

Chapter 11: File System Implementation

肖 卿 俊

办公室： 九龙湖校区计算机楼212室

电邮： `csqjxiao@seu.edu.cn`

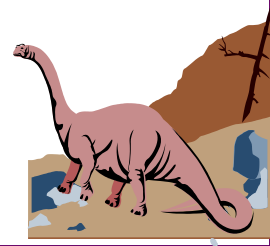
主页： <https://csqjxiao.github.io/PersonalPage>

电话： 025-52091022



Chapter 11: File System Implementation

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





File-System Structure

- In this chapter, “file” refers to either an ordinary file or a directory file
- File system resides on secondary storage (either local disks or remote disks).
- File structure information
 - ◆ Logical storage unit
 - ◆ Collection of related information
- *File control block (FCB)*
 - storage structure consisting of information about a file.

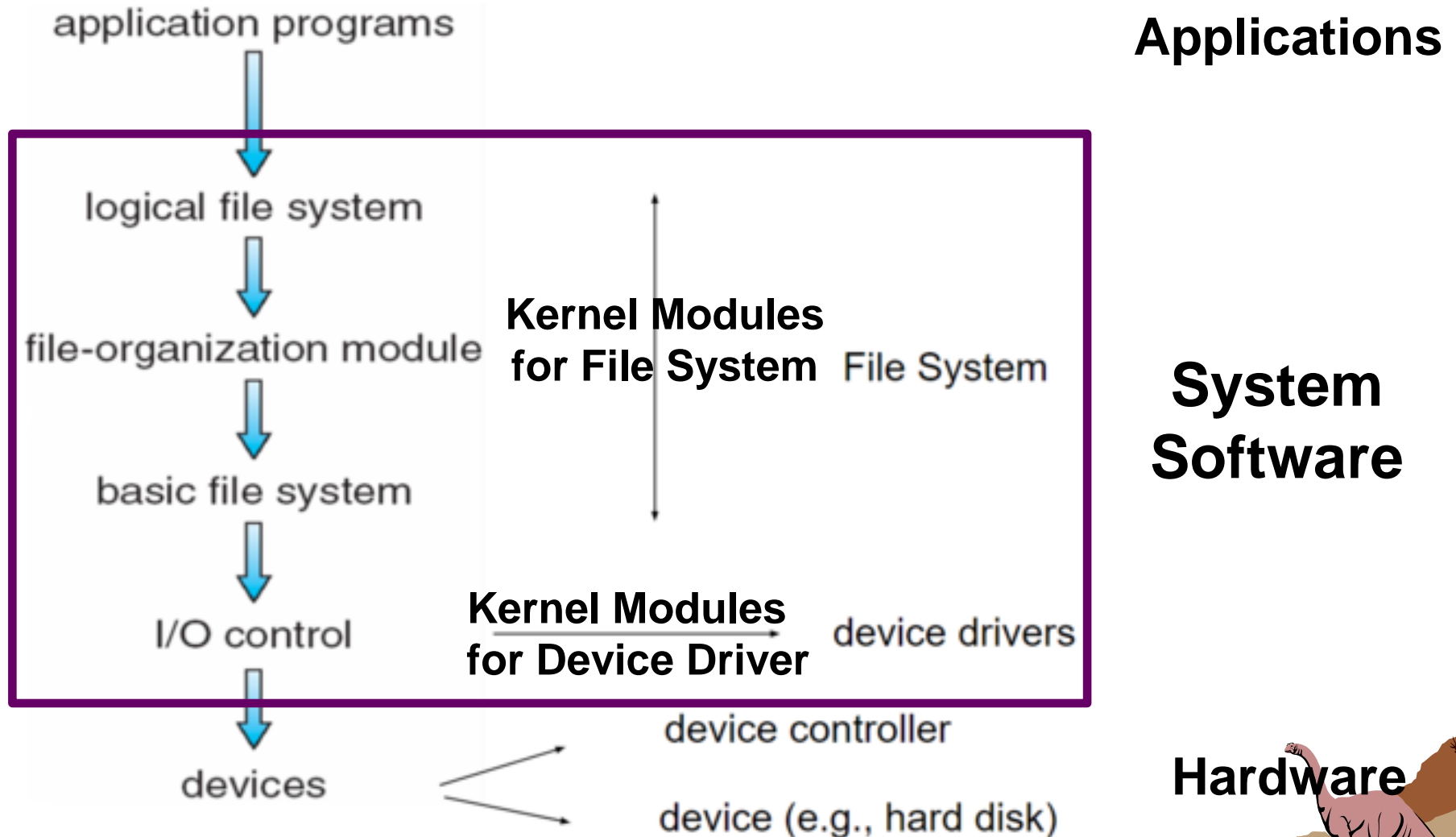
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks





Layered File System

File system is organized into layers

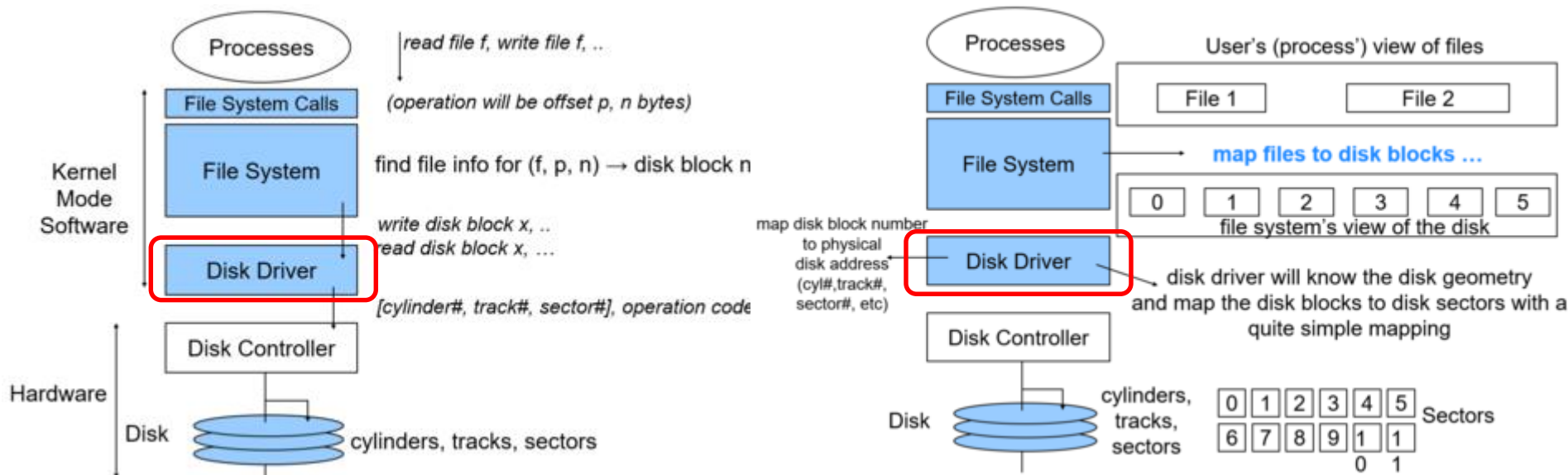




File System Layers

■ **Device drivers** manage I/O devices at the I/O control layer

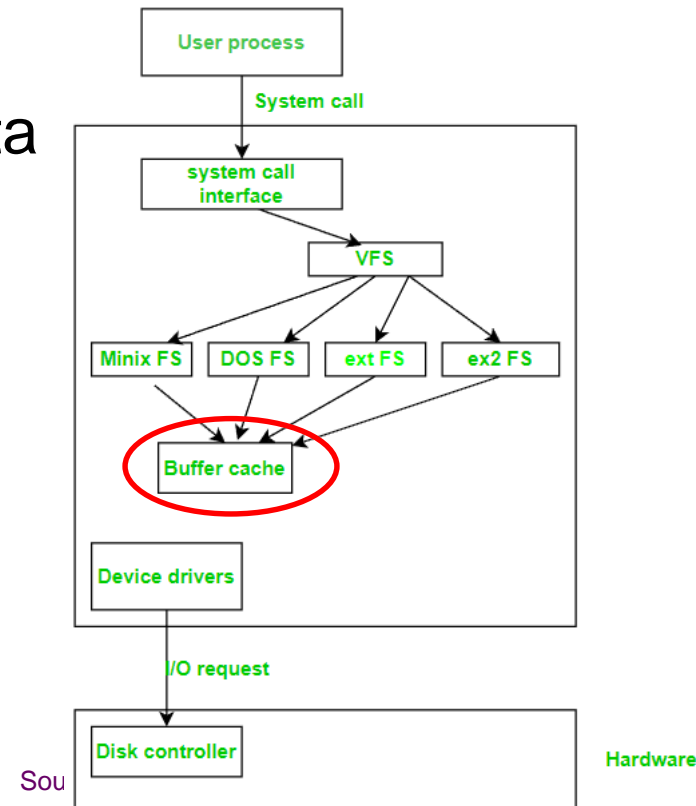
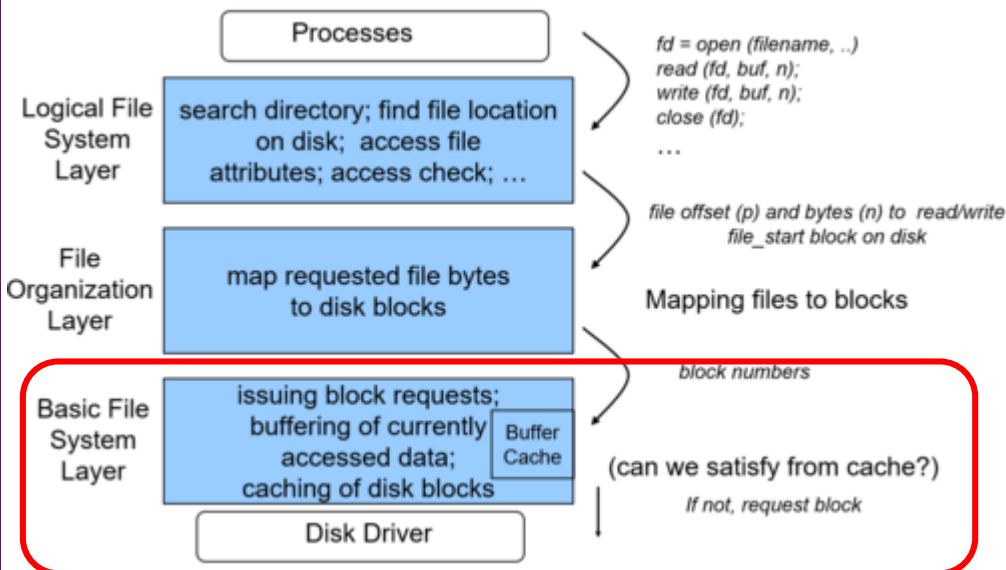
◆ Given commands “read/write disk block 587”, outputs low-level hardware specific commands to hardware controller, like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060”





File System Layers (Cont.)

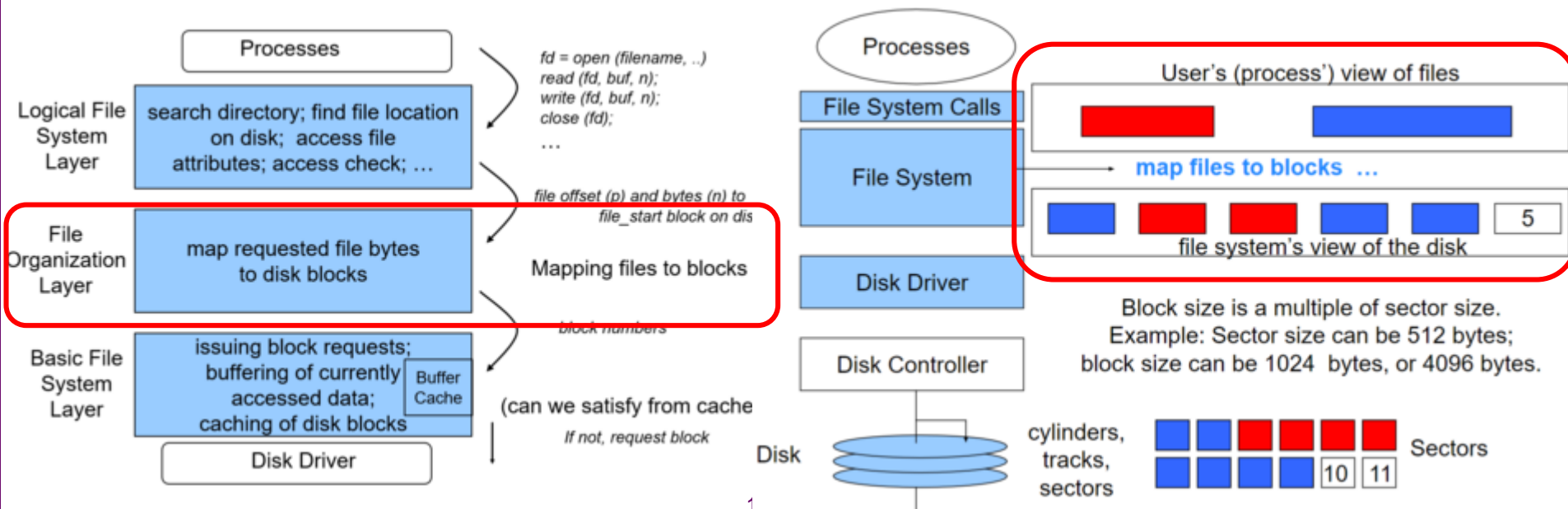
- **Basic file system** given command like “retrieve block 123” translates to device driver
- ◆ Also manages memory buffers and caches (allocation, freeing, replacement)
 - ✓ Buffers hold data in transit
 - ✓ Caches hold frequently used data





File System Layers (Cont.)

- **File organization module** understands files, logical address, and physical blocks
 - ◆ Translates logical block # to physical block #
 - ◆ Manages free space, disk allocation





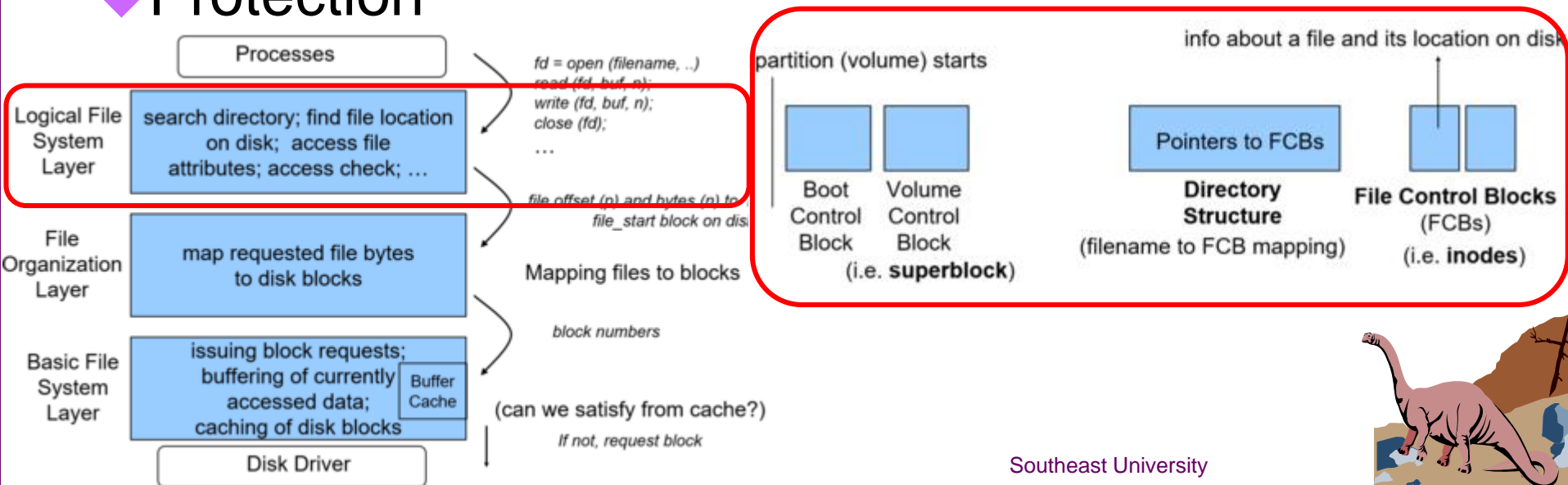
File System Layers (Cont.)

■ **Logical file system** manages metadata information

◆ Translates file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)

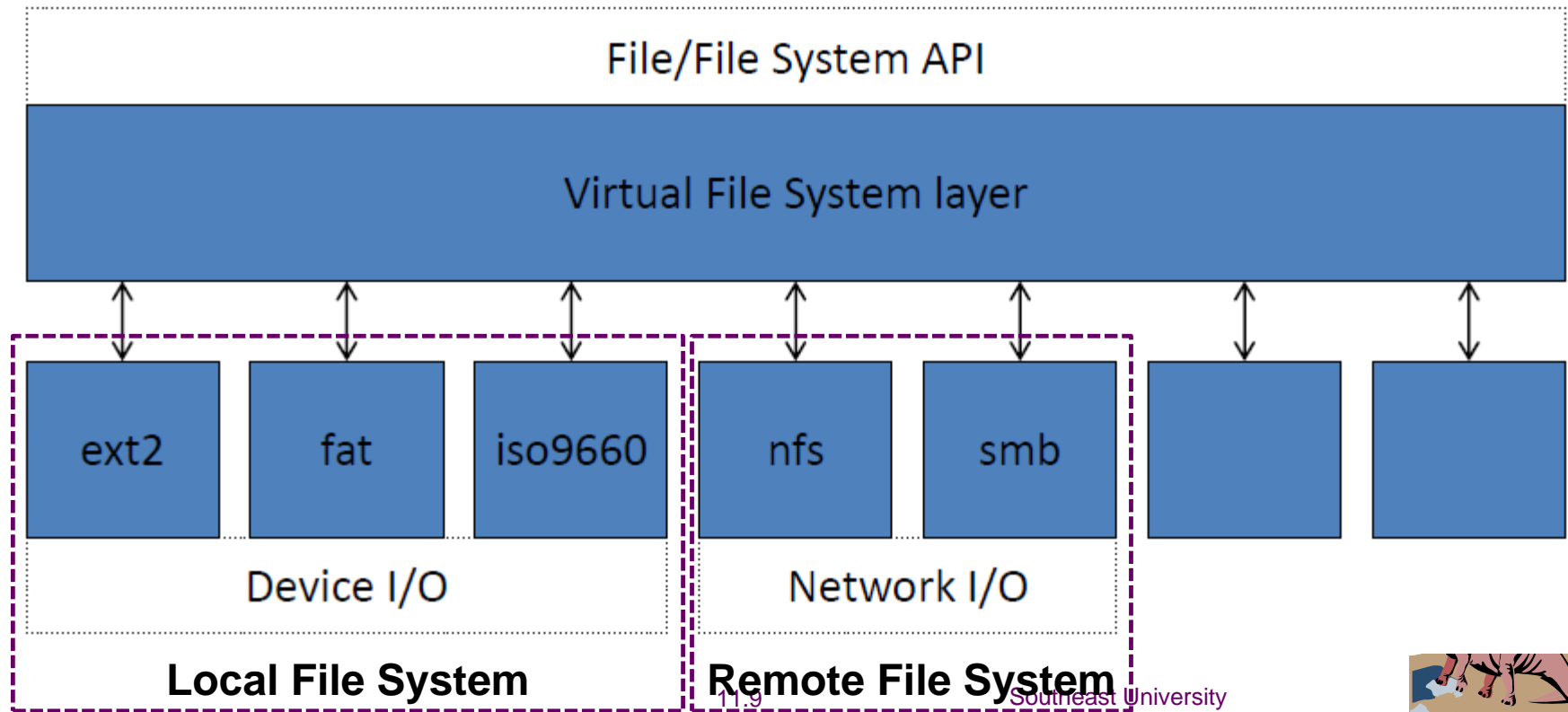
◆ Directory management

◆ Protection



Each OS with its own supported file system format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray; Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc)

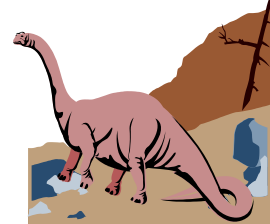
New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE





Chapter 11: File System Implementation

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





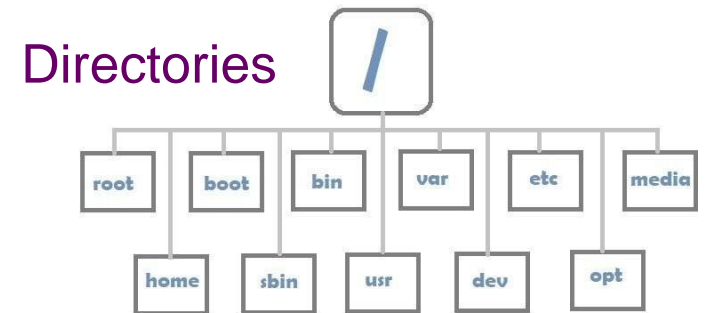
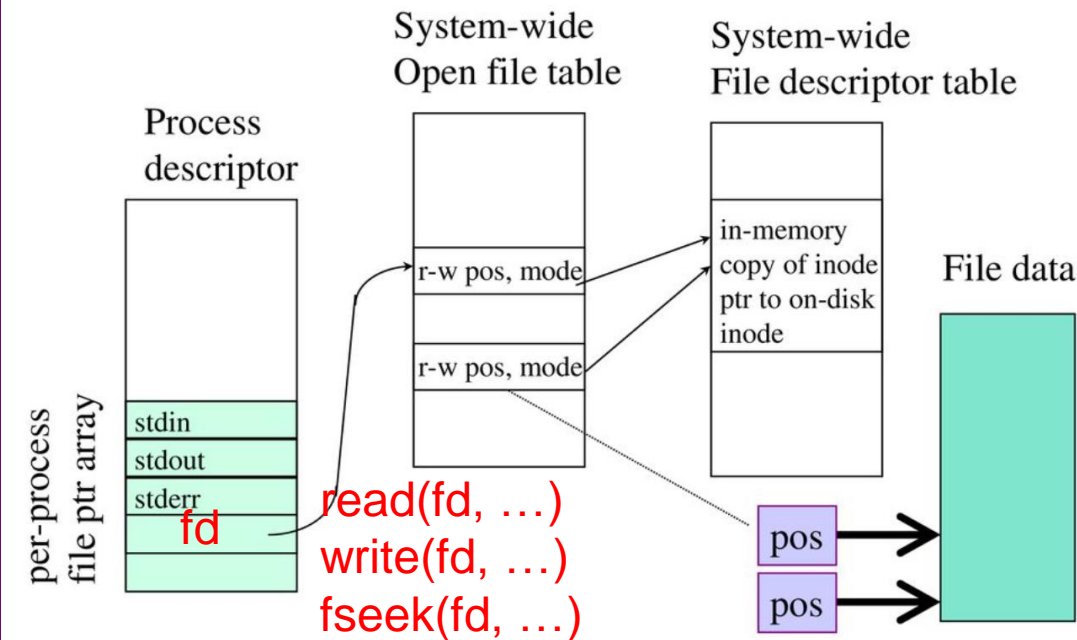
Two kinds of Structures of File System

■ We have file system calls at the **API level**, but how do we implement their functions?

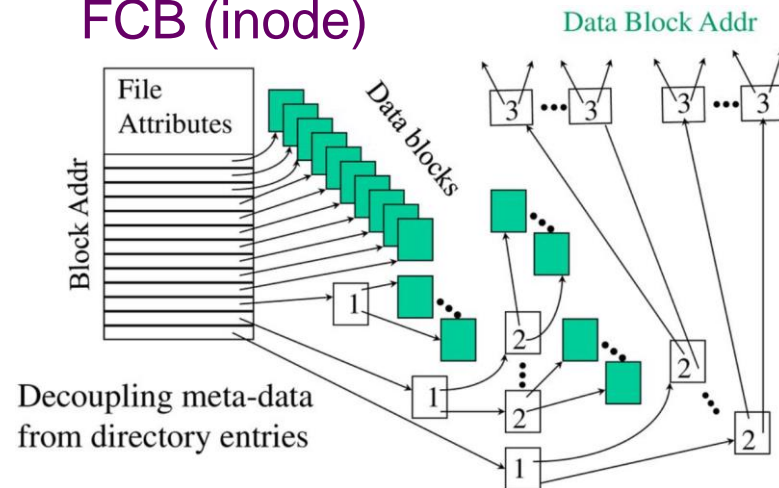
◆ In-memory and on-disk structures

In-memory data structure

On-disk data structure



FCB (inode)

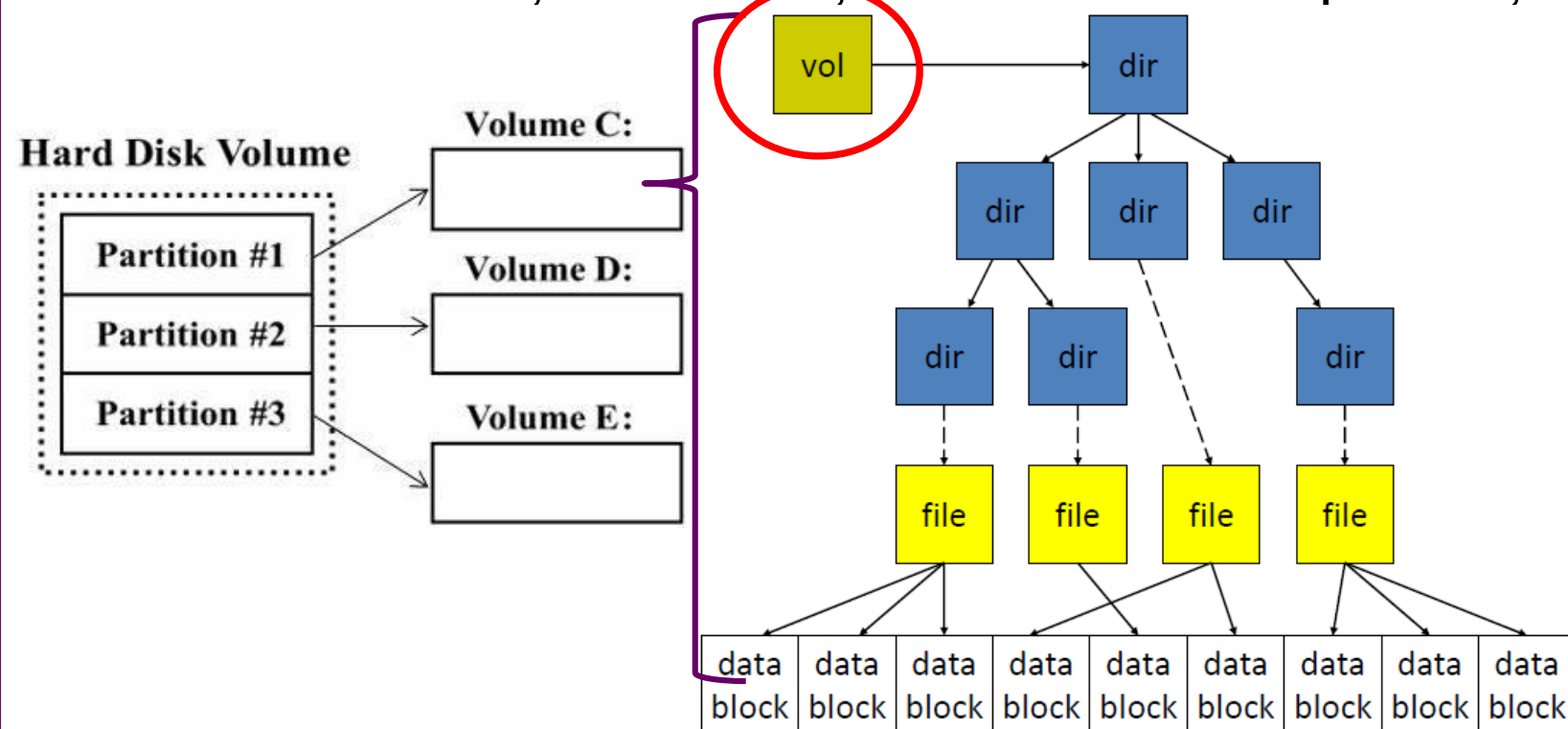




On-disk Structures of File System

■ Volume Control Block (Unix: "superblock")

- ◆ One per file system
- ◆ Detail information about the file system
- ◆ # of blocks, block size, free-block count/pointer, etc.



Disk





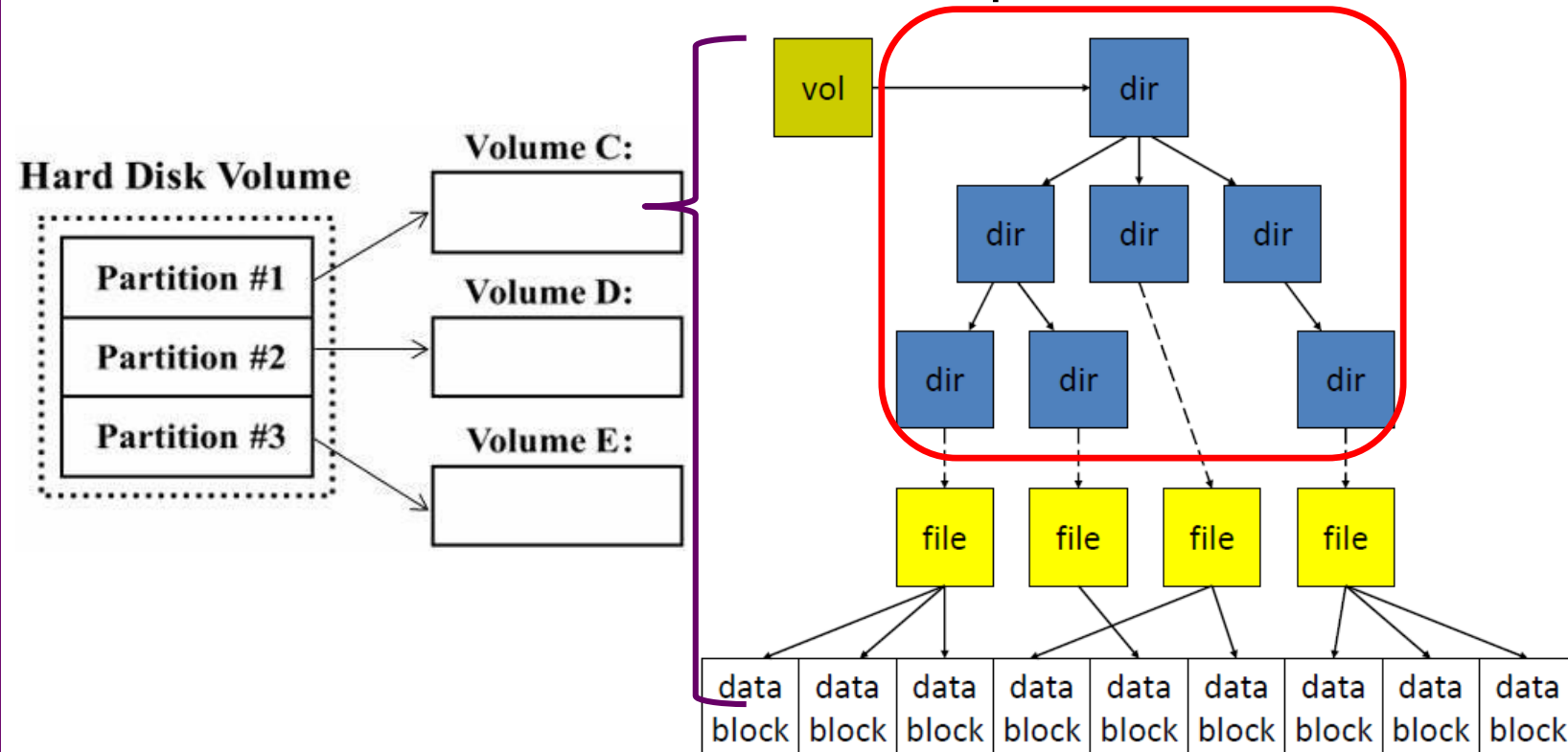
- ◆ One per file to provide detailed information about the file
- ◆ Permission, owner, size, data block locations, etc.





■ Directory Node (Linux: "dentry")

- ◆ One per directory entry (directory or file)
- ◆ Pointer to file control block, parent, list of entries, etc.



Disk





In-Memory Structures of File System

- The following figure illustrates the necessary file system structures provided by the operating systems.
- Figure (a)/(b) refers to opening/reading a file

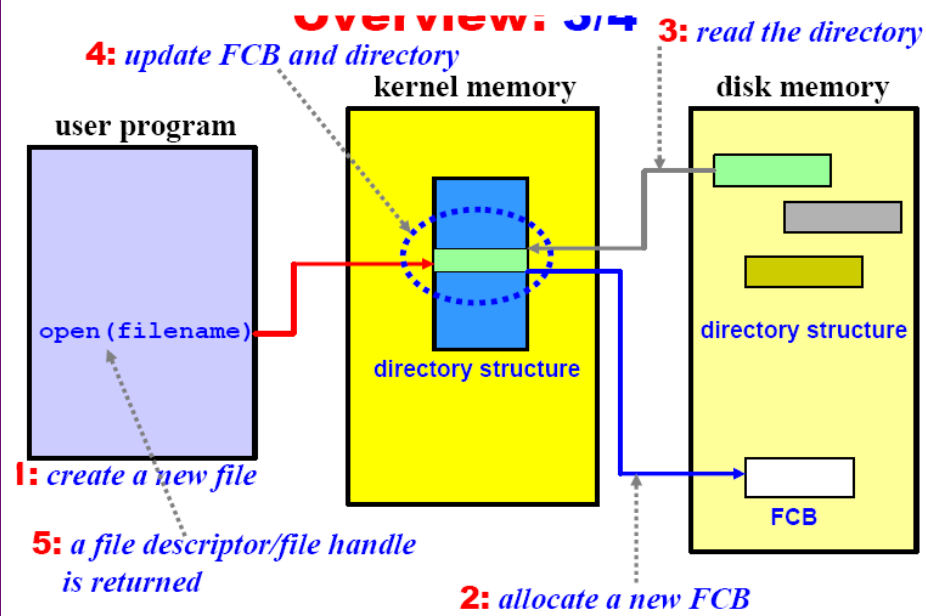


Figure (a)

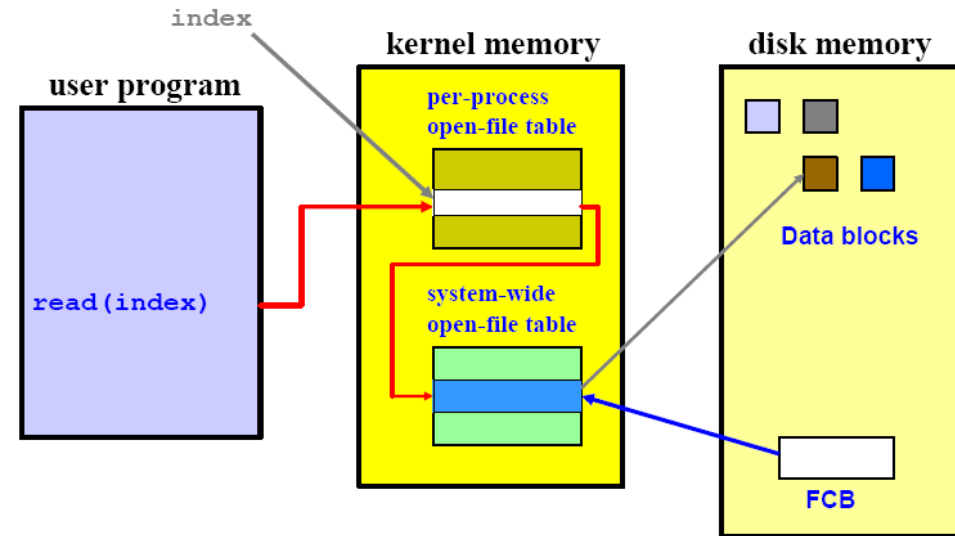
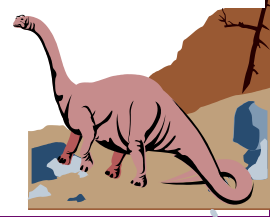


Figure (b)



In-Memory Structures of File System (cont)

- ◆ 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

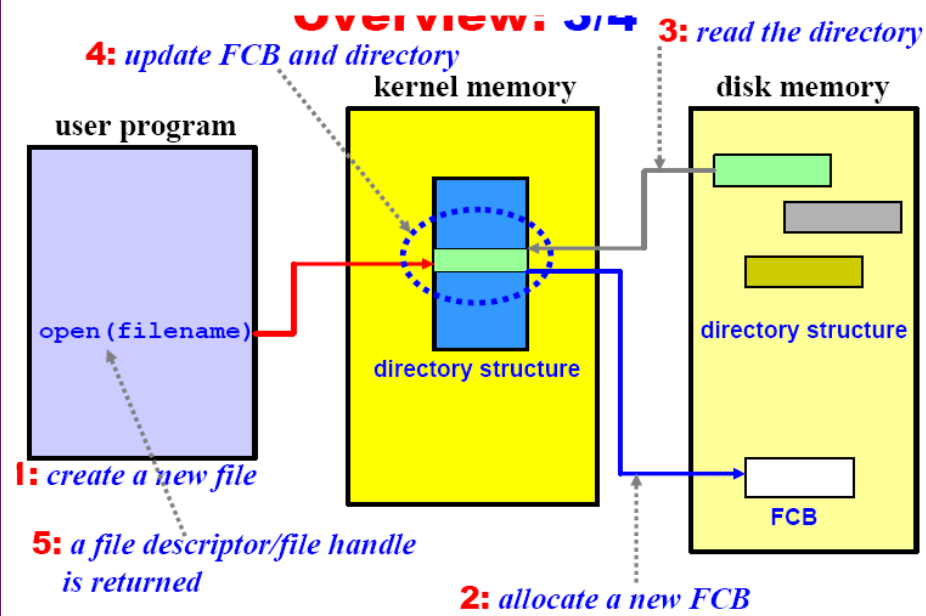


Figure (a)

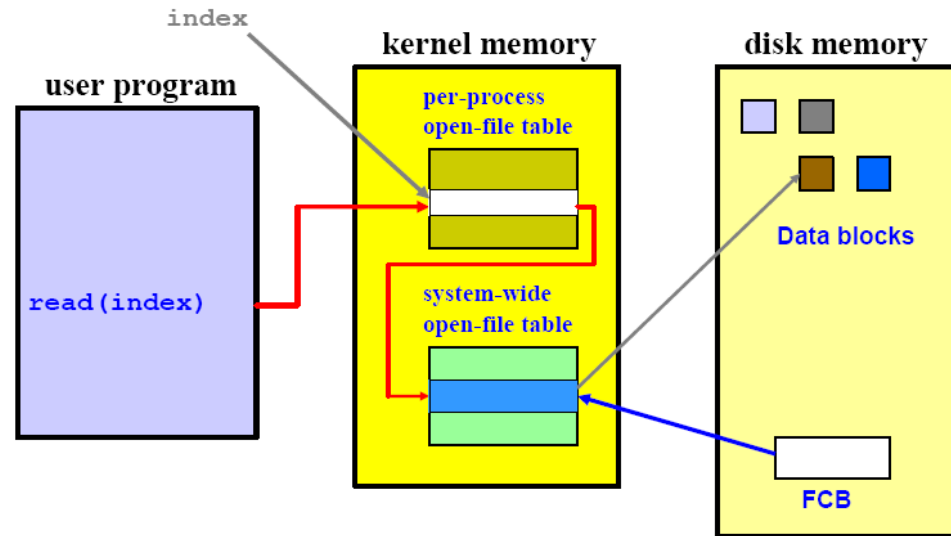
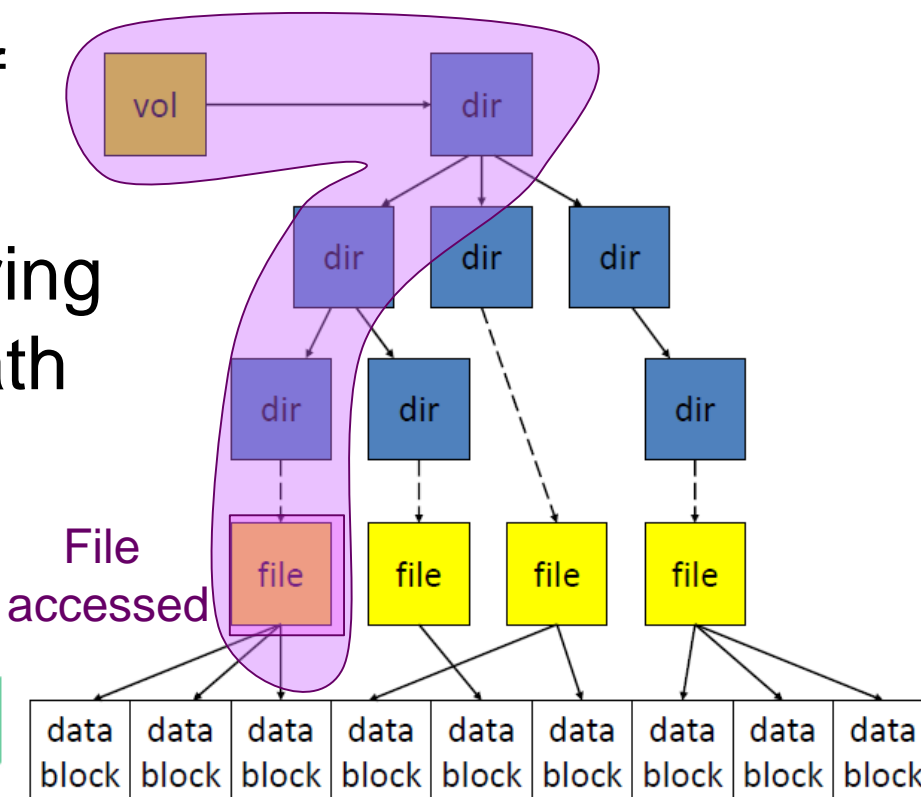
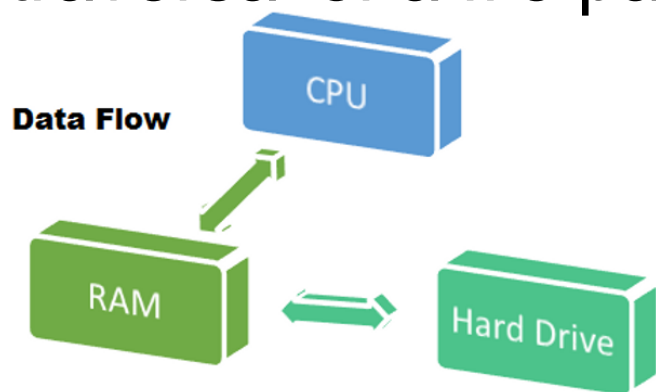


Figure (b)



On-demand Loading of On-disk Structures into Main Memory

- Loaded to memory when needed
 - ◆ Volume control block: in memory if file system is mounted
 - ◆ File control block: if the file is accessed
 - ◆ Directory node: during traversal of a file path



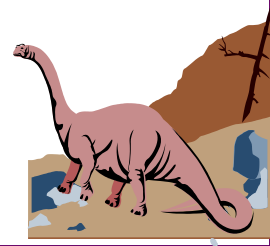
Disk





Chapter 11: File System Implementation

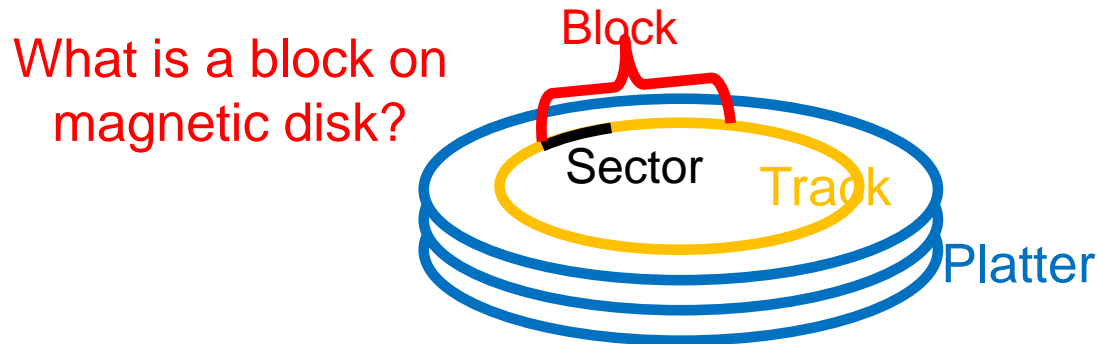
- File System Structure
- File System Implementation
- **Free-Space Management**
- Directory Implementation
- Allocation Methods
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS





Free-Space Management

- How do we keep track of free blocks on a disk?



- The techniques below are commonly used:
 - ◆ Bit Vector or Bit Map
 - ◆ Linked List: A free-list is maintained. When a new block is requested, we search this list to find one.
 - ◆ Linked List + Grouping
 - ◆ Linked List + Address + Count





Bit Vector

■ Bit vector (n blocks)



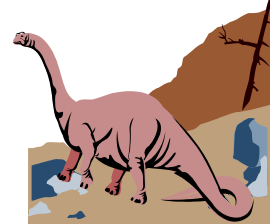
$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

CPUs have instructions to return offset within word of first “1” bit

The first free block number calculation:

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

Question: What the time cost of finding the number of 0-value words? Why it doesn't matter?





Free-Space Management

- Advantage of bit vector method: Easy to get contiguous files

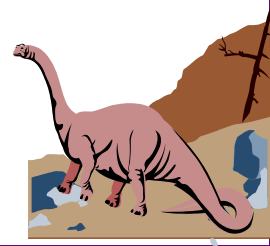
- Disadvantage: Bitmap requires extra space.

- An Example:

block size = 2^{12} bytes

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

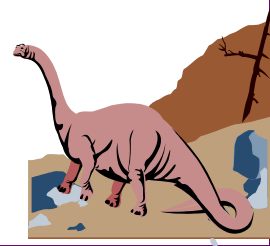
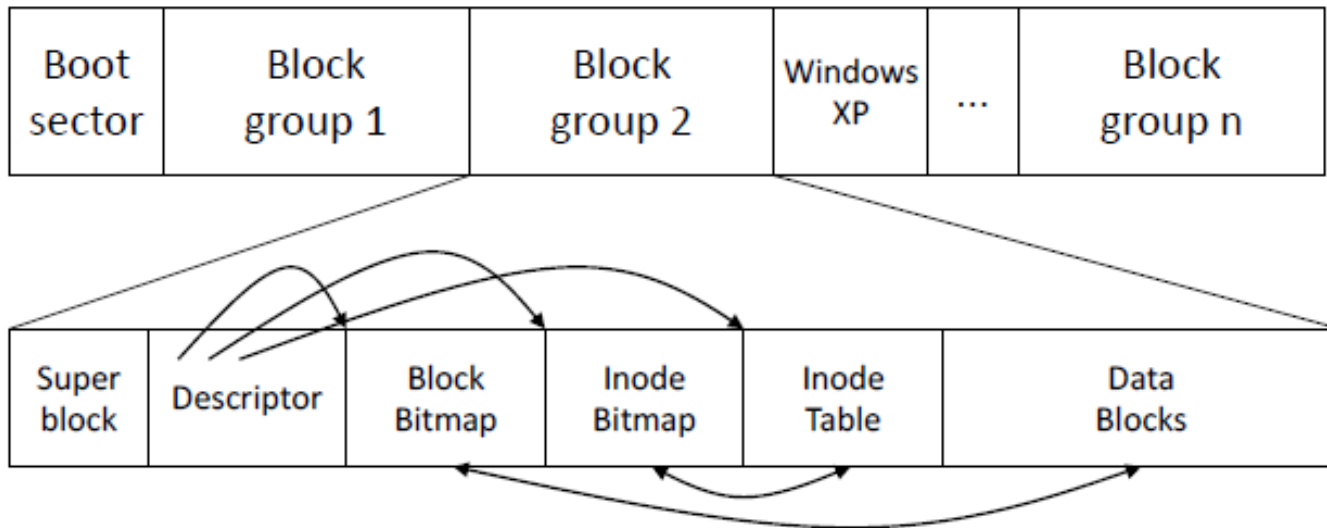
$n = 2^{40}/2^{12} = 2^{28}$ bits (or 32 mega bytes)





Linux Ext2 Disk Layout

- Block bitmap is used by Linux Ext2 to manage the disk free space.

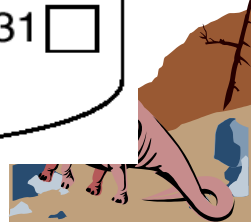
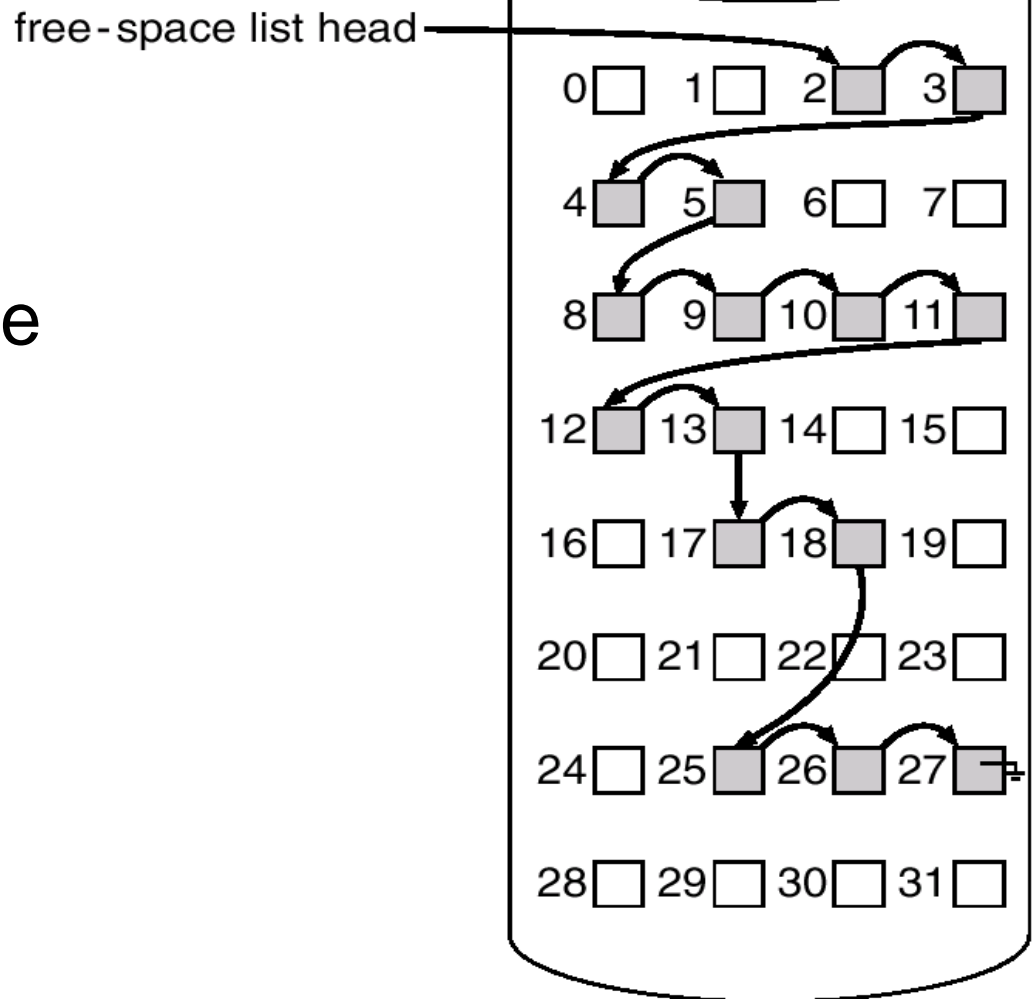




Linked Free Space List on Disk

■ Linked list (free list)

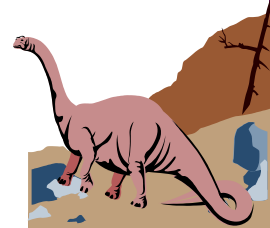
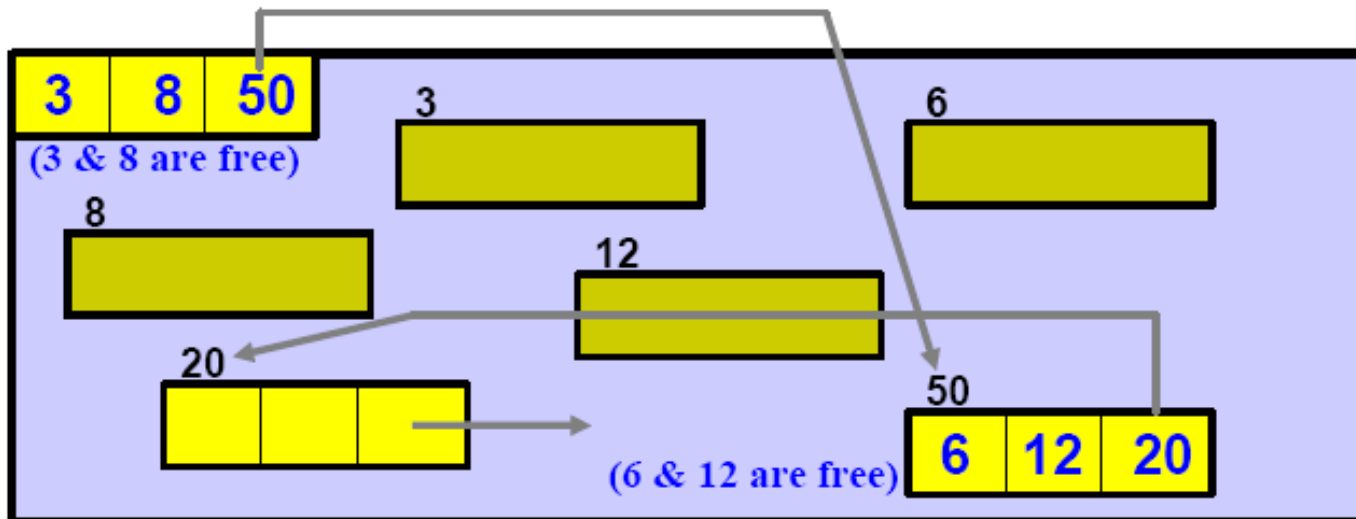
- ◆ Cannot get contiguous space easily
- ◆ No waste of space





Grouping of Multiple Free Blocks

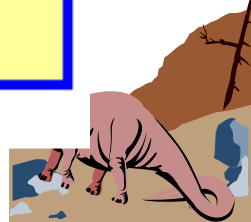
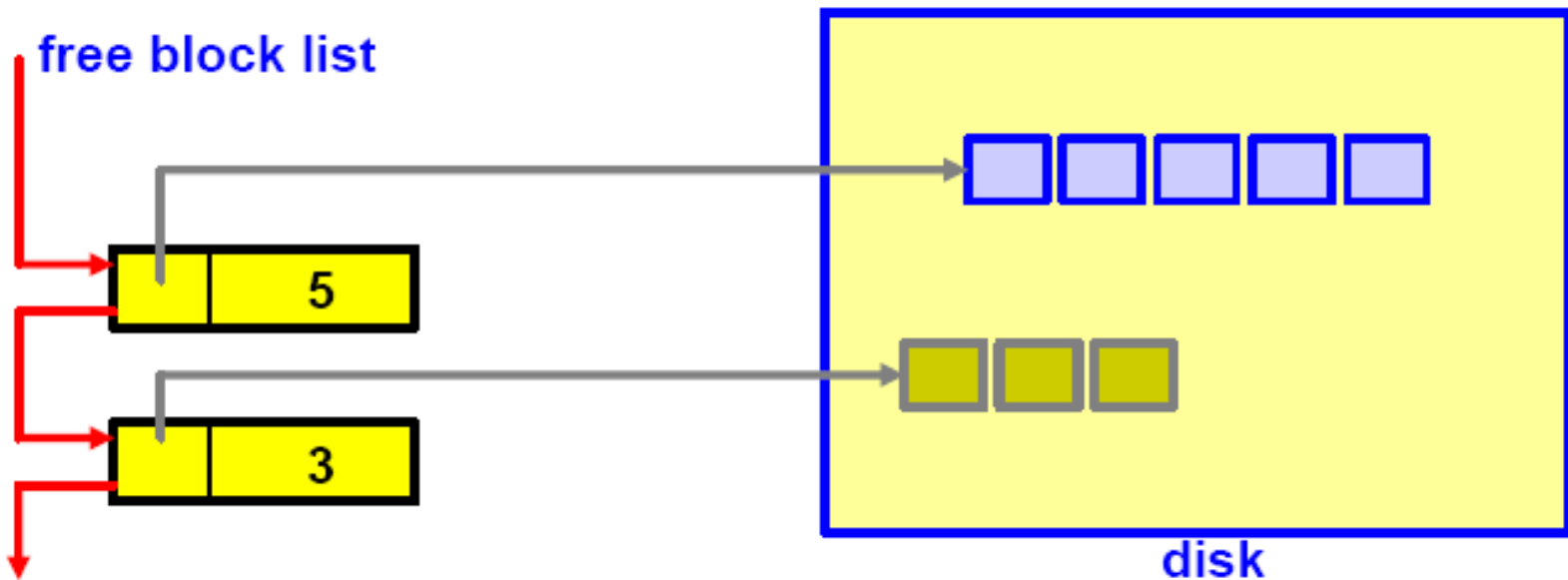
- The first free block contains the addresses of n other free blocks.
- For each group, the first $n-1$ blocks are actually free and the last (i.e., n -th) block contains the addresses of the next group.
- In this way, we can quickly locate free blocks.





Address Counting of Contiguous Free Blocks

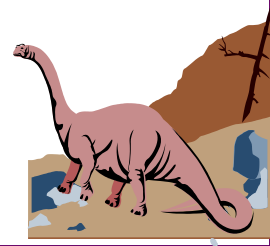
- We can make the list short with the following trick:
 - ◆ Blocks are often allocated and freed in groups
 - ◆ We can store the address of the first free block and the number of the following n free blocks.





Chapter 11: File System Implementation

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

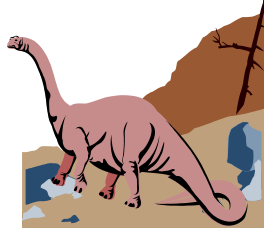


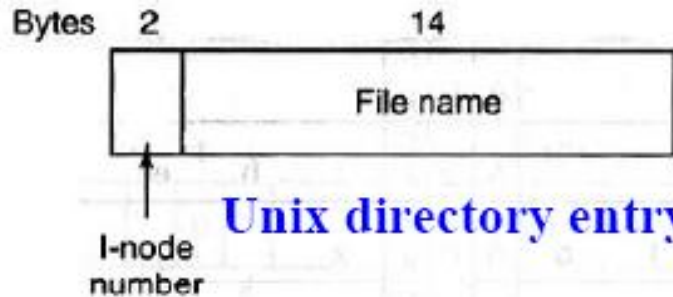


Directory Implementation

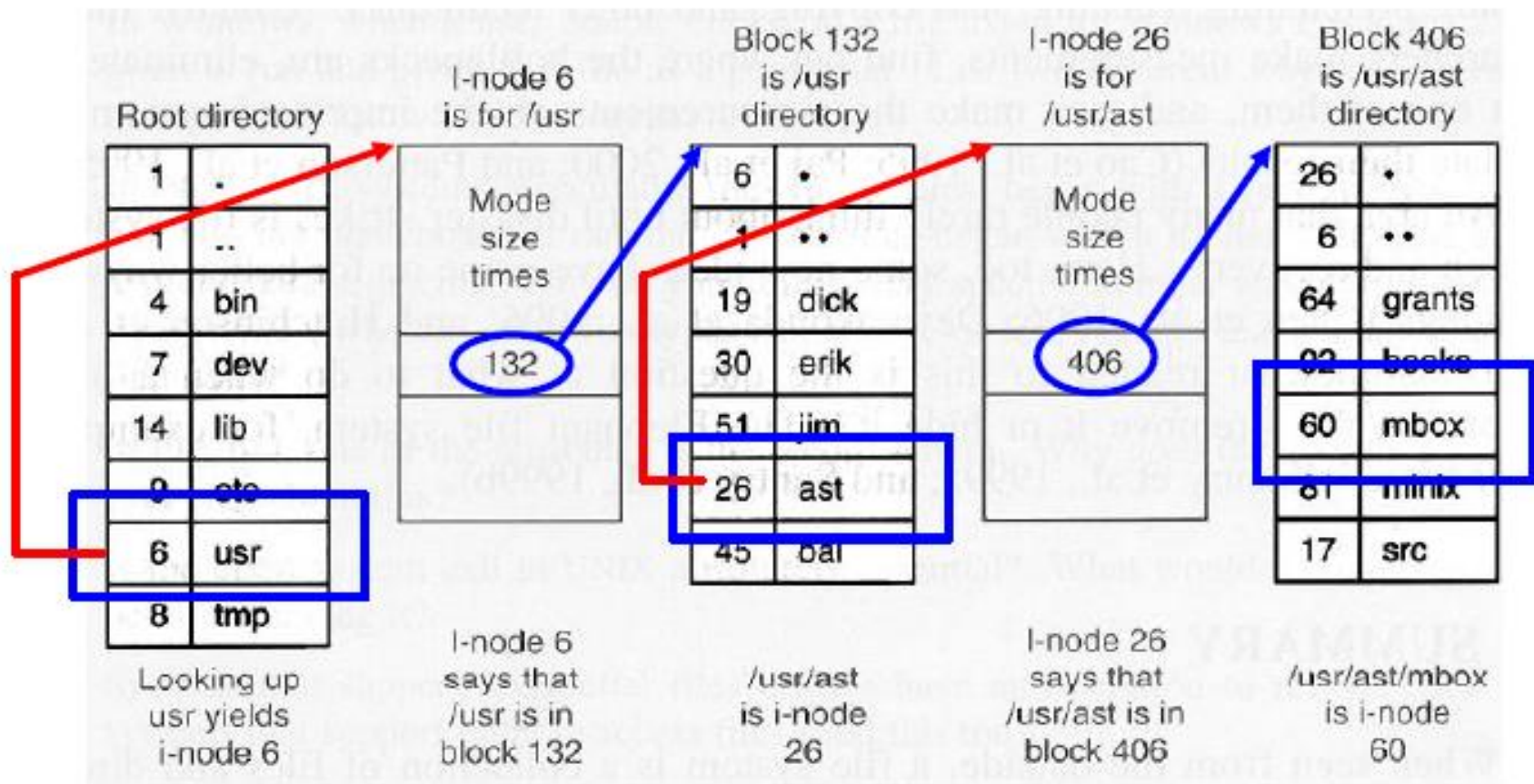
- Linear list of file names with pointer to the data blocks.
 - ◆ simple to program
 - ◆ time-consuming to execute

- Hash Table – linear list with hash data structure.
 - ◆ decreases directory search time
 - ◆ *collisions* – situations where two file names hash to the same location





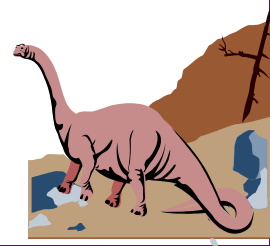
find /usr/ast/mbox





Chapter 11: File System Implementation

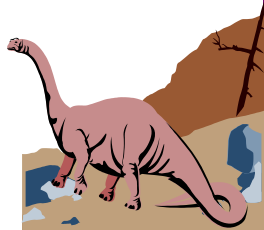
- File System Structure
- File System Implementation
- Free-Space Management
- Directory Implementation
- Allocation Methods
- Efficiency and Performance
- Recovery
- Log-Structured File Systems
- NFS





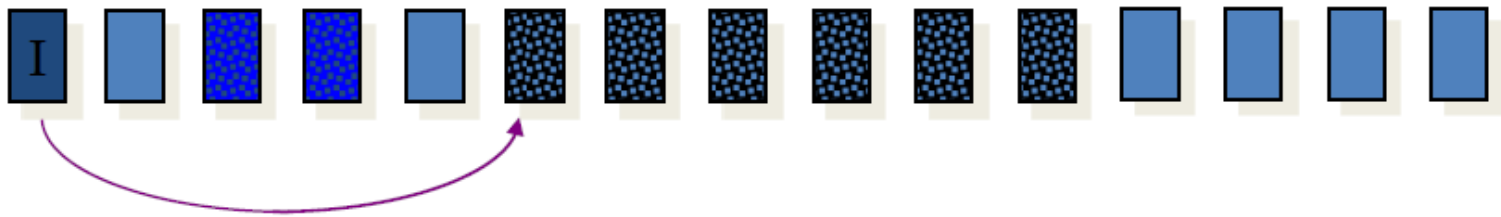
File Allocation Methods

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



Contiguous Allocation of Disk Space

- Each file occupies a set of contiguous blocks on the disk.



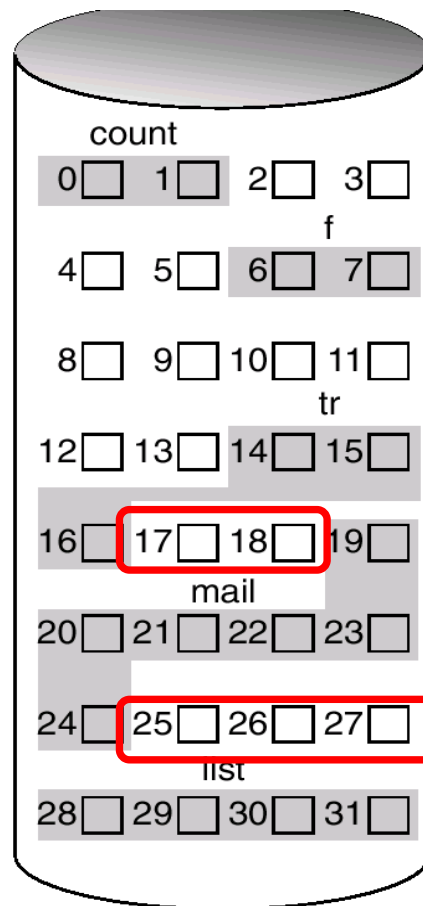
- Advantages:
 - Simple – only starting location (block #) and length (number of blocks) are required.
 - Random access.



Contiguous Allocation of Disk Space (con

■ Disadvantages

- Wasteful of space (recall the dynamic storage-allocation problem and external fragmentation).
- Files may not be able to grow.



directory

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

External fragmentation happens when a **dynamic space allocation method** allocates some disk spaces but leaves a **small amount of spaces** unusable.

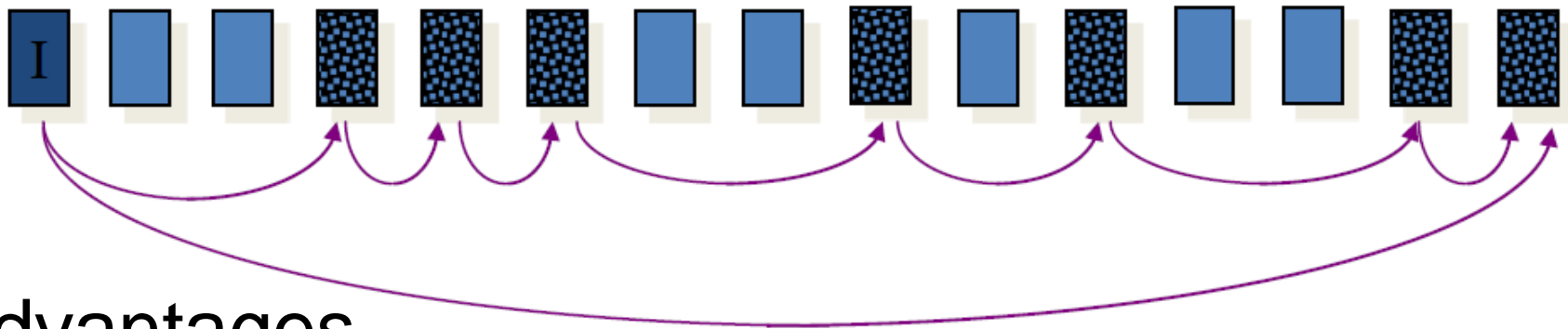
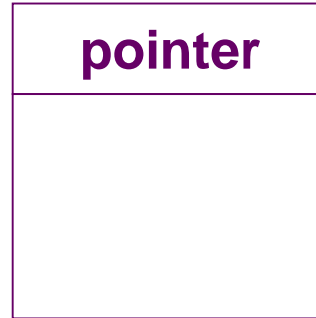




Linked Allocation

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

block = pointer



Advantages

- Simple – need only starting address
- Free-space management system – no waste of space
- Files can easily grow, if there are free blocks

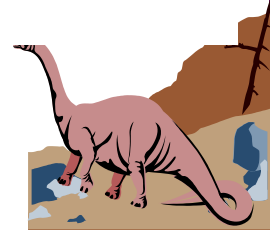
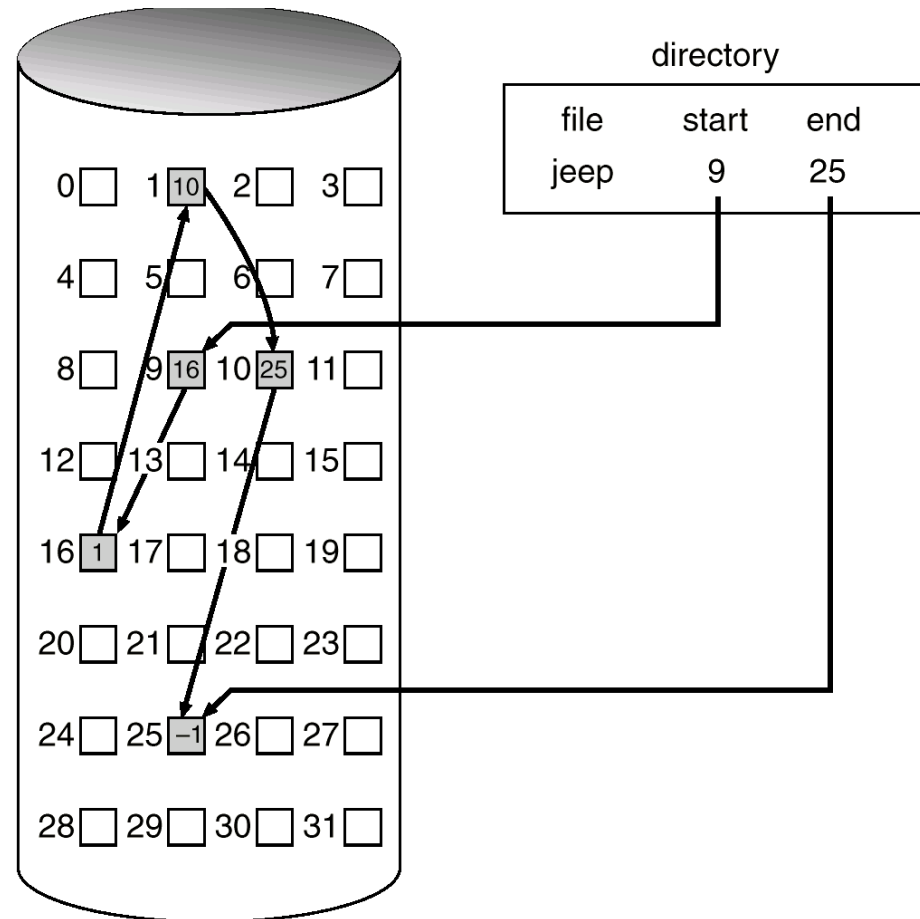




Linked Allocation (Cont.)

■ Disadvantages:

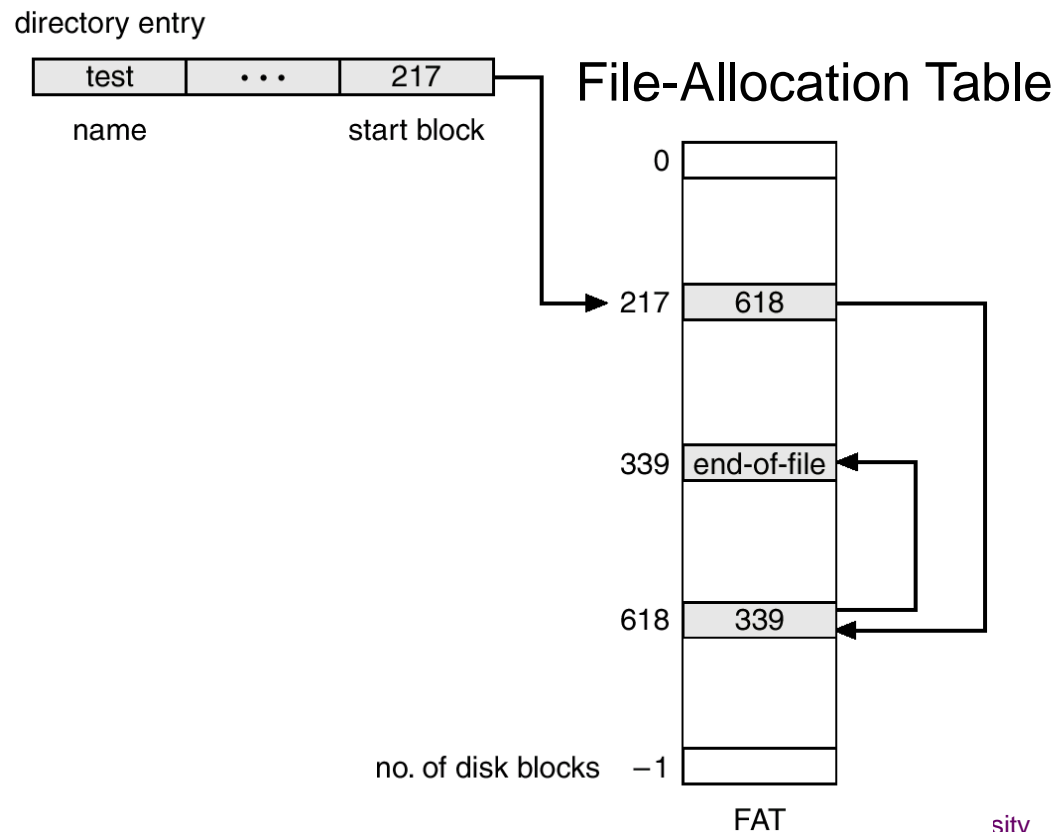
- No random access
- Each block contains a pointer, wasting space
- Blocks scatter everywhere and a large number of disk seeks may be necessary
- Reliability: what if a pointer is lost or damaged?

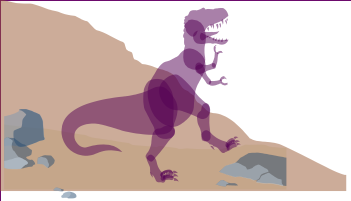


Variant of Linked Allocation Method

■ FAT (File Allocation Table) variation

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

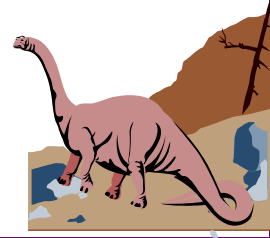




Question about FAT

- Given the values in the FAT, mark the block addresses that start a file

	Busy	Next	
0	0	-1	
1	1	6	
2	1	-1	
3	1	1	✓
4	0	-1	
5	1	-1	✓
6	1	-1	
7	1	2	✓





Problem about FAT

■ Assume:

- ◆ Disk Size = 32GB
- ◆ Block Size = 4 kB

■ Then,

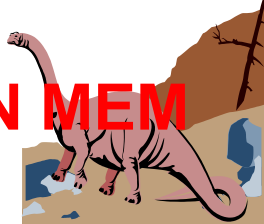
- ◆ Number of Blocks = 8M
- ◆ Size of FAT table = $8B * 8M = 64MB$, **CAN FIT IN MEMORY**

■ However, if we assume

- ◆ Disk Size = 4TB
- ◆ Block Size = 4 kB

■ Then,

- ◆ Number of Blocks = 1Giga
- ◆ Size of FAT table = $8B * 1G = 8GB$, **CANNOT FIT IN MEM**

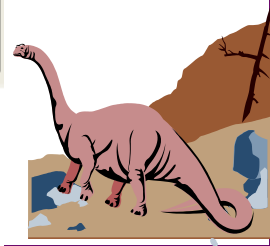
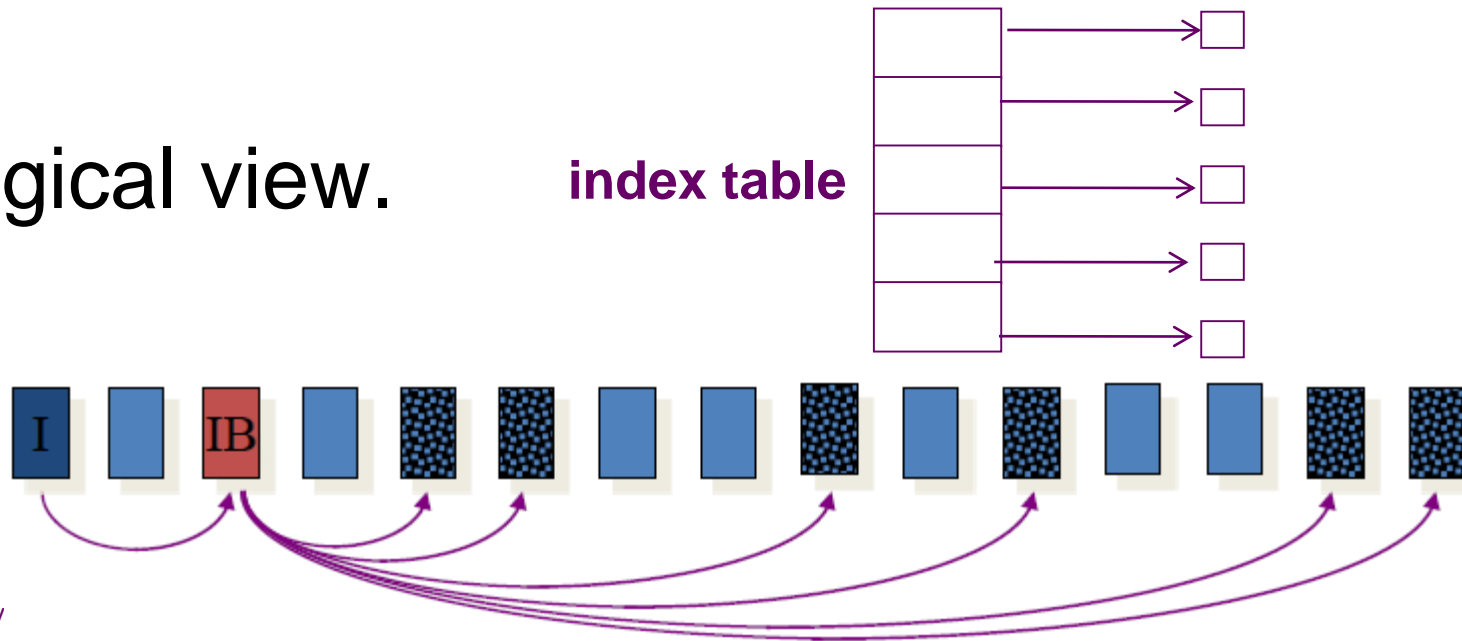




Indexed Allocation

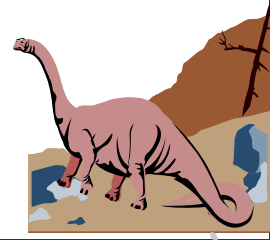
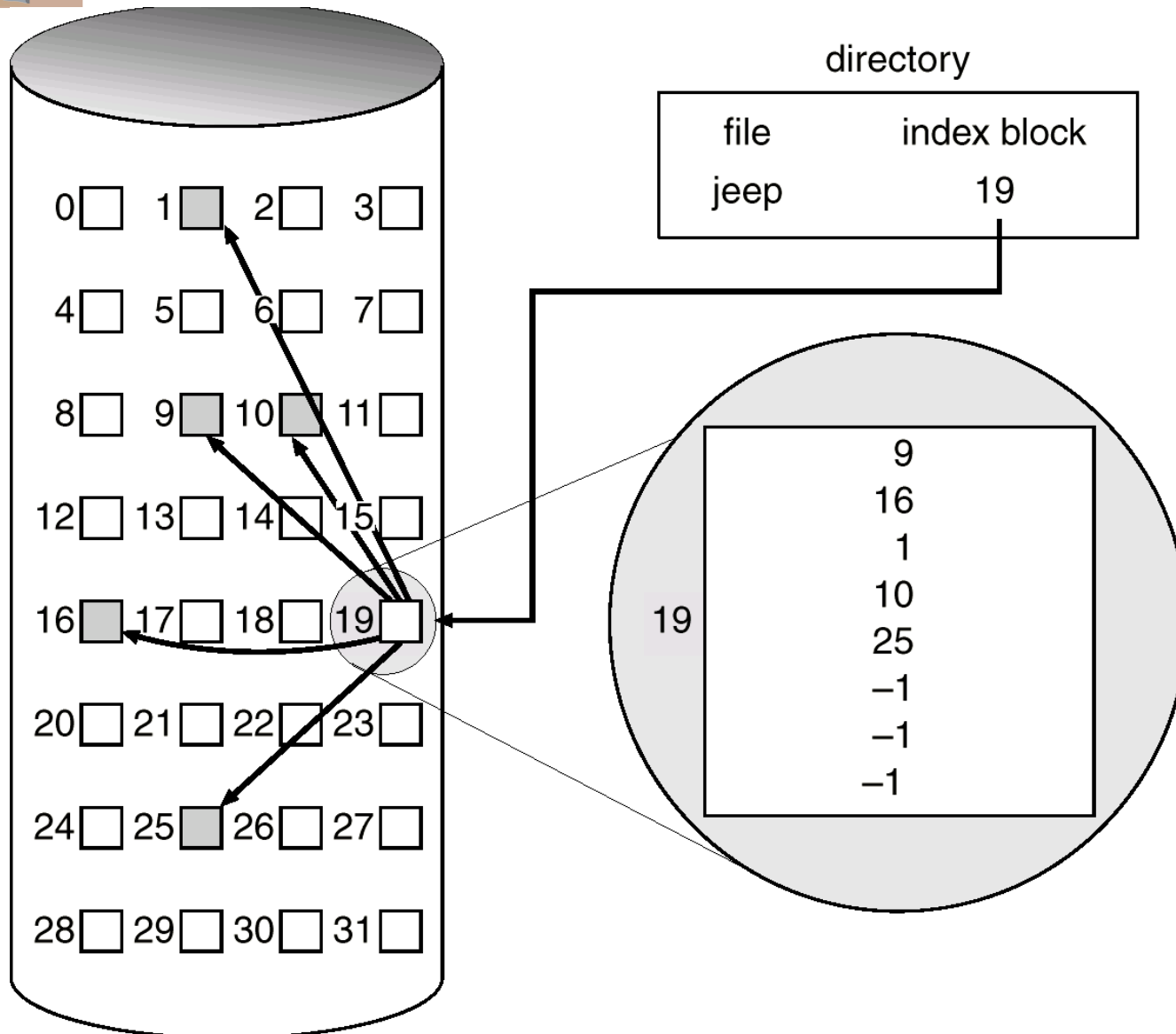
- Brings all pointers together into the index block.
- A file's directory entry contains a pointer to its index block.
- Hence, the index block of an indexed allocation plays the same role as the page table.

- Logical view.





Example of Indexed Allocation





Indexed Allocation (cont.)

- Support the random access
- The indexed allocation suffers from wasted space. The index block may not be fully used (i.e., internal fragmentation).
- The number of entries of an index table determines the upper bound for the size of a file. But the file size may exceed the bound.
- To overcome this problem, we must extend the indexed allocation method.

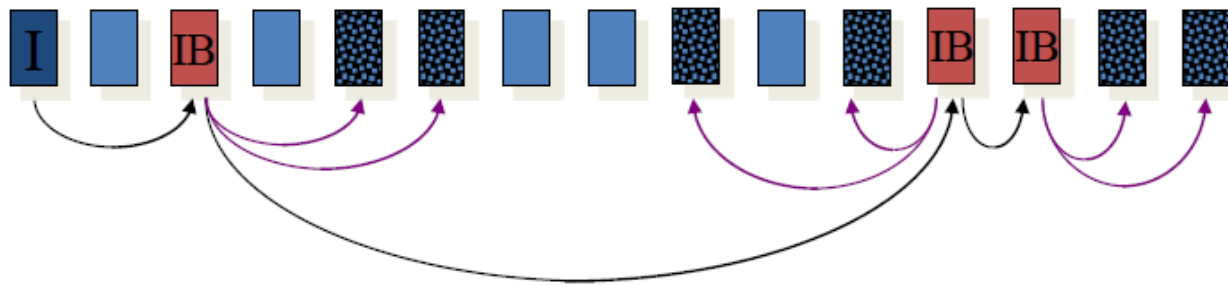




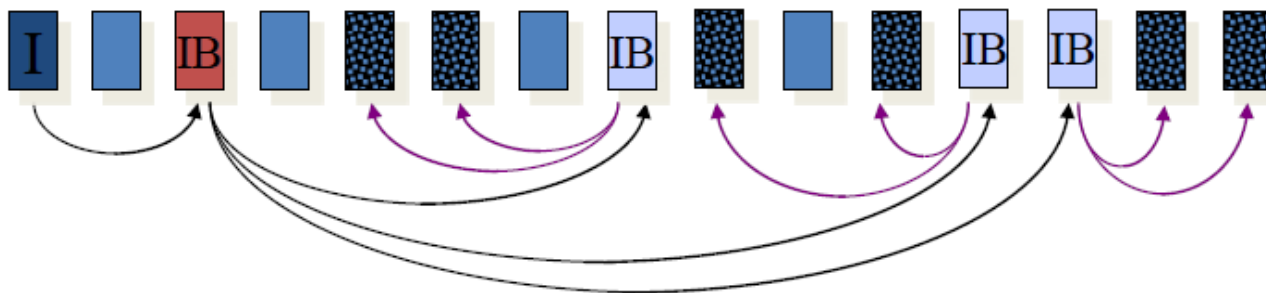
Indexed Allocation (cont.)

■ Improve index allocation method for large files

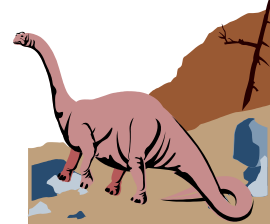
- ◆ multiple index blocks, chain them into a linked-list



- ◆ multiple index blocks, but make them a tree just like the multiple-level indexed access method

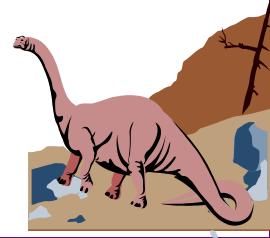
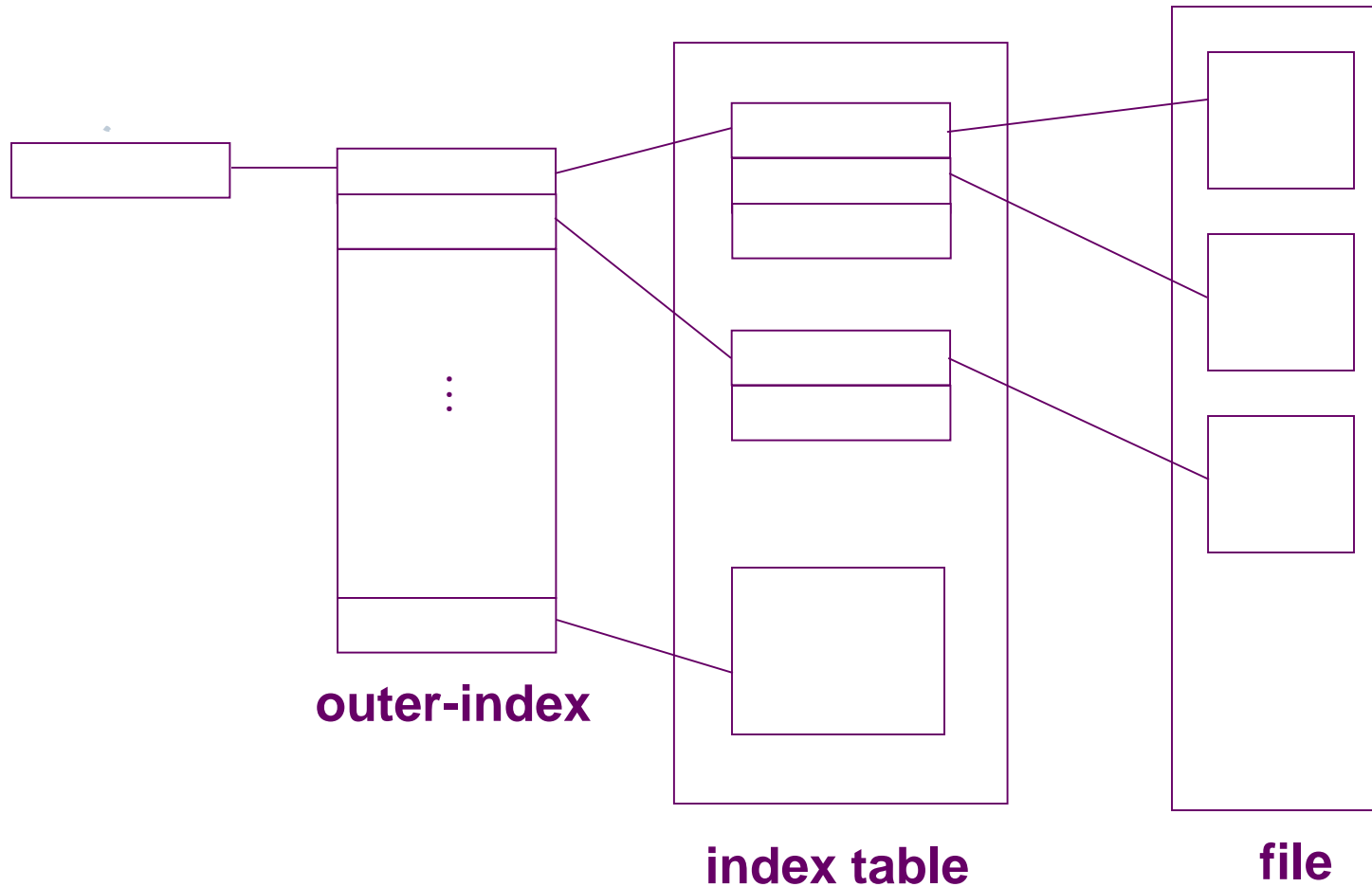


- ◆ a combination of both



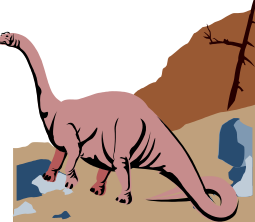
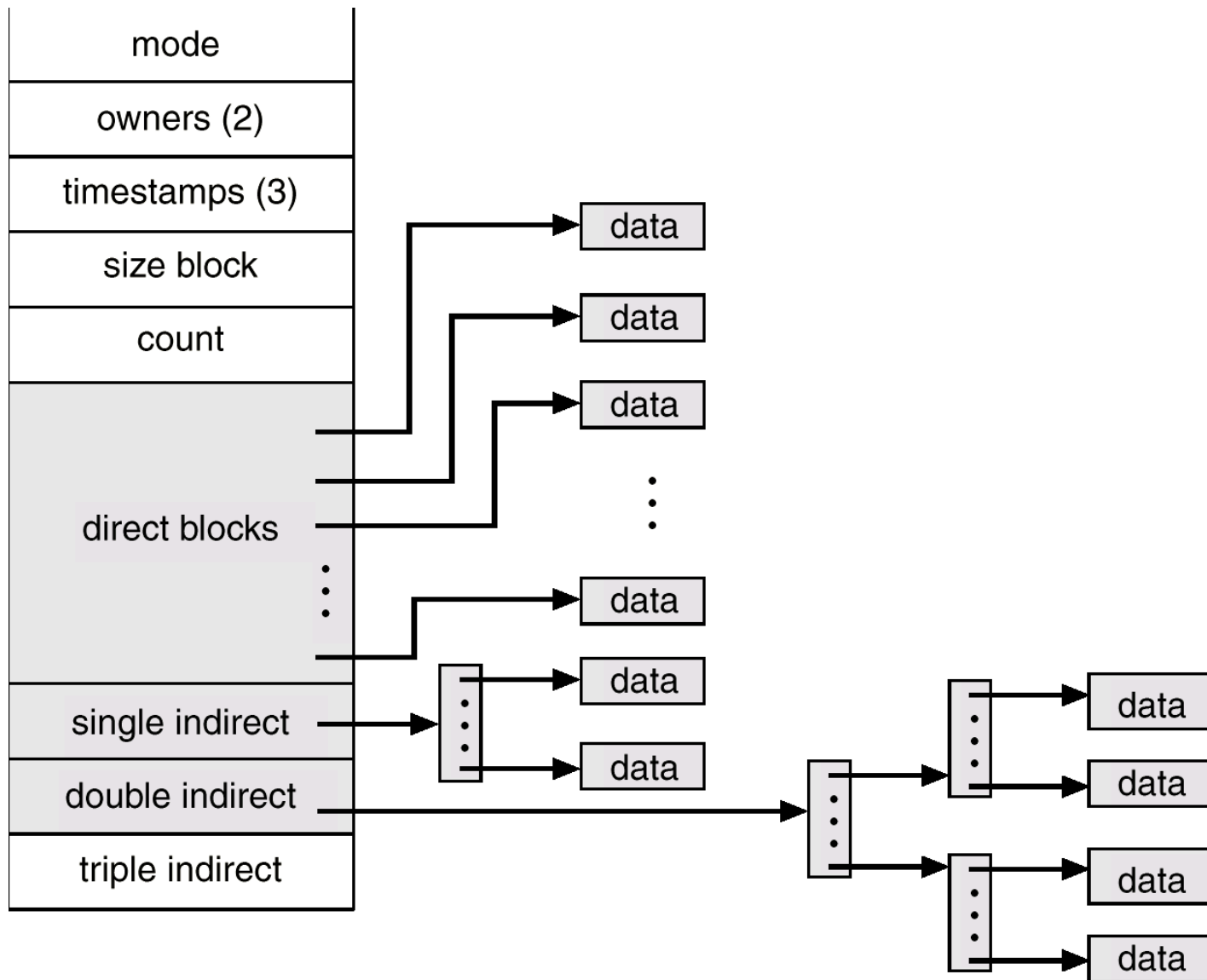


Indexed Allocation (cont.)



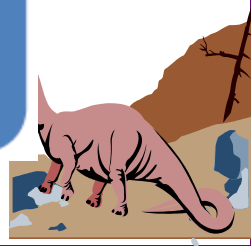
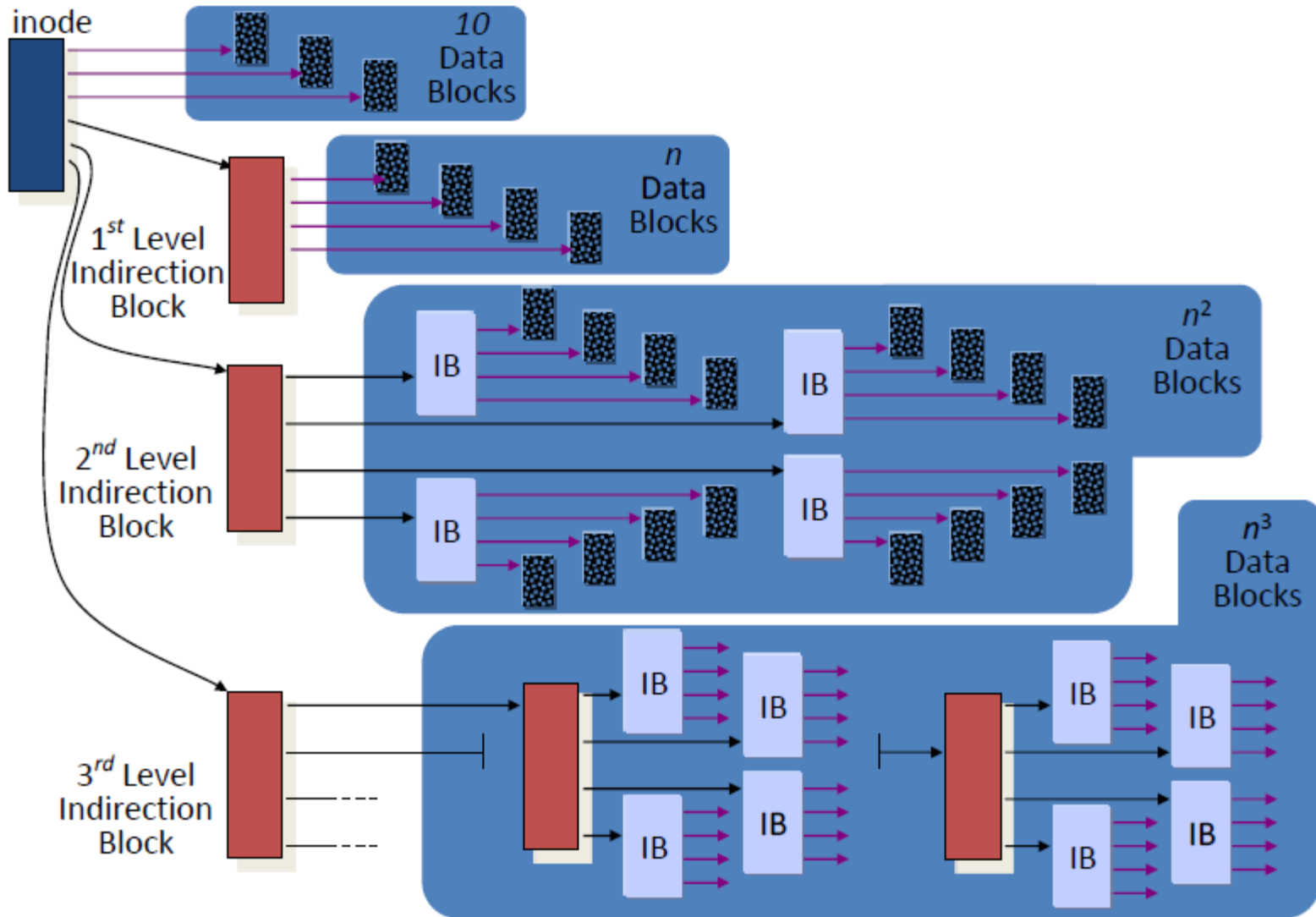


Combined Scheme: UNIX inode (4K Bytes per Block)





Another Illustration of Multi-level Indexed Allocation in UNIX





Performance

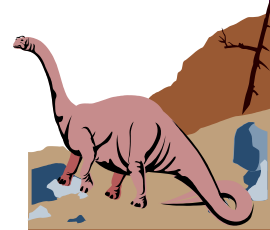
- Best method depends on file access type
 - ◆ Contiguous allocation method is great for both the sequential access and the random access
- Linked allocation method is good for sequential access, but not for random access
- Indexed allocation method is more complex
 - ◆ Good for both sequential access and random access
 - ◆ But single block access could require 2 index block reads and then data block read
- If the access type can be declared at file creation time, then select either contiguous or linked





Chapter 11: File System Implementation

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





Efficiency and Performance

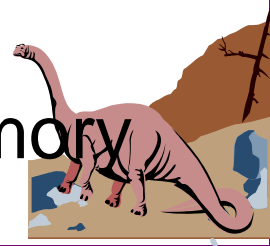
■ Efficiency dependent on:

- ◆ types of data kept in file's directory entry
- ◆ disk allocation and directory algorithms
- ◆ In systems that implement i-nodes, reading a file requires two disk accesses: one for the i-node and a second one to access the blocks.

■ Performance optimization methods

- ◆ free-behind and read-ahead – techniques to optimize sequential access
- ◆ improve PC performance by dedicating section of memory as virtual disk, or RAM disk
- ◆ disk cache – separate section of main memory

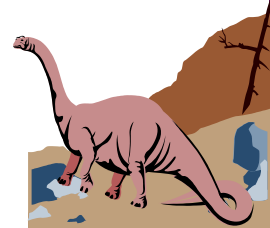
for frequently used blocks





Free-behind and Read-ahead

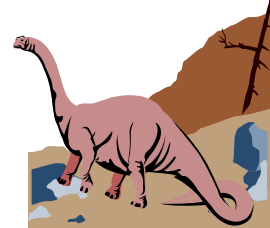
- **Free-behind** - removes a block from the buffer as soon as the next block is requested
 - The previous blocks are not likely to be used again and waste buffer space
 - E.g., in video files
- **Read-ahead** - a requested block and several subsequent blocks are read and cached





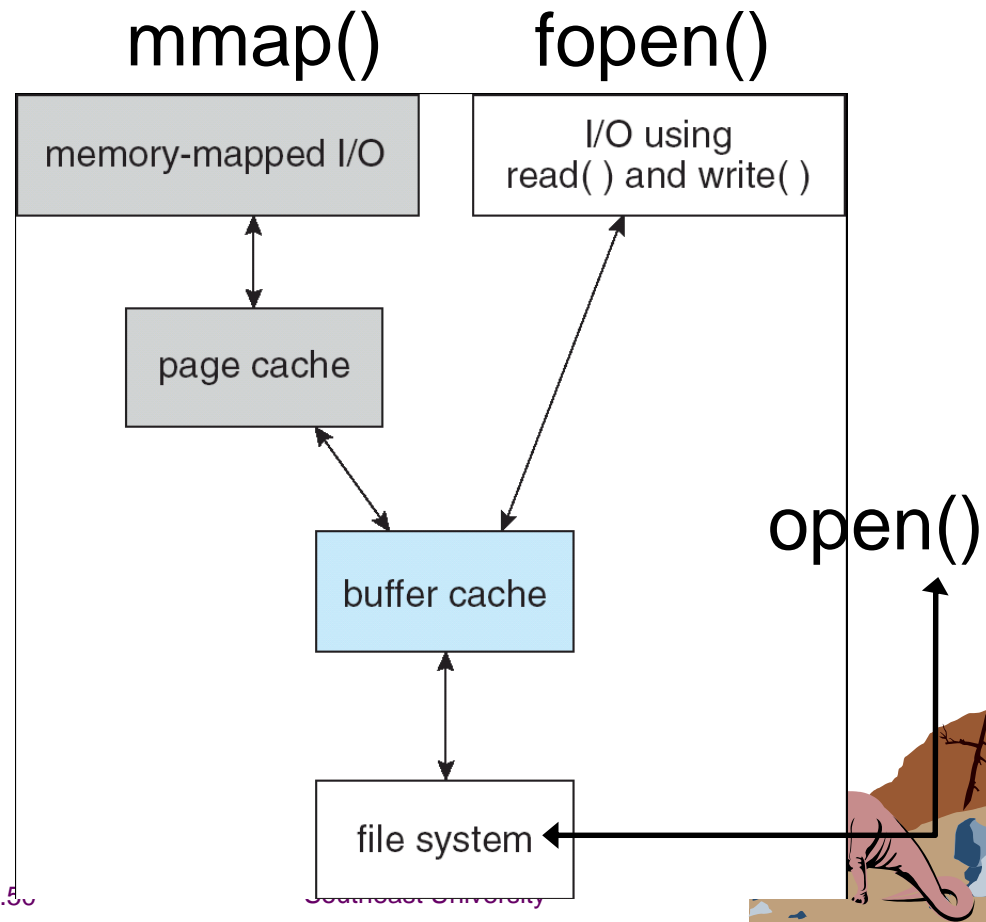
Virtual Disk or RAM Disk

- Performance is improved by having a section of memory set aside and treated as a virtual disk or RAM disk
 - average lifetime of a file on Unix is about 2 seconds
 - most files are temporary; created by programs and discarded
 - Unix has elaborate caching schemes & usually these temporary files are never written to disk



Page Cache

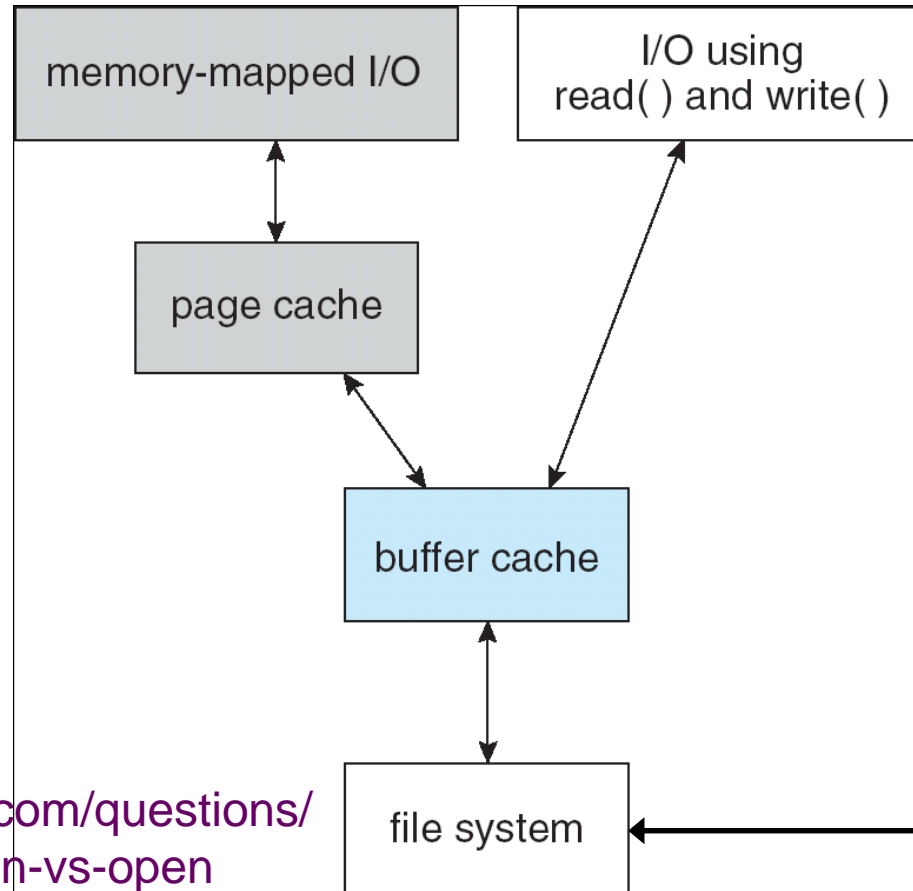
- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer(disk) cache
 - ◆ Buffer cache – separate section of main memory for frequently used blocks





I/O Without a Unified Buffer Cache

mmap()



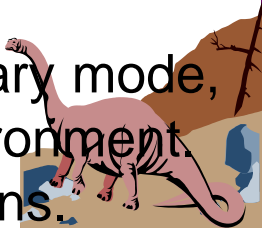
fopen()

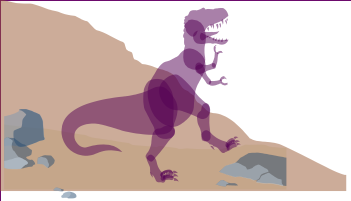
open()

<http://stackoverflow.com/questions/1658476/c-fopen-vs-open>

There are three main reasons to use fopen instead of open.

- fopen provides you with buffering IO that may turn out to be a lot faster than what you're doing with open.
- fopen does line ending translation if the file is not opened in binary mode, which can be helpful if your program is ported to a non-Unix environment.
- A FILE * gives you the ability to use fscanf and other stdio functions.





Unified Buffer Cache

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

