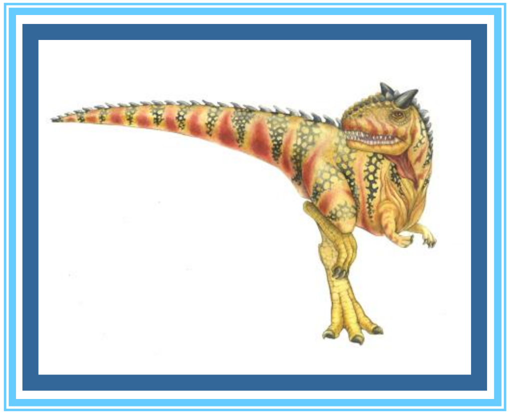
# Chapter 12: File System Implementation

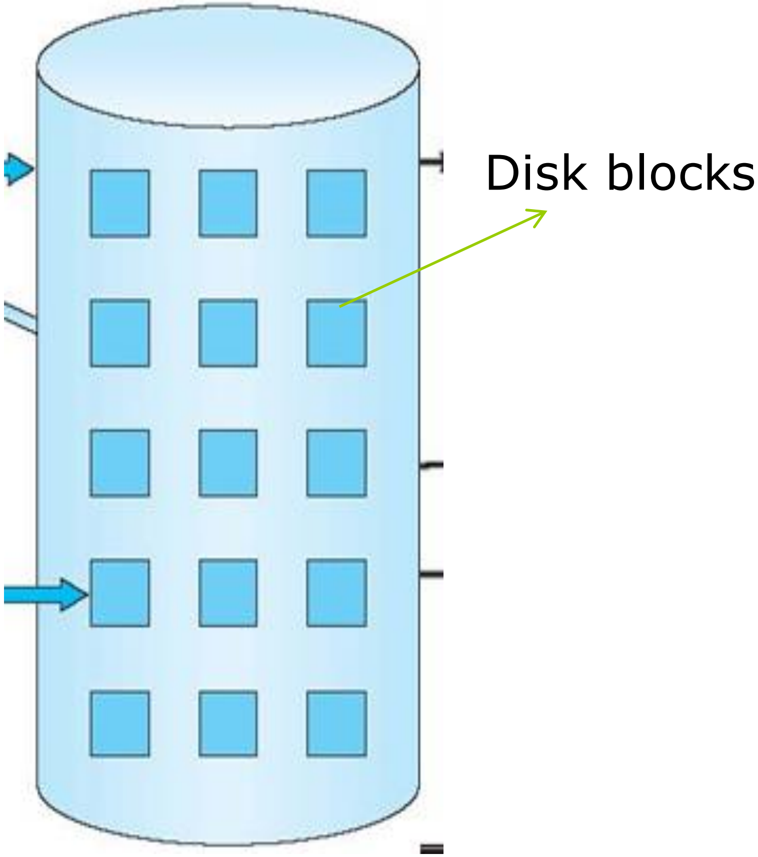


# File-System Structure

- Disks provide most of the secondary storage on which file systems are maintained.
- Two characteristics make them convenient for this purpose:
- Disk provides **in-place rewrite and random access**
  - A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
  - A disk can access directly any block of information it contains.
    - It is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read–write heads and waiting for the disk to rotate.

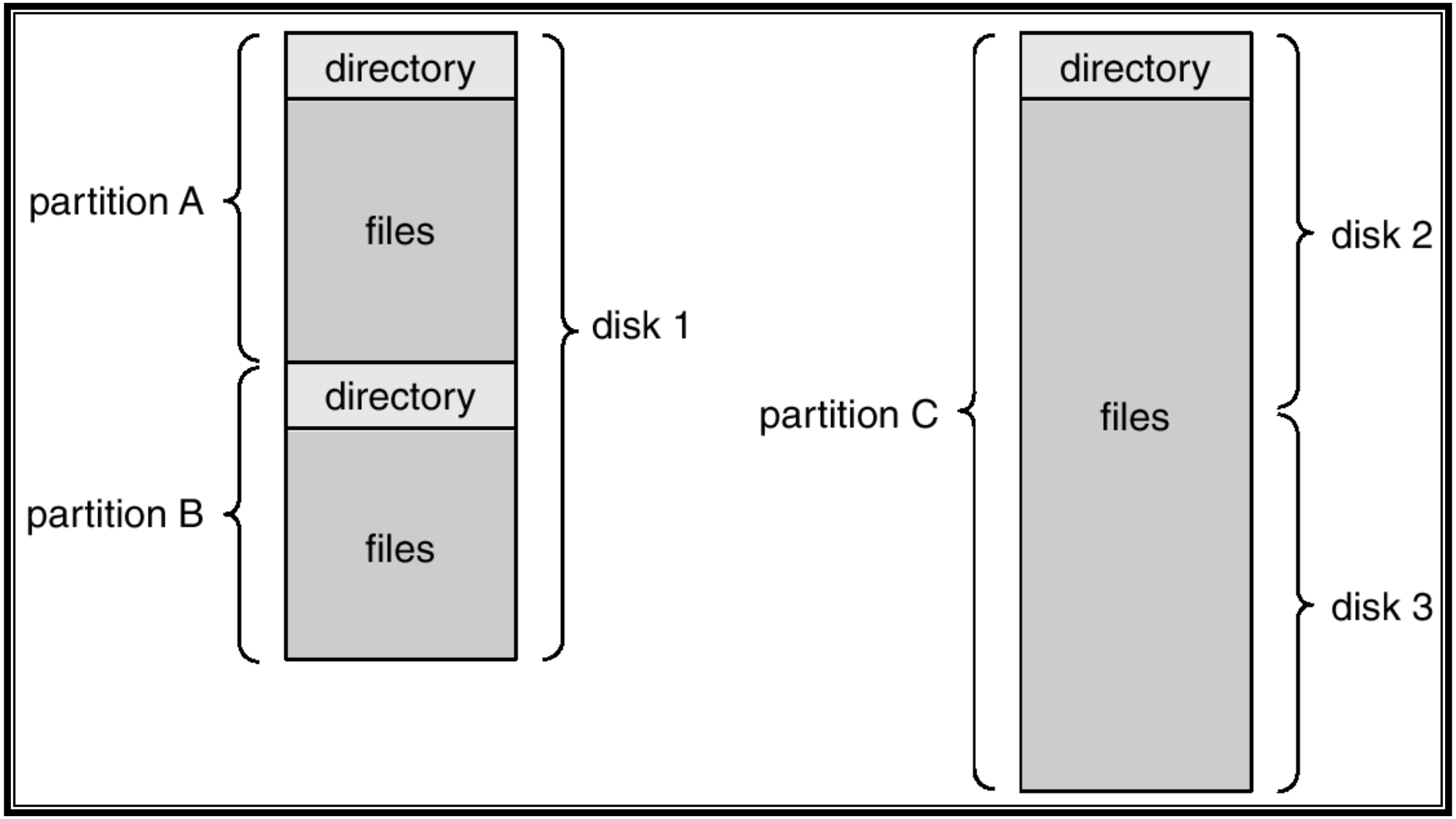
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**.
- Each block has one or more sectors.
- Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.

- File structure
  - Logical storage unit
  - Collection of related information
- Files stored on disks.
- Disks broken up into one or more partitions, with separate file system on each partition



Data stored within the block

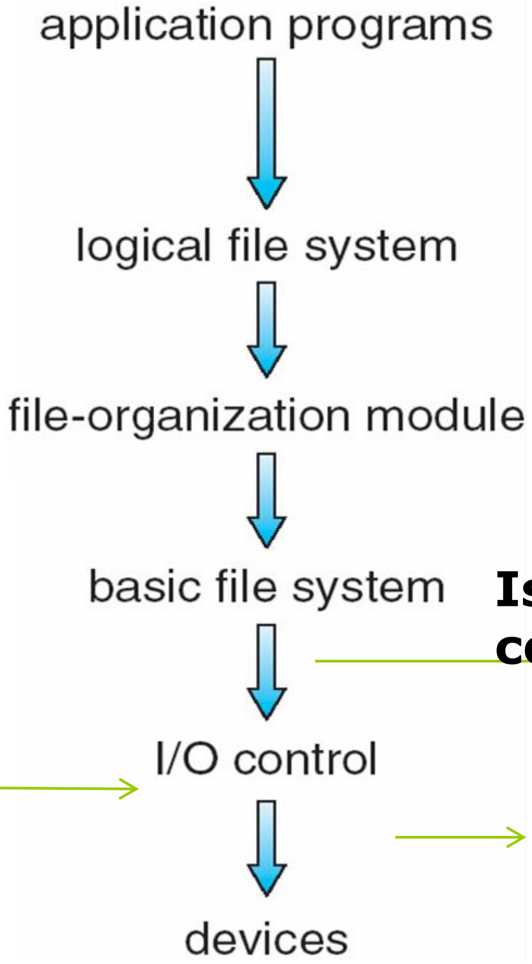
# A Typical File-system Organization



- A file system poses two quite different design problems.
- The first problem is defining how the file system should look to the user.
  - This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
- The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- **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

# Layered File System

- Each level uses the feature of low level
- Create new features for higher level



Manages **FCB**

Translates logical to physical blocks

**Issue commands**

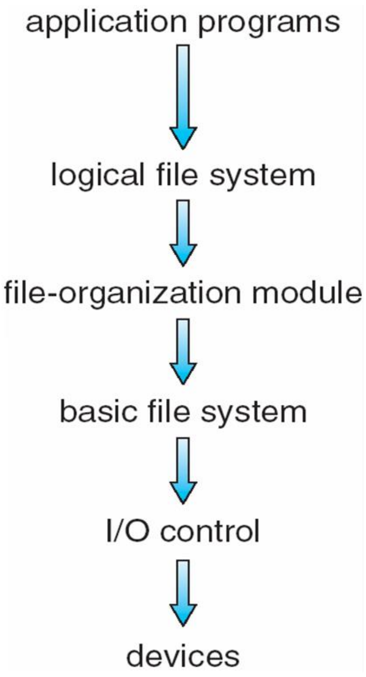
R/W Physical block (cylinder, track, sector)

Device driver and interrupt handlers to transfer information between memory/disk

Hardware specific instruction

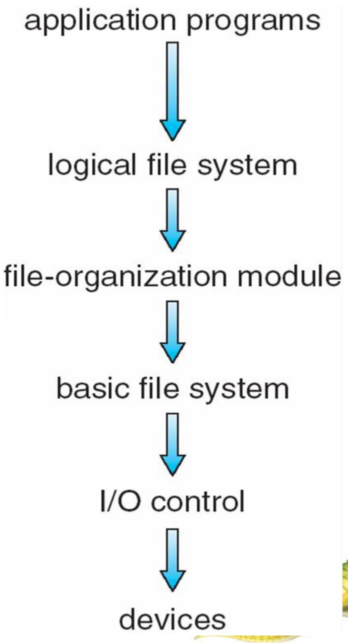
# File System Layers

- **I/O control layer** consists of device drivers
  - **Device driver** controls the physical device ; They 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
  - The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.



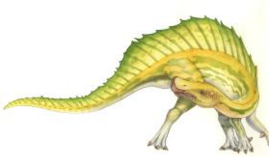
# File System Layers (Cont.)

- **Basic file system** Issues generic commands to the appropriate device driver to read and write physical blocks on the disk (sector, track)
  - Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).
- Also manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data



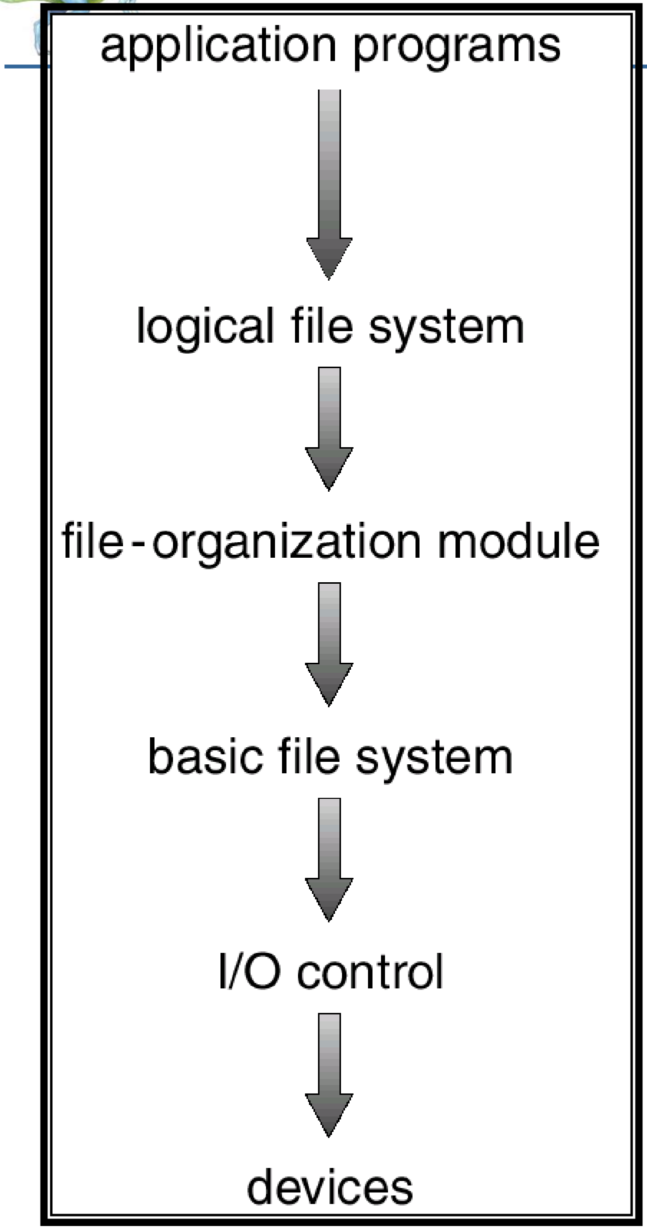
# File System Layers (Cont.)

- **File organization module** understands files, logical address, and physical blocks
- Translates logical block # to physical block #
  - By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
  - Each file's logical blocks are numbered from 0 (or 1) through *N.*
  - Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block.
- Manages free space, disk allocation
  - The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested



# File System Layers (Cont.)

- **Logical file system** manages metadata information
- Directory management
  - The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name.
- Translates file name into file number, file handle, location by maintaining file control blocks
  - It maintains file structure via file-control blocks
  - A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents.
- Protection



application programs

logical file system

file-organization module

basic file system

I/O control

devices

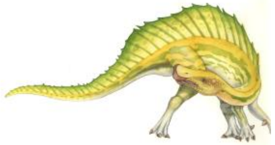
Manages meta data about files, file organization, directory structure, file control blocks, etc.

Mapping of logical block# (0..n) to physical block# (sector, track #, etc), free space mgmt

Issues generic commands to device drive to R/W physical blocks on disk

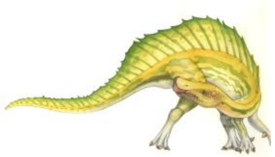
Device drivers, interrupt service routines, etc

- Logical layers can be implemented by any coding method according to OS designer
- When a layered structure is used for file-system implementation, duplication of code is minimized.
- The I/O control and sometimes the basic file-system code can be used by multiple file systems.
- Each file system can then have its own logical file-system and file-organization modules.
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance.
- The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.



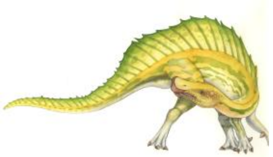
# File System Layers (Cont.)

- 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 40 types, with **extended file system** ext2 and ext3 leading;
    - Distributed file systems, etc.
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

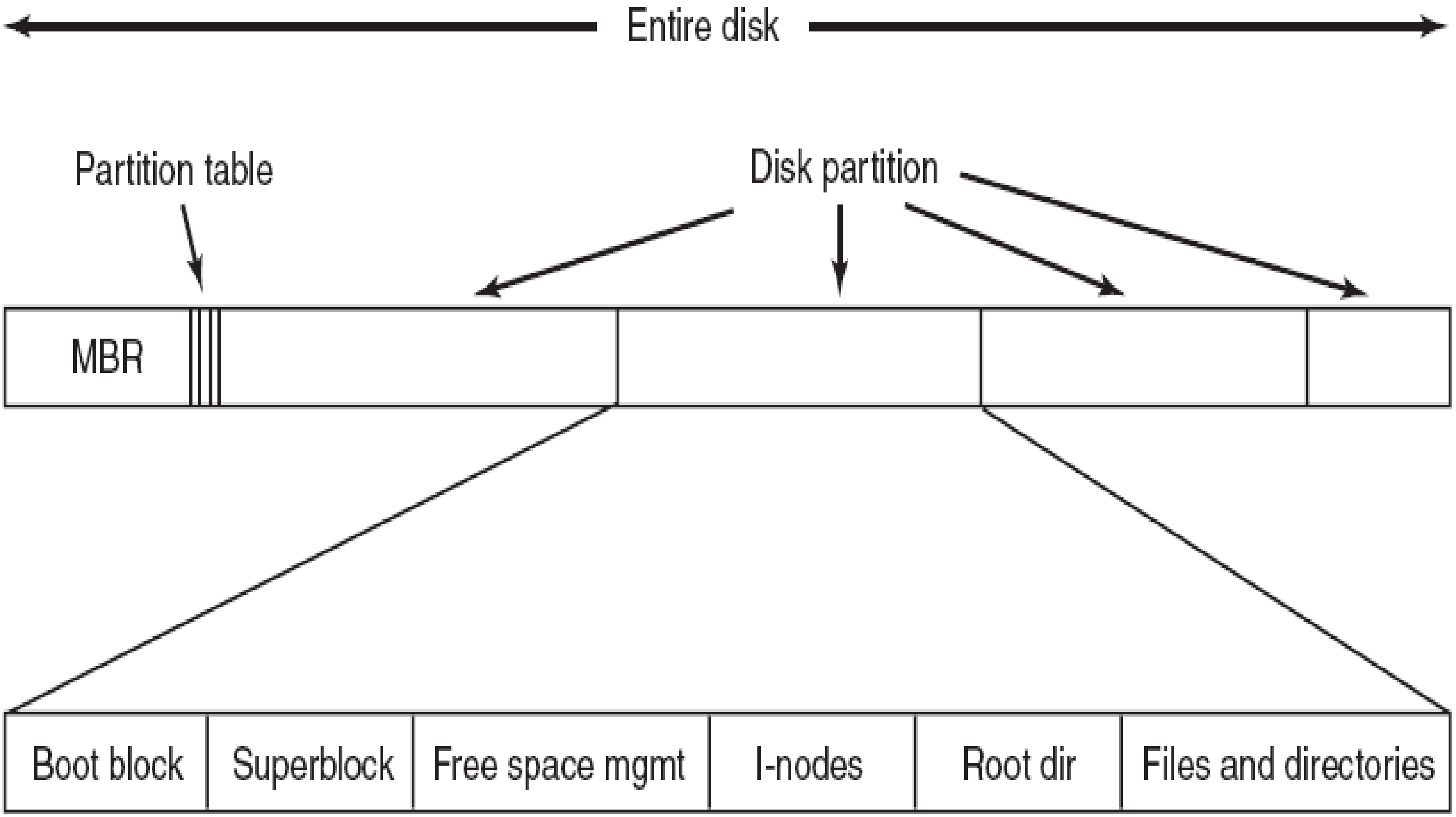


# Disk Layout

- Files stored on disks. Disks broken up into one or more partitions, with separate file system on each partition
- Sector 0 of disk is the Master Boot Record
- Used to boot the computer
- End of MBR has partition table. Has starting and ending addresses of each partition.
- One of the partitions is marked active in the master boot table

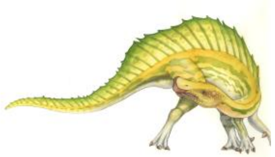


# Disk Layout


Entire disk
Partition table
Disk partition
MBR
Boot block
Superblock
Free space mgmt
I-nodes
Root dir
Files and directories

# File-System Implementation

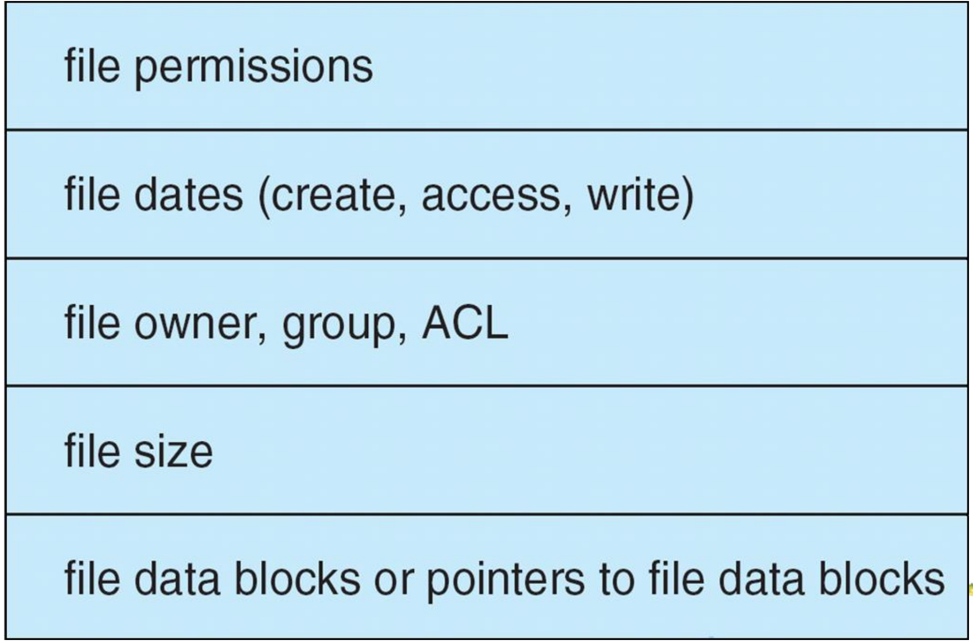
- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- **Boot control block** contains info needed by system to boot OS from that volume
  - If the disk does not contain an OS, this block can be empty.
  - It is typically the first block of a volume.
  - In UFS, it is called the **boot block**.
  - In NTFS, it is the **partition boot sector**.



- **Volume control block (superblock, master file table)** contains volume details
  - Total # of blocks in the partition , # of free blocks, block size, free block pointers or array and a free-FCB count and FCB pointers.
  - In UFS, this is called a **superblock**.
  - In NTFS, it is stored in the **master file table**.
- A **directory structure (per file system)** is used to organize the files.
  - In UFS, this includes file names and associated inode numbers.
  - In NTFS, it is stored in the master file table.

# File-System Implementation (Cont.)

- **File control block** – storage structure consisting of information about a file
  - It has a unique identifier number to allow association with a directory entry.
  - In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file



file permissions

file dates (create, access, write)

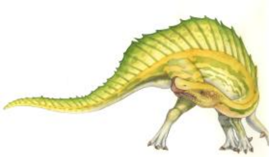
file owner, group, ACL

file size

file data blocks or pointers to file data blocks

# In-Memory File System Structures

- The in-memory information is used for both file-system management and performance improvement via caching.
- The data are loaded at mount time, updated during file-system operations, and discarded at dismount
  - An in-memory **mount table** contains information about each mounted volume.
  - An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)

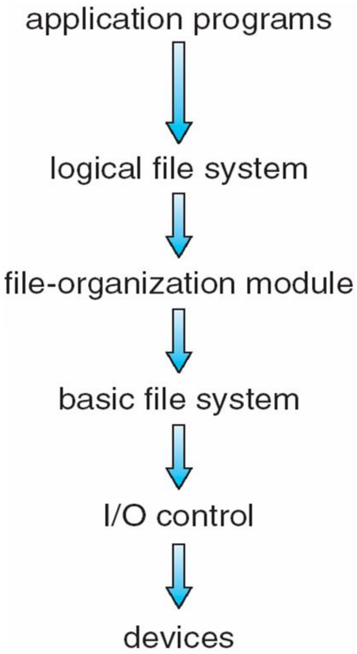


# In-Memory File System Structures

- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- **Buffers** hold file-system blocks when they are being read from disk or written to disk.

# File handling

- **Create a new file**
  - Application program calls the logical file system
- The LFS knows the format of the directory structures.
  - Allocates a new FCB
  - (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.)
  - Reads the appropriate directory into memory
  - Updates directory with new filename and FCB
  - Write it back to disk



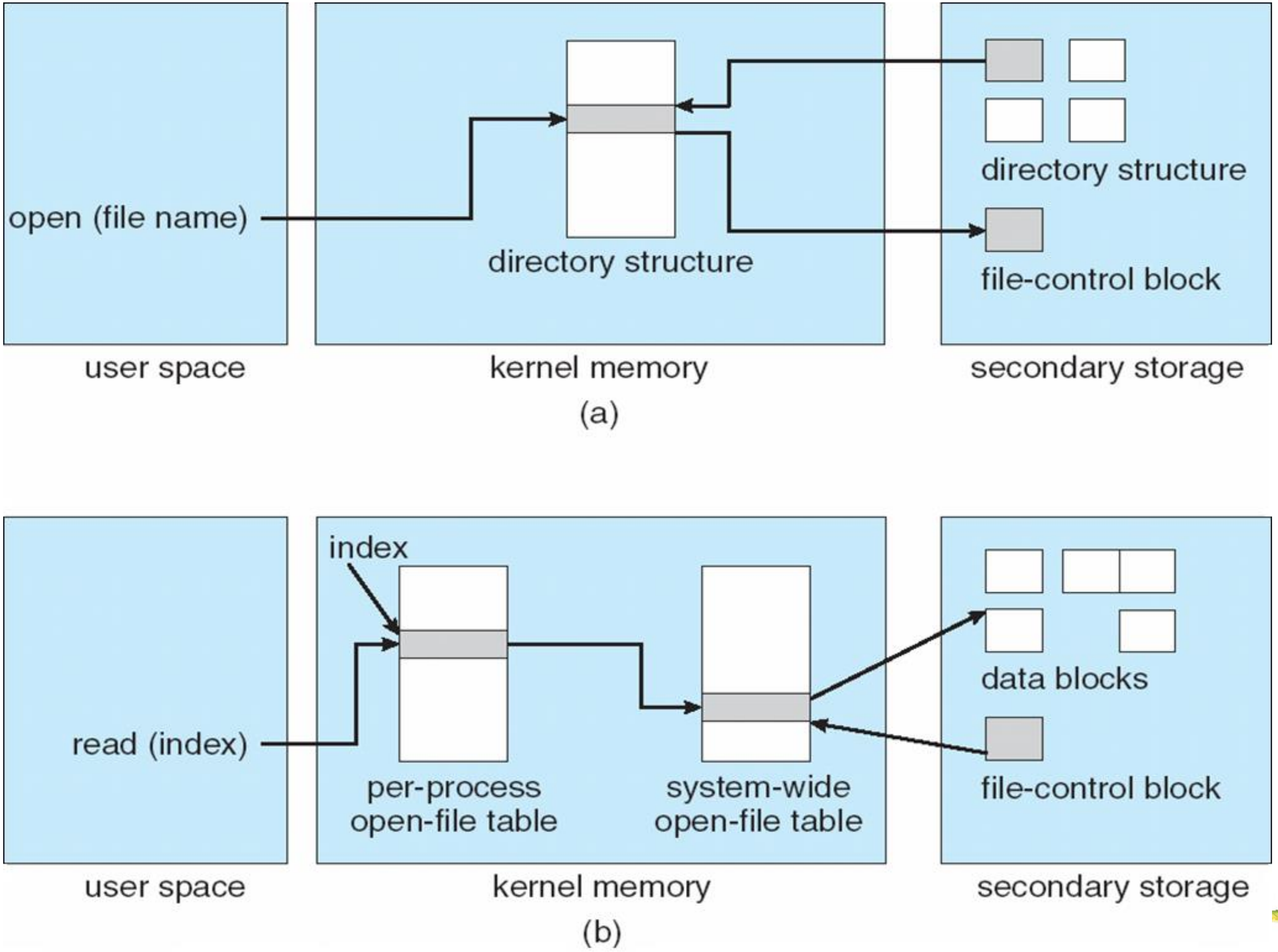
# File handling

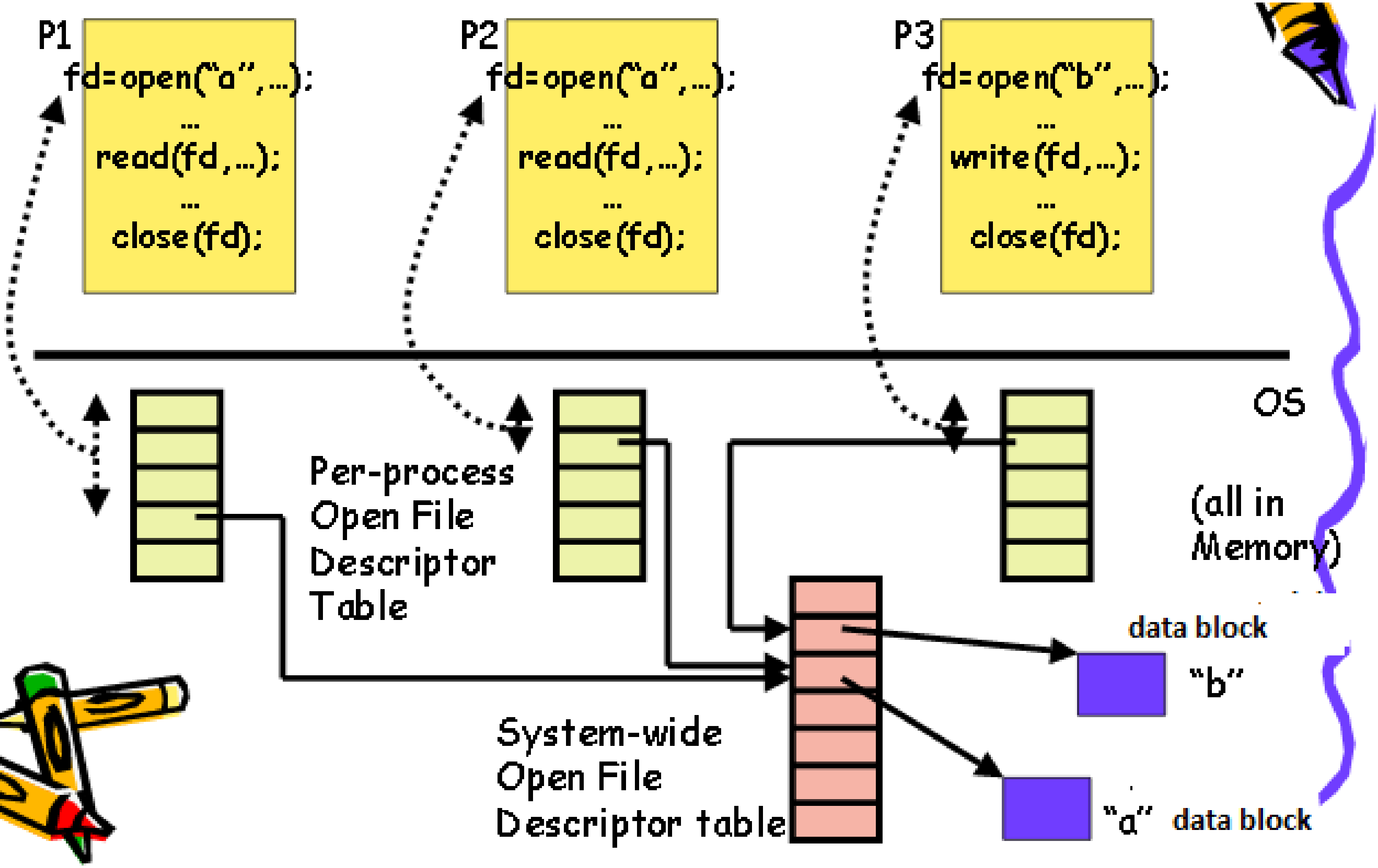
- Using the file (I/O)
- The open() call passes a file name to the logical file system.
- The open() system call first searches the system-wide open-file table to see if the file is already in use by another process.
- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- If the file is not already open, the directory structure is searched for the given file name.
- Parts of the directory structure are usually cached in memory to speed directory operations.
- Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- This table not only stores the FCB but also tracks the number of processes that have the file open.

- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.
- These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open.
- The open() call returns a pointer to the appropriate entry in the per-process file-system table.
- All file operations are then performed via this pointer.
- The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk.
- It could be cached, though, to save time on subsequent opens of the same file.

- The following figure illustrates the necessary file system structures provided by the operating systems
- Figure 12-3(a) refers to opening a file
- Figure 12-3(b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- The name given to the entry varies.
  - UNIX systems refer to it as a **file descriptor**
  - Windows refers to it as a **file handle**
- Data from read eventually copied to specified user process memory address

# In-Memory File System Structures





```
fd=open("a",…);
…
read(fd,…);
…
close(fd);
```

```
fd=open("a",…);
…
read(fd,…);
…
close(fd);
```

```
fd=open("b",…);
…
write(fd,…);
…
close(fd);
```

OS
Per-process Open File Descriptor Table
(all in Memory)
data block
"b"
System-wide Open File Descriptor table
"a"
data block

- A process closes a file
  - Per process table entry removed
  - System table count decremented
- All processes closed the file
  - Updated file info is copied back to disk
  - System wide open file table entry removed

# Partitions and Mounting

- Partition can be a volume containing a file system (“cooked”) or **raw** – just a sequence of blocks with no file system
  - **Raw disk** is used where no file system is appropriate.
  - UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system
- Boot information can be stored in a separate partition
  - It has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
  - Boot information is usually a sequential series of blocks, loaded as an image into memory.
  - Execution of the image starts at a predefined location, such as the first byte.

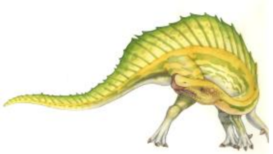
- This **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing
- It can contain more than the instructions for how to boot a specific operating system.
- **Dual-booted -** allowing us to install multiple operating systems on a single system.
- How does the system know which one to boot?
- A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space.
- Once loaded, it can boot one of the operating systems available on the disk.
- The disk can have multiple partitions, each containing a different type of file system and a different operating system.

# Partitions and Mounting

- **Root partition** contains the OS, other partitions can hold other Oses, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- As part of a successful mount operation, the os verifies that the device contains a valid file system.
- It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

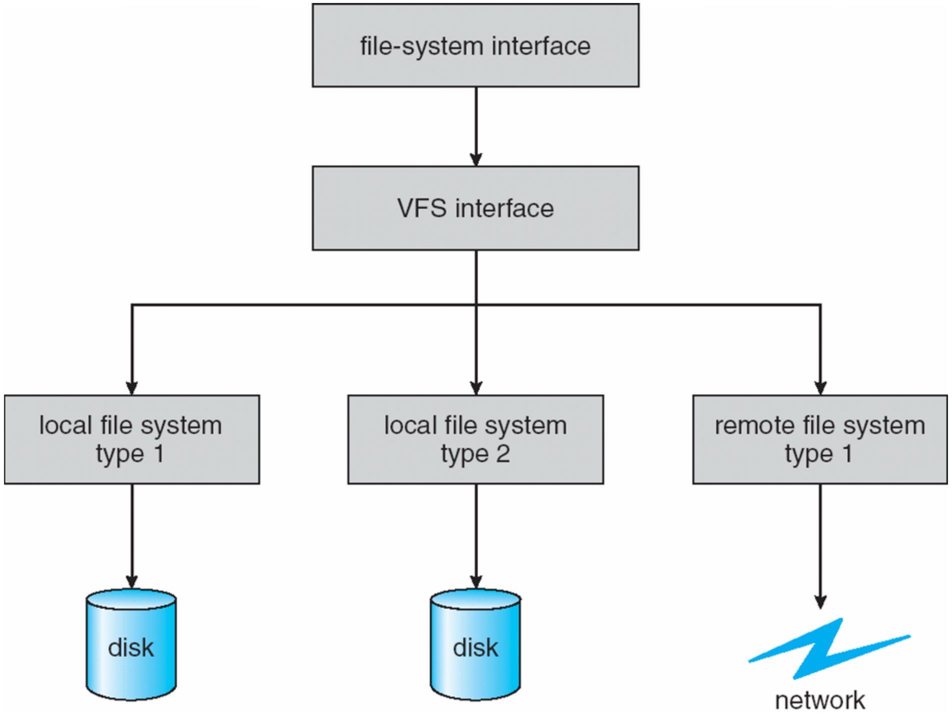
# Virtual File Systems

- **Virtual File Systems** (**VFS**) on Unix 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
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements **vnodes** which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines



# Virtual File Systems (Cont.)

- The API is to the VFS interface, rather than any specific type of file system

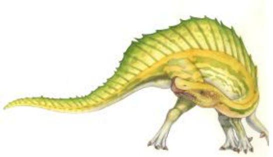


# Virtual File System Implementation

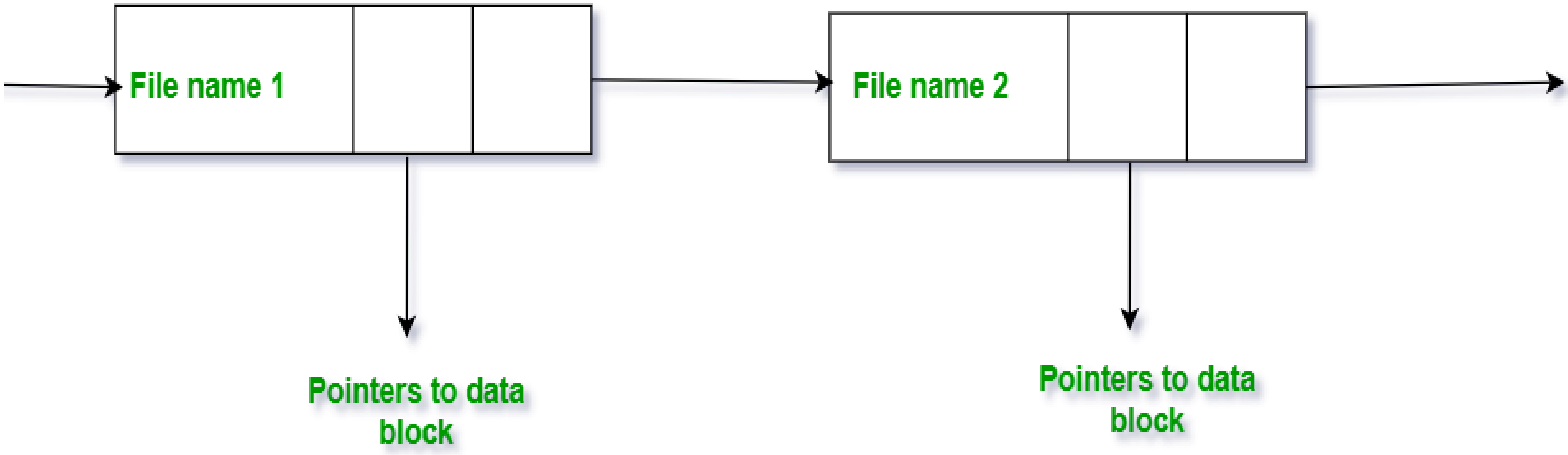
- For example, Linux has four object types:
  - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
  - Every object has a pointer to a function table
    - Function table has addresses of routines to implement that function on that object
    - For example:
    - • **`int open(. . .)`**—Open a file
    - • **`int close(. . .)`**—Close an already-open file
    - • **`ssize t read(. . .)`**—Read from a file
    - • **`ssize t write(. . .)`**—Write to a file
    - • **`int mmap(. . .)`**—Memory-map a file

# Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program but Time-consuming to execute
  - To create a new file, first search the directory to be sure that no existing file has the same name. Then, add a new entry at the end of the directory.
  - To delete a file, search the directory for the named file and then release the space allocated to it.



## Directory Implementation Using Singly Linked List



File name 1

Pointers to data block

File name 2

Pointers to data block

# Directory Implementation using Singly Linked List

- To reuse the directory entry
  - mark the entry as unused (by assigning it a special name, such as an all-blank name, or by including a used–unused bit in each entry),
  - attach it to a list of free directory entries.
  - copy the last entry in the directory into the freed location and to decrease the length of the directory.

# Disadvantage - Singly Linked List

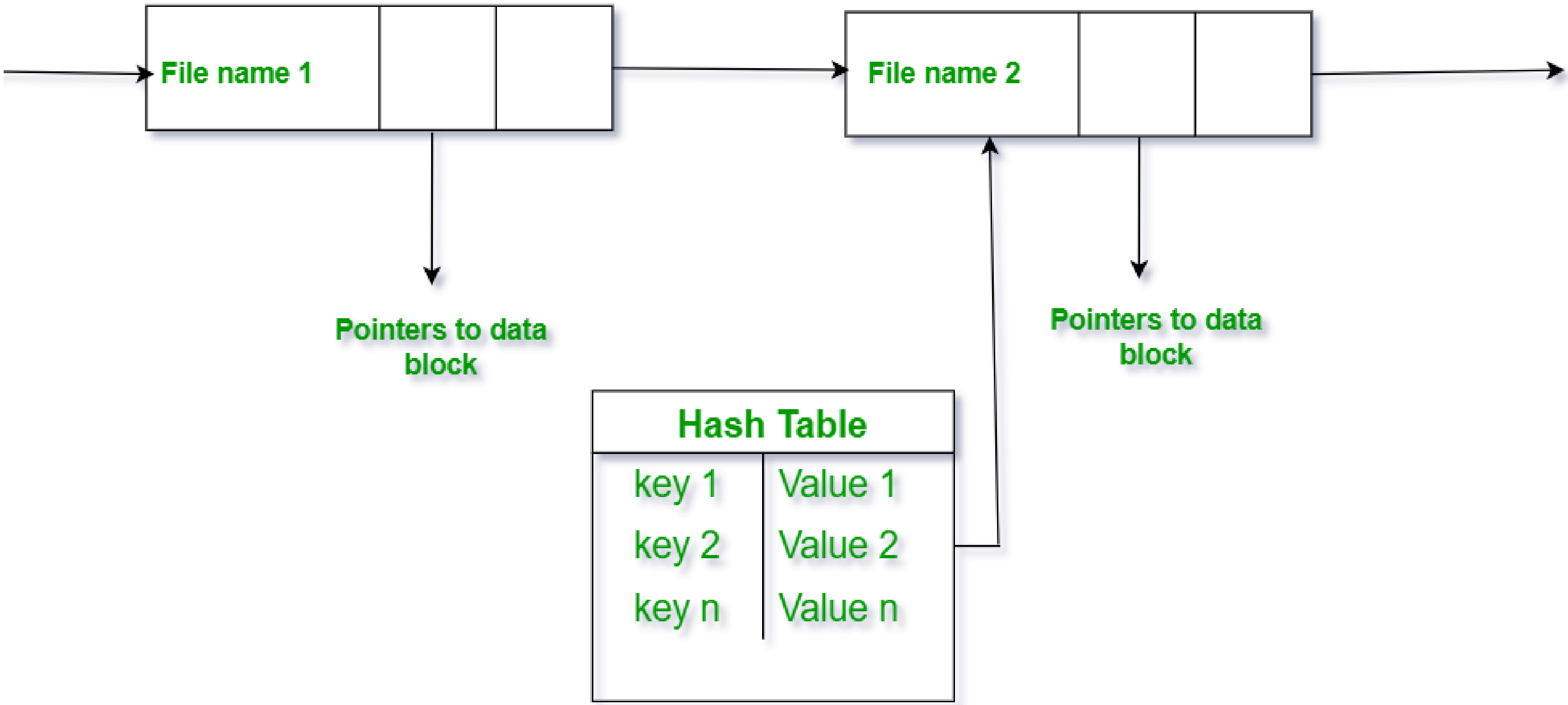
- A linked list can also be used to decrease the time required to delete a file
- Disadvantage is that finding a file requires a linear search.
- Directory information is used frequently, and users will notice if access to it is slow.
  - In fact, many OS implement a software cache to store the most recently used directory information.
  - A cache hit avoids the need to constantly reread the information from disk.

- A sorted list allows a binary search and decreases the average search time.
  - complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory.
- A balanced tree, might help here.

# Directory Implementation

- **Hash Table** – linear list with hash data structure
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list
- 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
  - Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries.

# Directory Implementation Using Hash Table



File name 1

Pointers to data block

File name 2

Pointers to data block

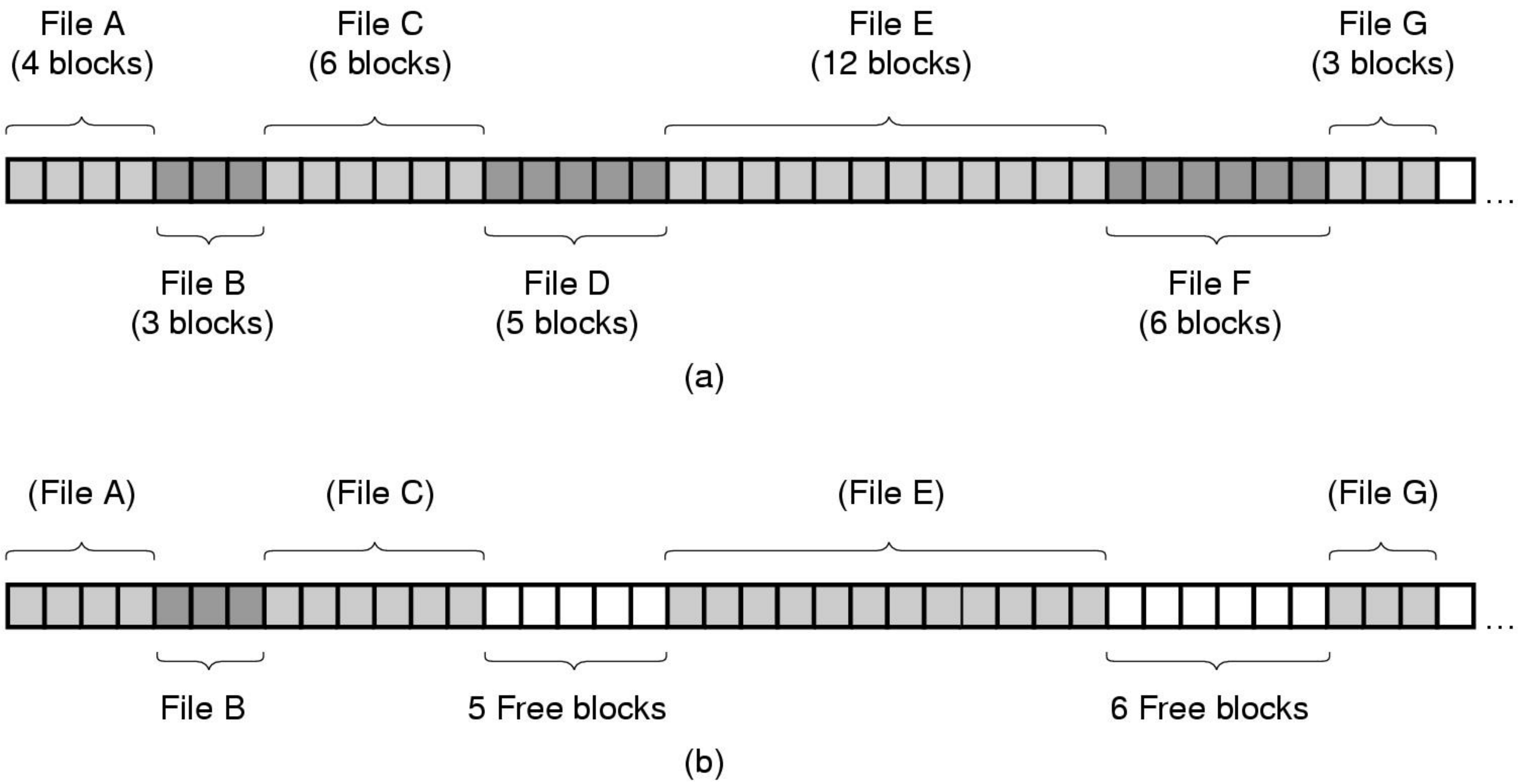
| Hash Table | |
|---|---|
| key 1 | Value 1 |
| key 2 | Value 2 |
| key n | Value n |

# Allocating Blocks to files

- Most important implementation issue
- Methods
  - Contiguous allocation
  - Linked list allocation
  - Linked list using table
  - I-nodes

# Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – 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 for file
    - knowing file size
    - external fragmentation
    - need for **compaction off-line** (**downtime**) or **on-line**

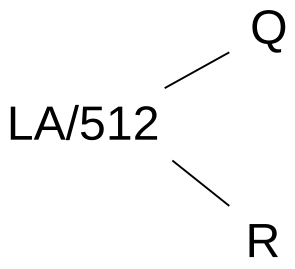


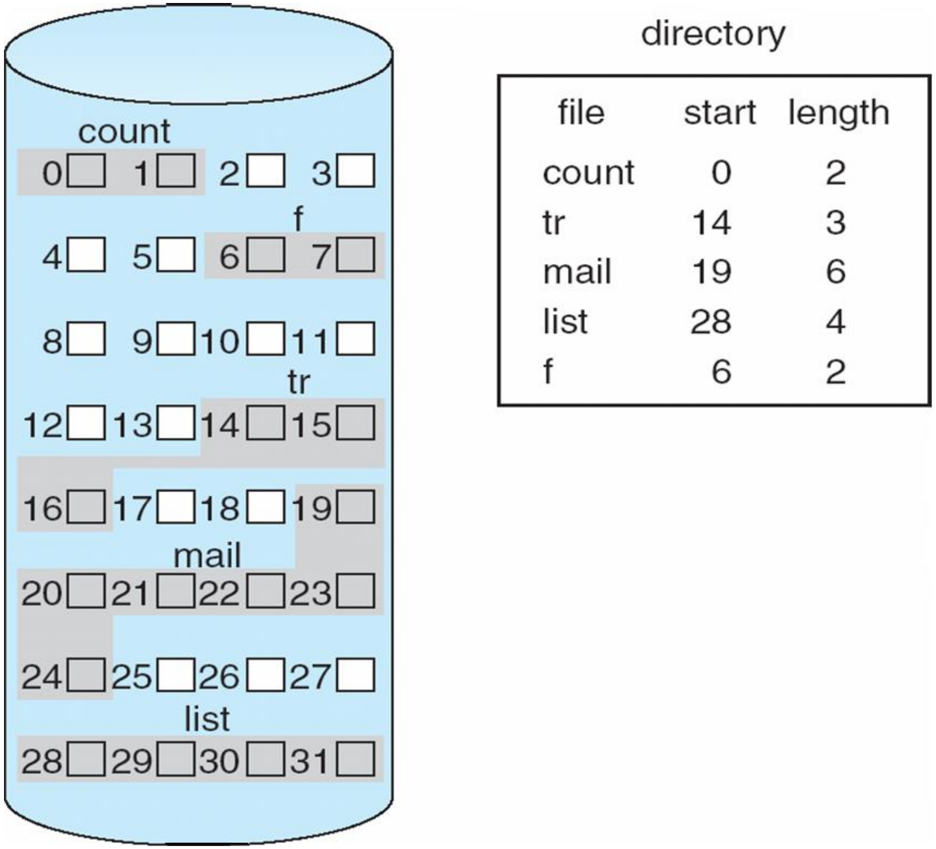
(a) Contiguous allocation of disk space for 7 files.

(b) The state of the disk after files D and F have been removed.

# Contiguous Allocation

- Mapping from logical to physical





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

Block to be accessed =
Q + starting address
Displacement into block = R

# Contiguous Allocation

The good

- Easy to implement
- Read performance is great. Only need one seek to locate the first block in the file. The rest is easy.

The bad

- Disk becomes fragmented over time

# Recap : Contiguous File Allocation

- Access method suits sequential and direct access
- Directory table maps files into starting physical address and length
- Easy to recover in event of system crash
- Fast, often requires no head movement and when it does, head only moves one track
- Each file occupies a set of contiguous blocks on the disk.
- Allocation using first fit / best fit.
- A Need for compaction.
- Only starting block and length of file in blocks are needed to work with the file.
- Problems with files that grow.

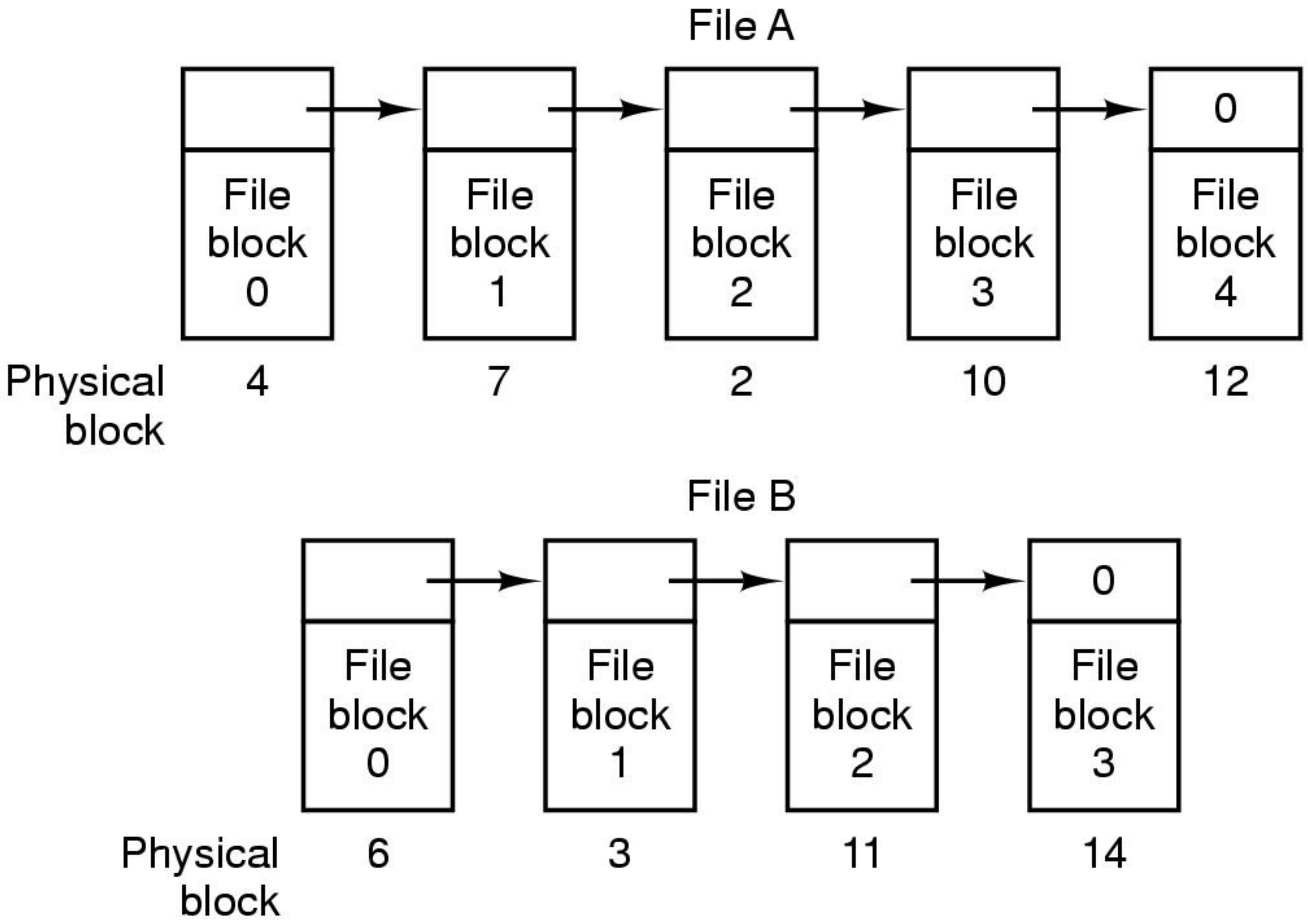
# Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
  - Contiguous chunk of space is allocated initially.
  - Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added.
- The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.
- A file consists of one or more extents
- Internal fragmentation - if the extents are too large
- External fragmentation - extents of varying sizes are allocated and deallocated.

# Allocation Methods - Linked

- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - Each block contains pointer to next block
  - No compaction, external fragmentation
    - Any free block on the free-space list can be used to satisfy a request.
  - Free space management system called when new block needed
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation
  - Locating a block can take many I/Os and disk seeks

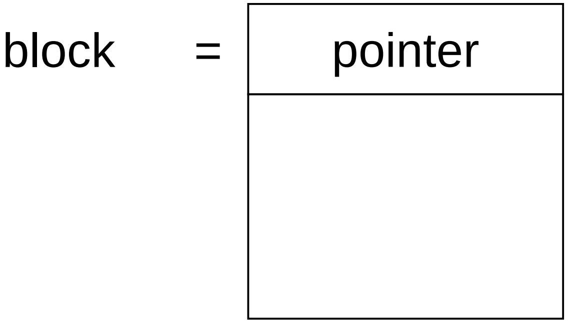
# Storing a file as a linked list of disk blocks



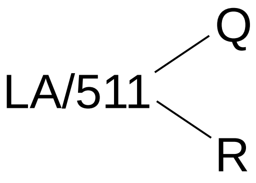
- To create a new file, create a new entry in the directory.
- With linked allocation, each directory entry has a pointer to the first disk block of the file.
- This pointer is initialized to null to signify an empty file. The size field is also set to 0.
- A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To read a file, read blocks by following the pointers from block to block.
- The size of a file need not be declared when the file is created.
- A file can continue to grow as long as free blocks are available.

# Linked Allocation

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



block = pointer



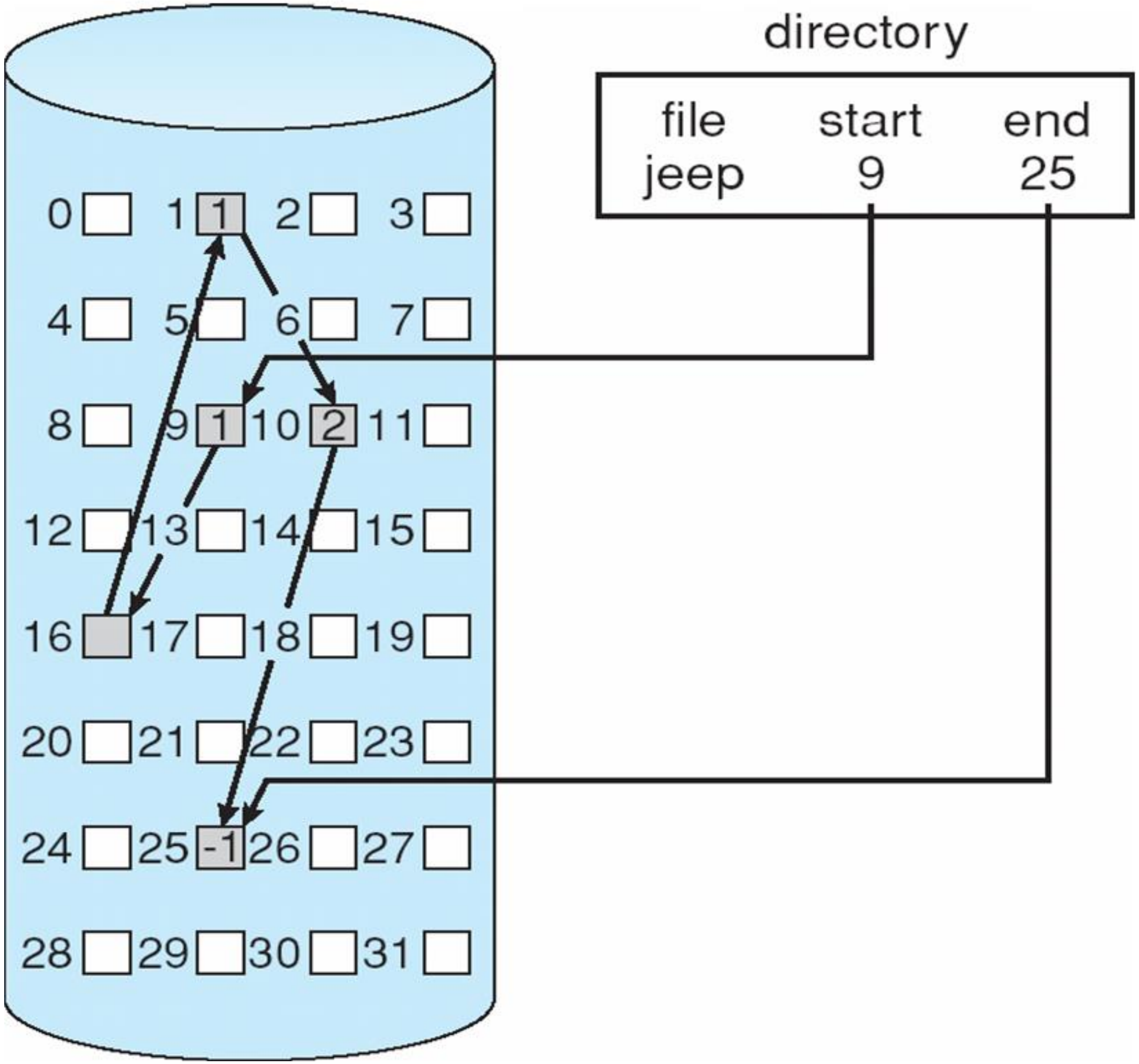
LA/511

Q

R

- Mapping
  - Block to be accessed is the Qth block in the linked chain of blocks representing the file.
  - Displacement into block = R + 1
- Each block contains a pointer to the next block. These pointers are not made available to the user.
  - If each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

# Linked Allocation



# Linked List

The good

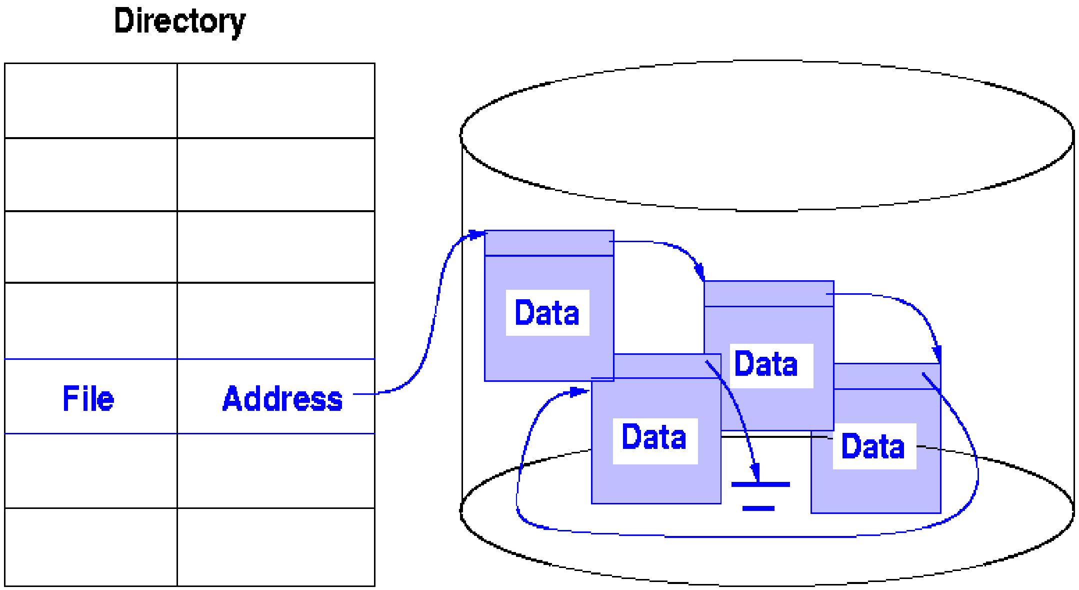
- Gets rid of fragmentation

The bad

- Random access is slow.
- Need to chase pointers to get to a block
- Space required for the pointers
- Reliability can be a problem
    - pointer were lost or damaged.
    - A bug in the OS software or a disk hardware failure might result in picking up the wrong pointer.
    - One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block.
    - More overhead for each file

# Recap : Linked File Allocation

- Each file is a linked list of blocks.
- No external fragmentation.
- Effective for sequential access.
- Problematic for direct access.

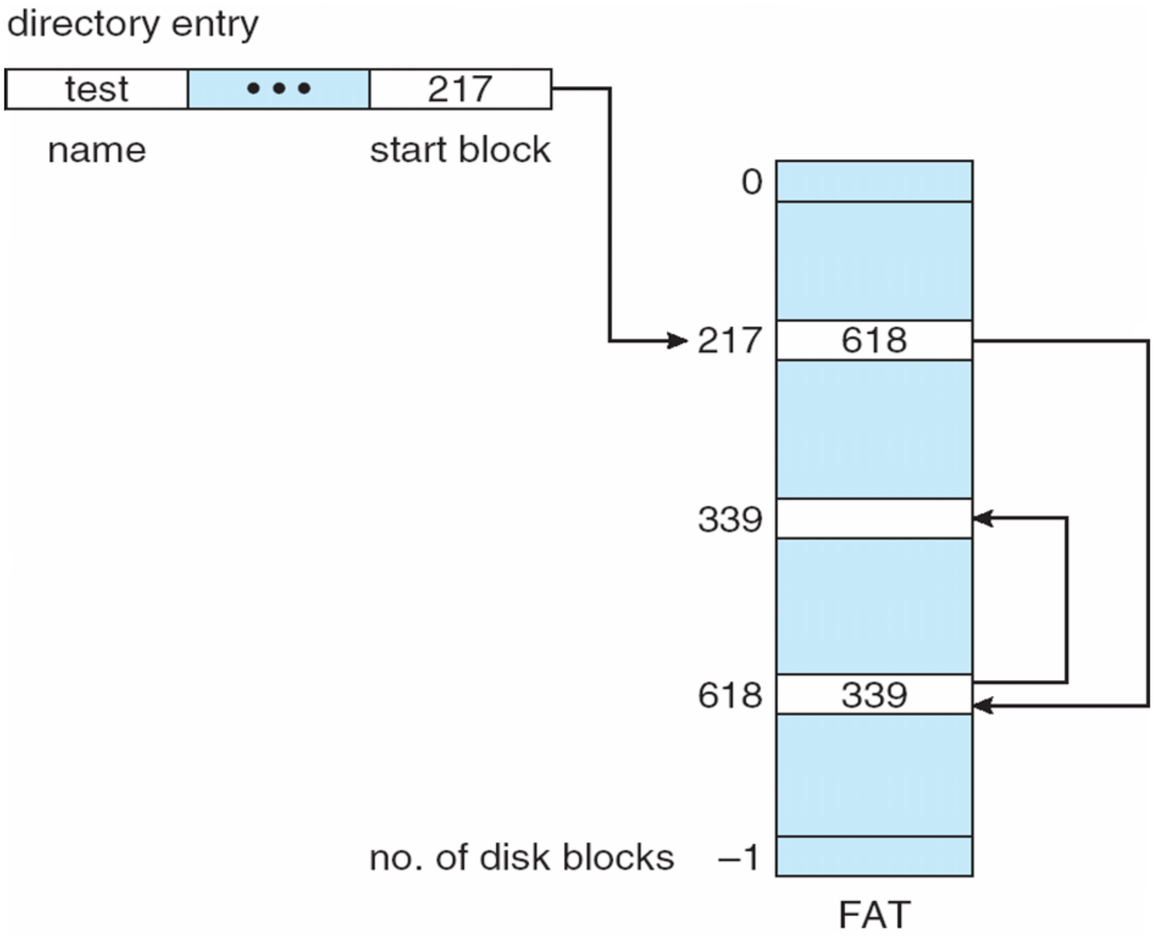


# FAT (File Allocation Table) variation

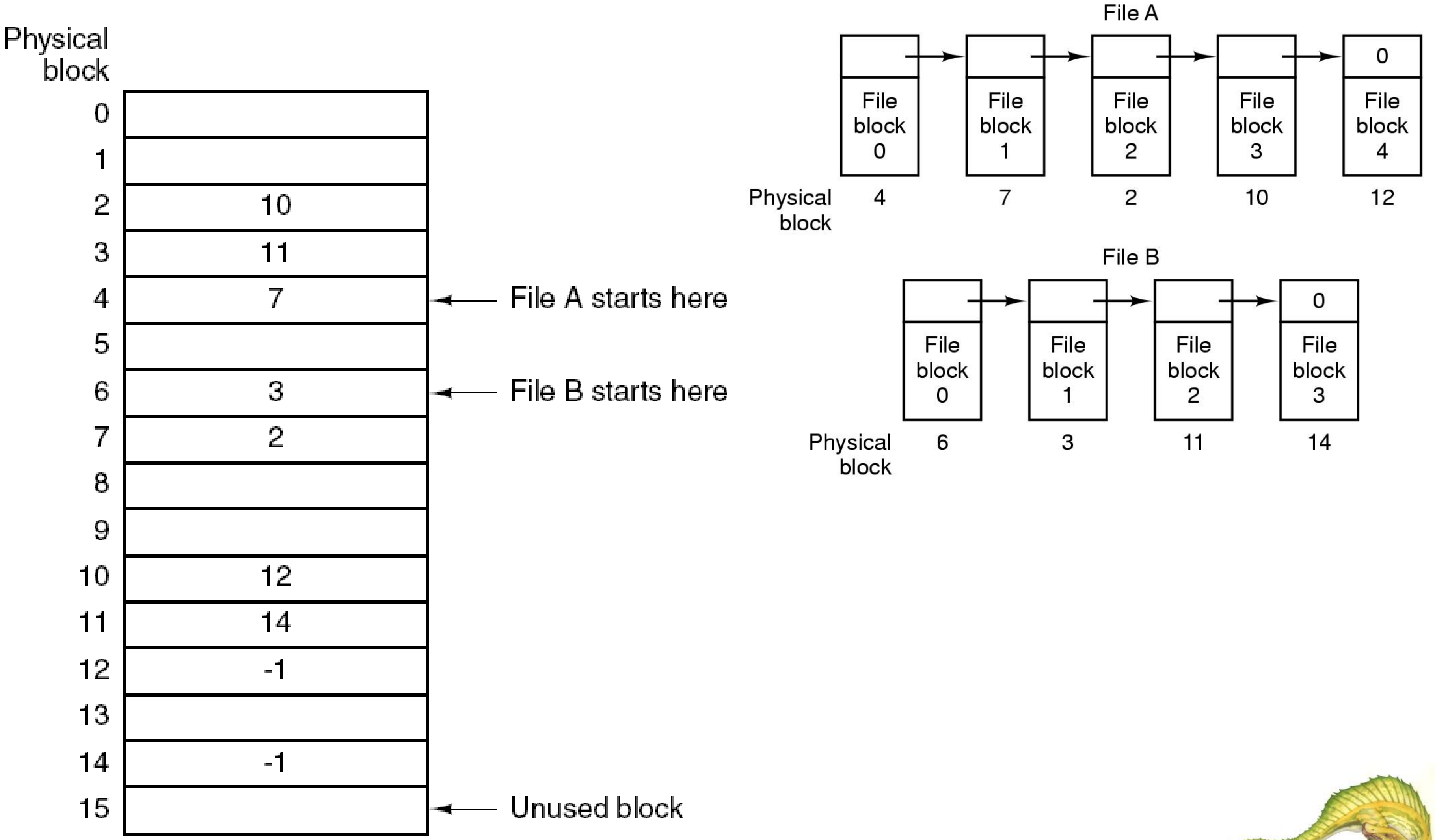
  - Much like a linked list, but faster on disk and cacheable

- A section of disk at the beginning of each volume is set aside to contain the table.
- The table has one entry for each disk block and is indexed by block number.
- The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.
- This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.
- An unused block is indicated by a table value of 0.
- Allocating a new block to a file- finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block.
- The 0 is then replaced with the end-of-file value.

# File-Allocation Table



# Linked list allocation using a file allocation table in main memory.



| Physical block | |
|---|---|
| 0 | |
| 1 | |
| 2 | 10 |
| 3 | 11 |
| 4 | 7 |
| 5 | |
| 6 | 3 |
| 7 | 2 |
| 8 | |
| 9 | |
| 10 | 12 |
| 11 | 14 |
| 12 | -1 |
| 13 | |
| 14 | -1 |
| 15 | |

# Recap : FAT

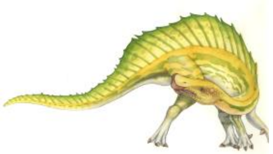
Variation of the linked list (MS/DOS and OS/2).

- A section of the disk at the beginning of each partition (Volume) is set aside to contain a FAT.
- FAT has one entry for each disk block, pointing to the next block in the file.
- The linklist is implemented in that section.
- Actual file blocks contain no links.

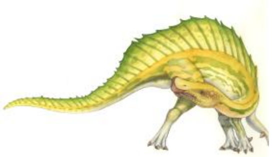
# Linked lists using a table in memory

The bad

- Table becomes really big
- Growth of the table size is linear with the growth of the disk size



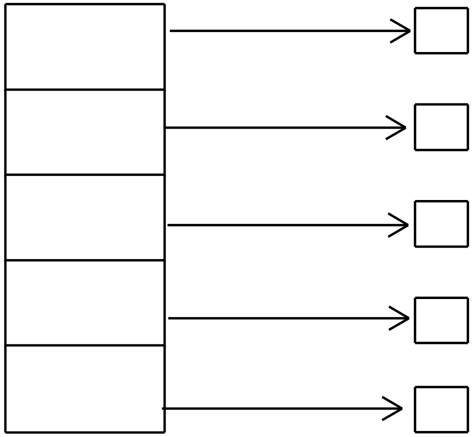
- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.



# Allocation Methods - Indexed

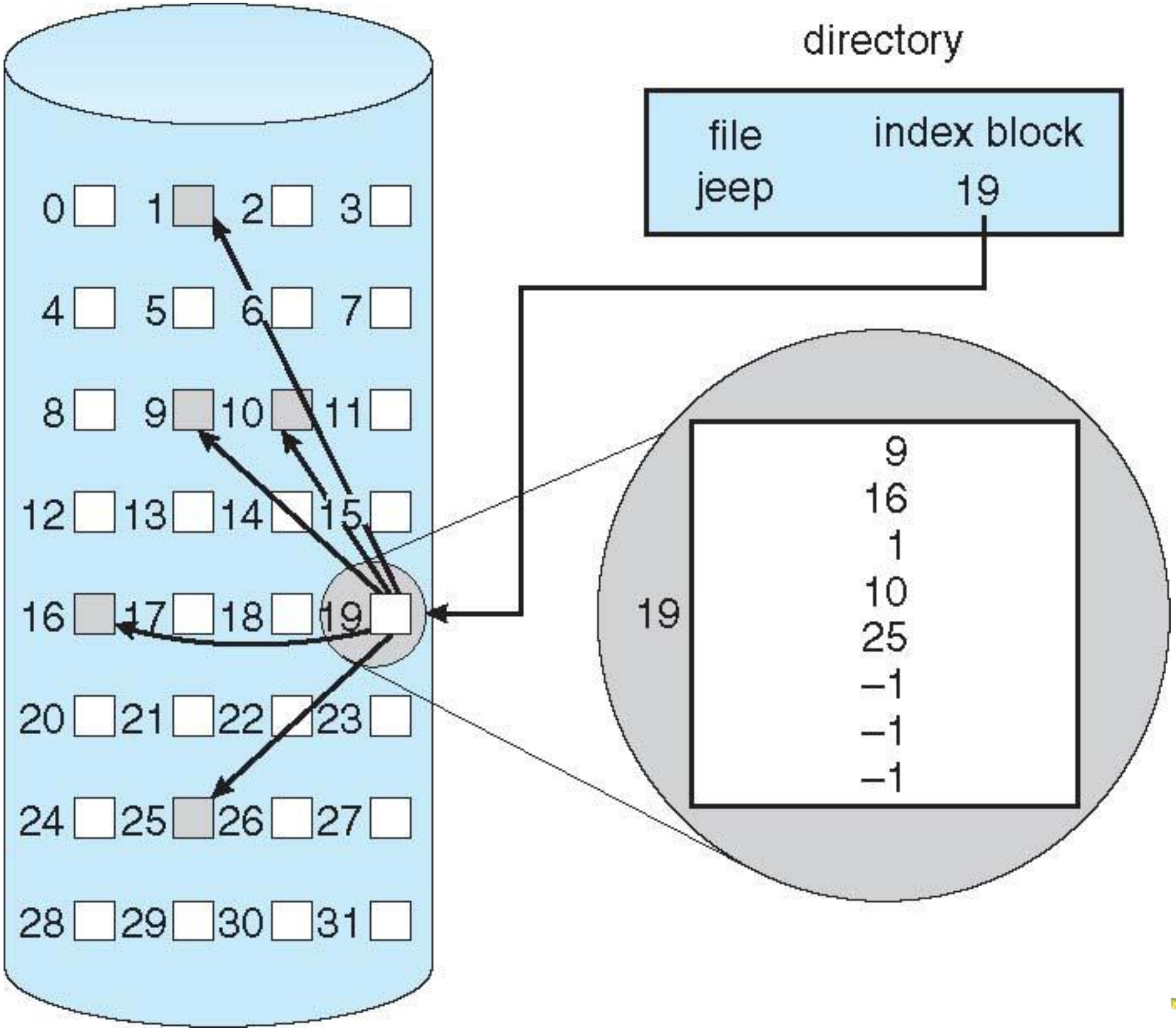
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks
    - Modification of linked allocation where the disk block pointers for a file are all placed in an index block.

- Logical view



index table

# Example of Indexed Allocation



directory

| file | index block |
|---|---|
| jeep | 19 |

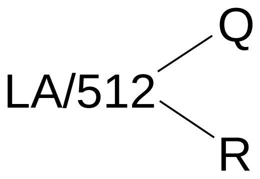
- When the file is created, all pointers in the index block are set to null.
- When the *ith* block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- Indexed allocation does suffer from wasted space, however.

- The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- Consider a file of only one or two blocks.
- With linked allocation, we lose the space of only one pointer per block.
- With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.
- How large the index block should be.
  - Every file must have an index block, so we want the index block to be as small as possible.
  - If the index block is too small, however, it will not be able to hold enough pointers for a large file

- Access requires at most one access to index block first. This can be cached in main memory
- File allocation table contains a separate one level index for each file
- The index has one entry for each portion allocated to the file
- File indexes are not stored as part of file allocation table
- File indexes are stored in separate block and the entry for the file in the file allocation table points to that block
- Allocation may be on basis of either fixed size block or variable size blocks

# 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 bytes and block size of 512 bytes. We need only 1 block for index table



LA/512

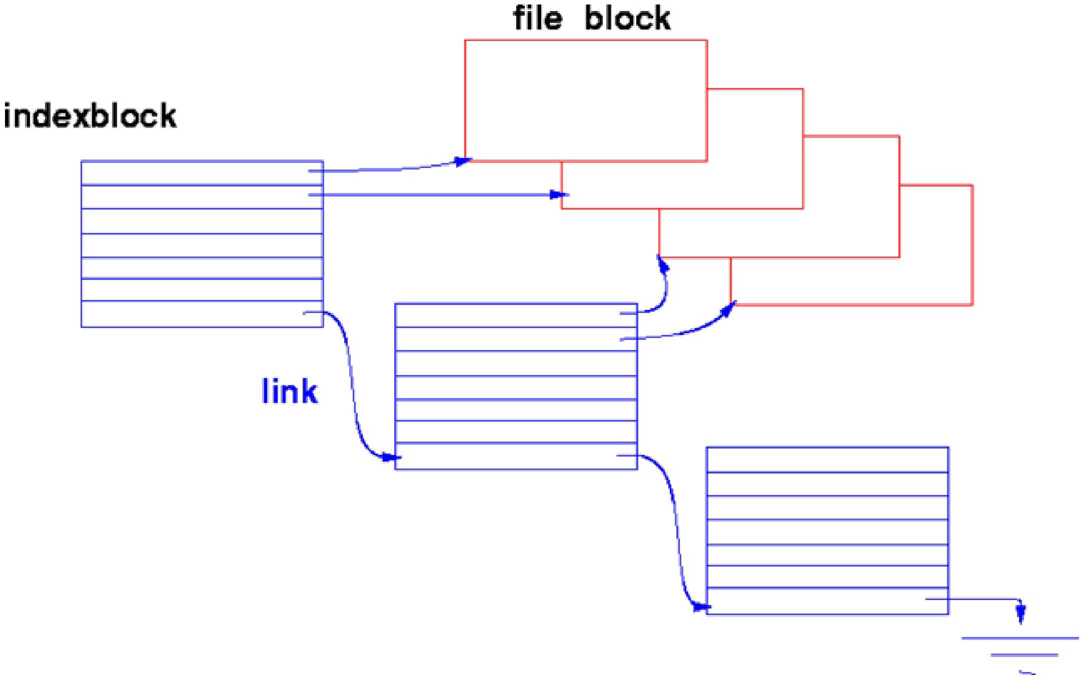
Q

R

Q = displacement into index table

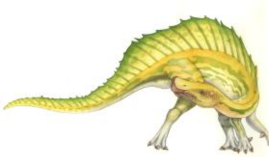
R = displacement into block

- Requires extra space for index block, possible wasted space
- How to extend to big files?
  - A file can be extended by using linked indexed files or multilevel indexed files
  - Link full index blocks together using last entry.



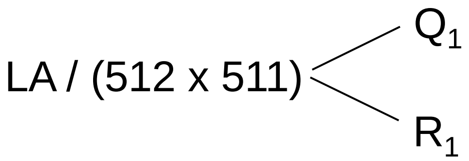
# Linked scheme

- An index block is normally one disk block.
- Thus, it can be read and written directly by itself.
- To allow for large files, we can link together several index blocks.
- For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.
- The next address (the last word in the index block) is null (for a small file) or is a pointer to another index block (for a large file).



# 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)

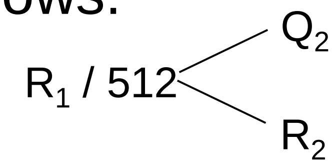


LA / (512 x 511)

$Q_1$

$R_1$

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

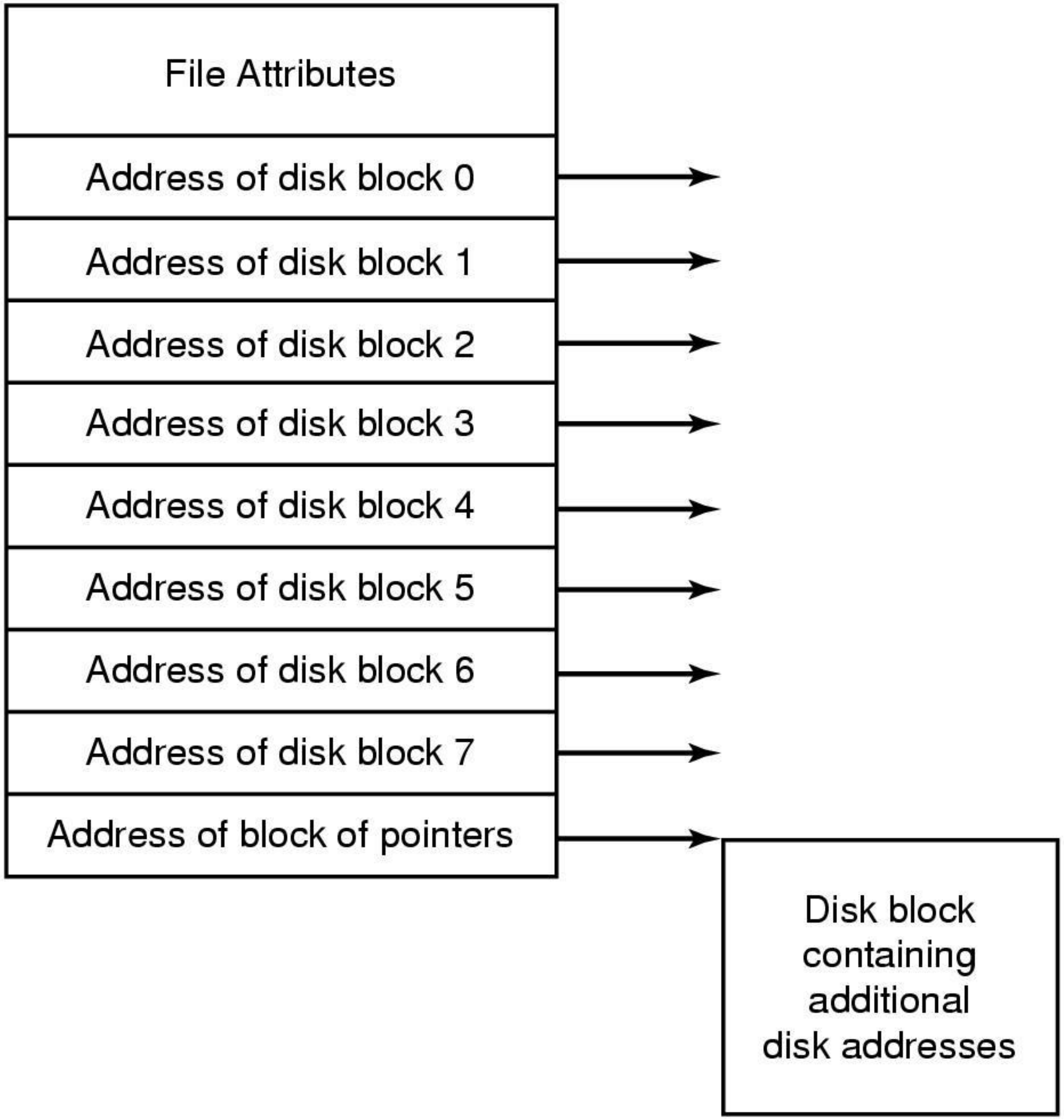


$R_1$ / 512

$Q_2$

$R_2$

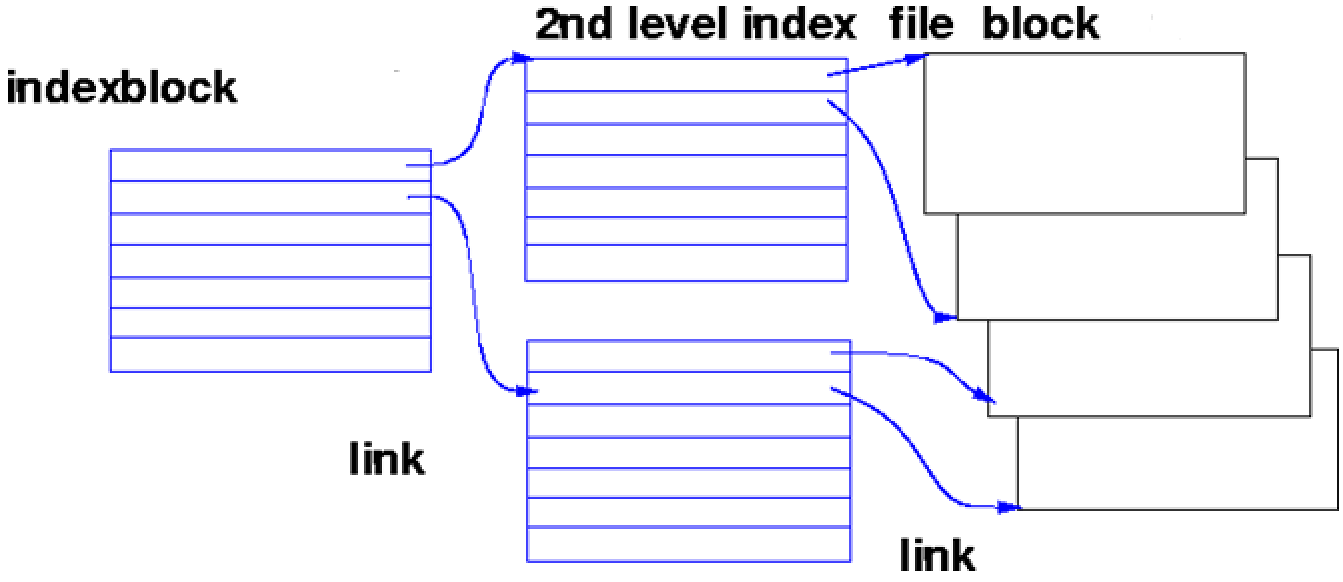
$Q_2$ = displacement into block of index table
$R_2$ displacement into block of file:


File Attributes
Address of disk block 0
Address of disk block 1
Address of disk block 2
Address of disk block 3
Address of disk block 4
Address of disk block 5
Address of disk block 6
Address of disk block 7
Address of block of pointers
Disk block containing additional disk addresses

# Multilevel index

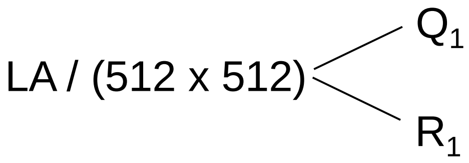
- A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.
- To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
- This approach could be continued to a third or fourth level, depending on the desired maximum file size.
- With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block.
- Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

# Multilevel Indexed File

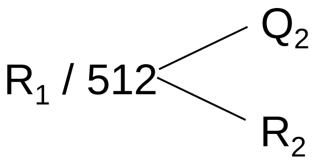


# Indexed Allocation – Mapping (Cont.)

- Two-level index (4K blocks could store 1,024 four-byte pointers in outer index -> 1,048,567 data blocks and file size of up to 4GB)

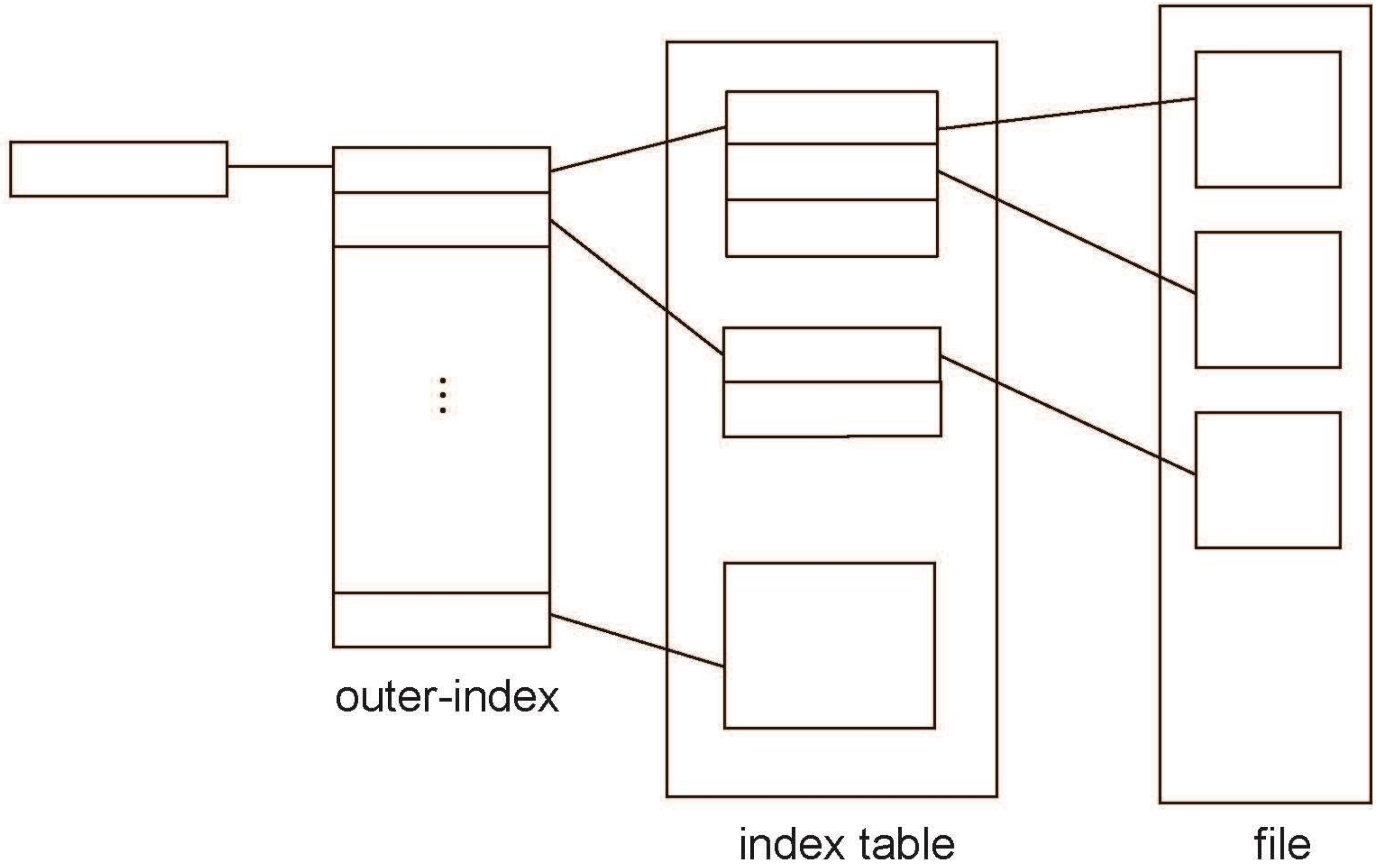


$Q_1$ = displacement into outer-index
$R_1$ is used as follows:



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

# Indexed Allocation – Mapping (Cont.)



# Combined scheme

- Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file’s inode.
- The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.
- Thus, the data for small files (of no more than 12 blocks) do not need a separate index block.
- If the block size is 4 KB, then up to 48 KB of data can be accessed directly.
- The next three pointers point to **indirect blocks**.
- The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.
- The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
- The last pointer contains the address of a **triple indirect block**.

# Layout of a UNIX file

- File allocation is done on a block basis and allocation is dynamic (as needed).

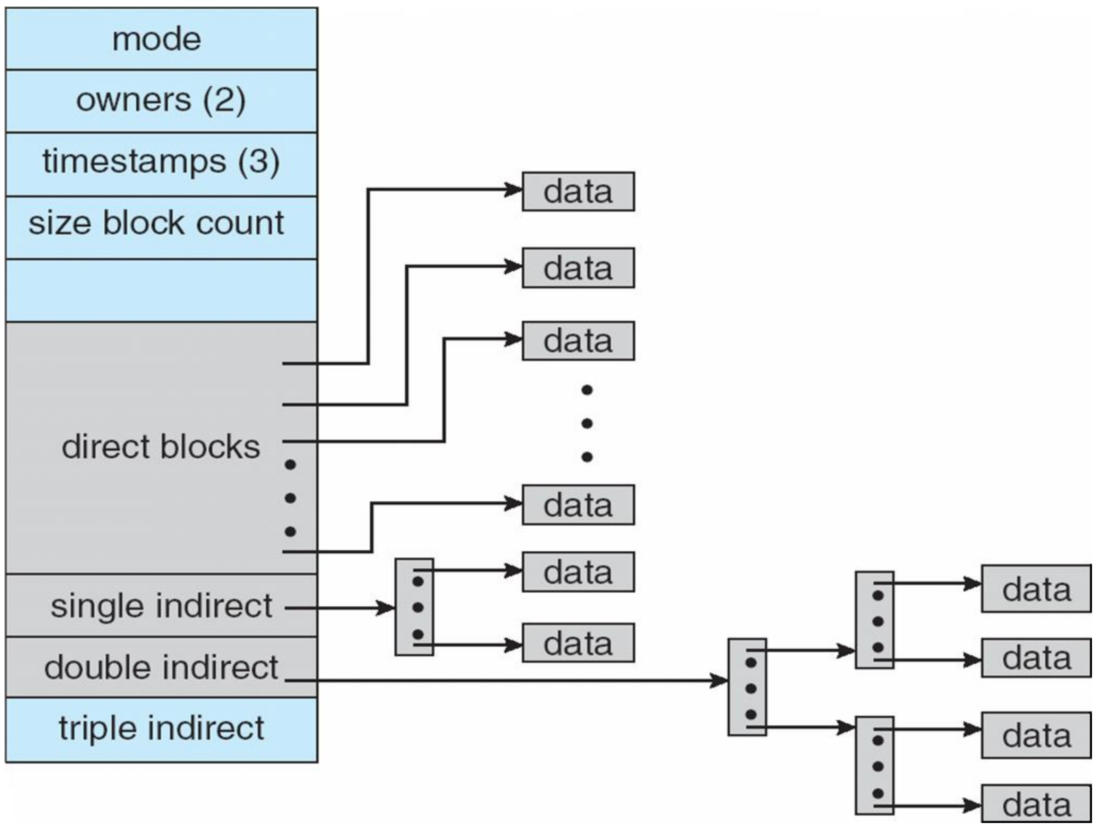
- UNIX uses a multilevel indexing mechanism for file allocation on disk.
- Addresses of first 10 data blocks + 3 index blocks (first, second, and third level of indexing)

- In UNIX System V the length of a block is 1 Kbyte and each block can hold a total of 256 block addresses

- According to above parameters, maximum size for a file is slightly over 16Gbytes

# Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses

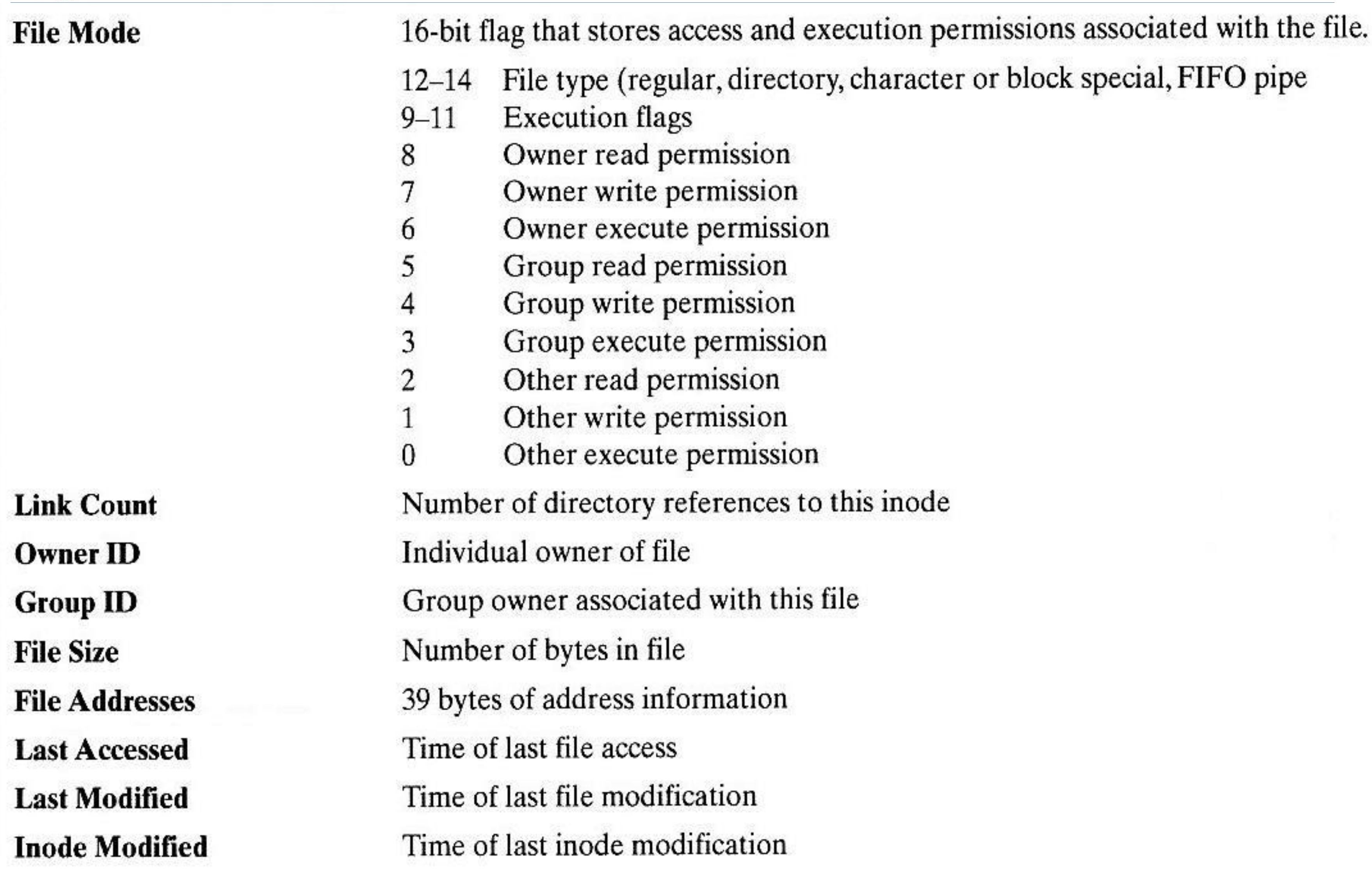


More index blocks than can be addressed with 32-bit file pointer

# What is an inode?

- An inode (index node) is a control structure that contains key information needed by the OS to access a particular file.
- Several file names may be associated with a single inode, but each file is controlled by exactly ONE inode.
- On the disk, there is an inode table that contains the inodes of all the files in the filesystem.
- When a file is opened, its inode is brought into main memory and stored in a memory-resident inode table.

# Information in the inode



| | | |
|---|---|---|
| **File Mode** | 16-bit flag that stores access and execution permissions associated with the file. | |
| | 12–14 | File type (regular, directory, character or block special, FIFO pipe |
| | 9–11 | Execution flags |
| | 8 | Owner read permission |
| | 7 | Owner write permission |
| | 6 | Owner execute permission |
| | 5 | Group read permission |
| | 4 | Group write permission |
| | 3 | Group execute permission |
| | 2 | Other read permission |
| | 1 | Other write permission |
| | 0 | Other execute permission |
| **Link Count** | Number of directory references to this inode | |
| **Owner ID** | Individual owner of file | |
| **Group ID** | Group owner associated with this file | |
| **File Size** | Number of bytes in file | |
| **File Addresses** | 39 bytes of address information | |
| **Last Accessed** | Time of last file access | |
| **Last Modified** | Time of last file modification | |
| **Inode Modified** | Time of last inode modification | |

# 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