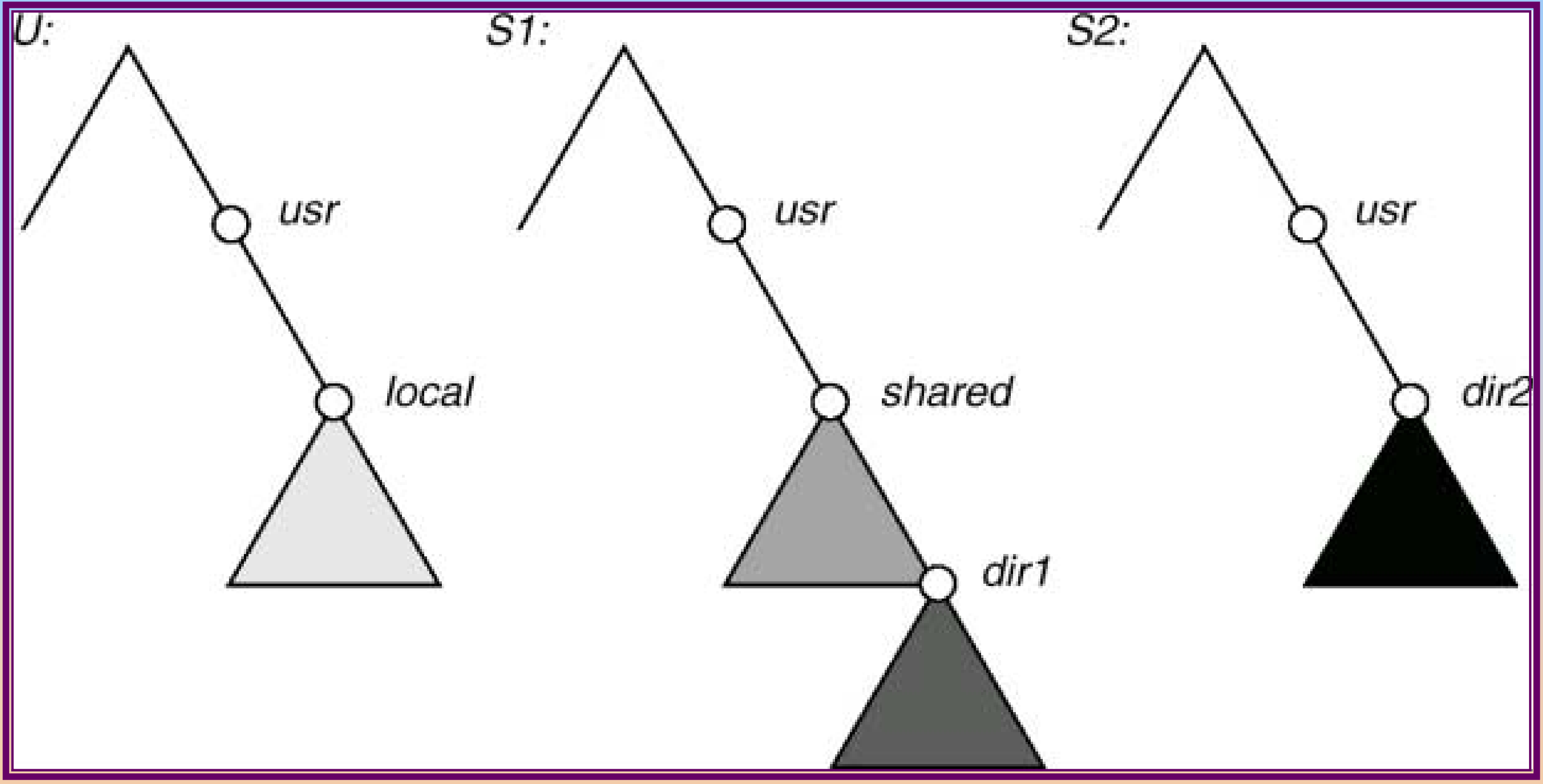


NFS (Cont.)

- 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.



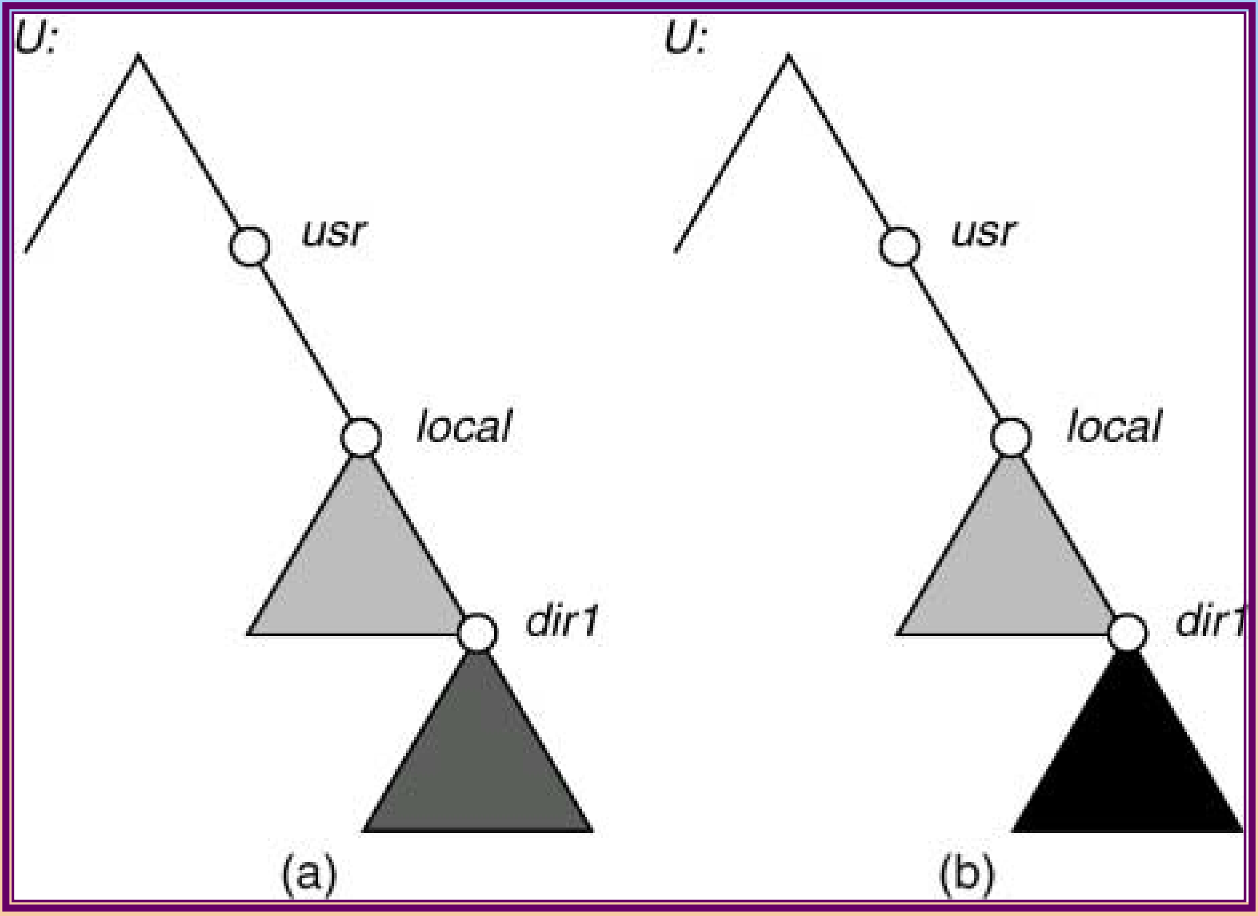
Three Independent File Systems



Mount S1:/usr/shared over U:/usr/local

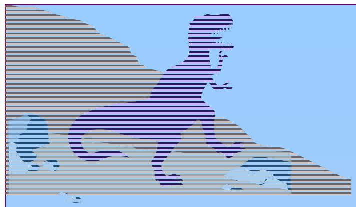
Cascaded Mount S2:/usr/dir2 over U:/usr/local/dir1

Mounting in NFS



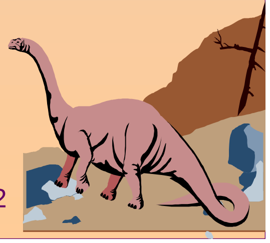
Mounts

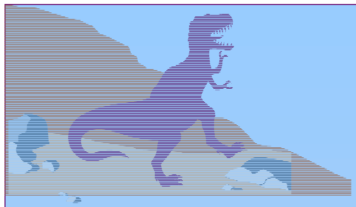
Cascading mounts



NFS Mount Protocol

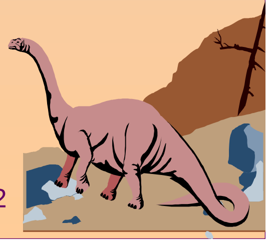
- Establishes initial logical connection between server and client.
- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine.
  - *Export list* – specifies 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 Protocol

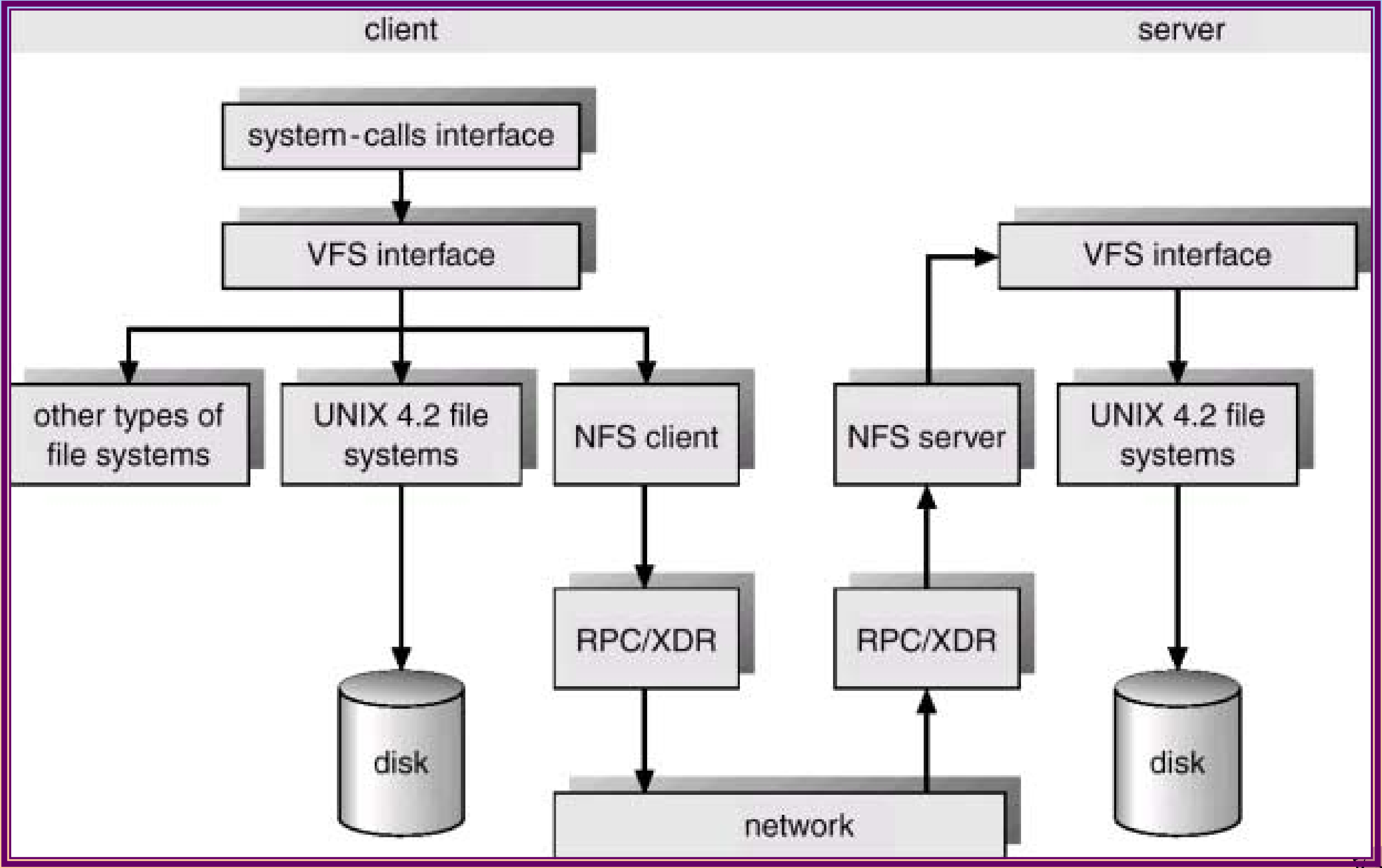
- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are *stateless*; each request has to provide a full set of arguments.
- 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.

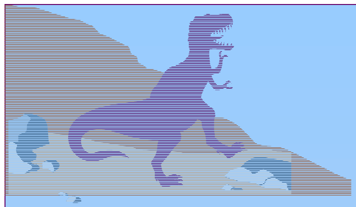


Three Major Layers of NFS Architecture

- 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.

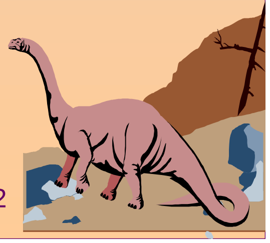
Schematic View of NFS Architecture

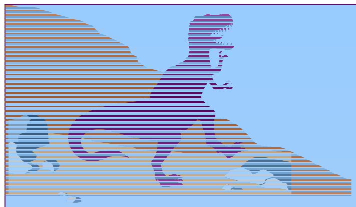




NFS Path-Name Translation

- 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.