Chapter 11 File-System Implementation

LI Wensheng, SCS, BUPT

Strong points:

File System Structure, File System Implementation

Directory Implementation

Allocation Methods, Free-Space Management

Chapter Objectives

- To describe the details of implementing local file systems and directory structures.
- To describe the implementation of remote file systems.
- To discuss block allocation and free-block algorithms and trade-offs.

Contents

11.1 File System Structure

11.2 File System Implementation

11.3 Directory Implementation

11.4 Allocation Methods

11.5 Free-Space Management

11.6 Efficiency and Performance（*）

11.7 Recovery（*）

11.8 Log-Structured File Systems（*）

11.9 NFS（*）

11.10 Example: The WAFL File System（*）

11.1 File-System Structure

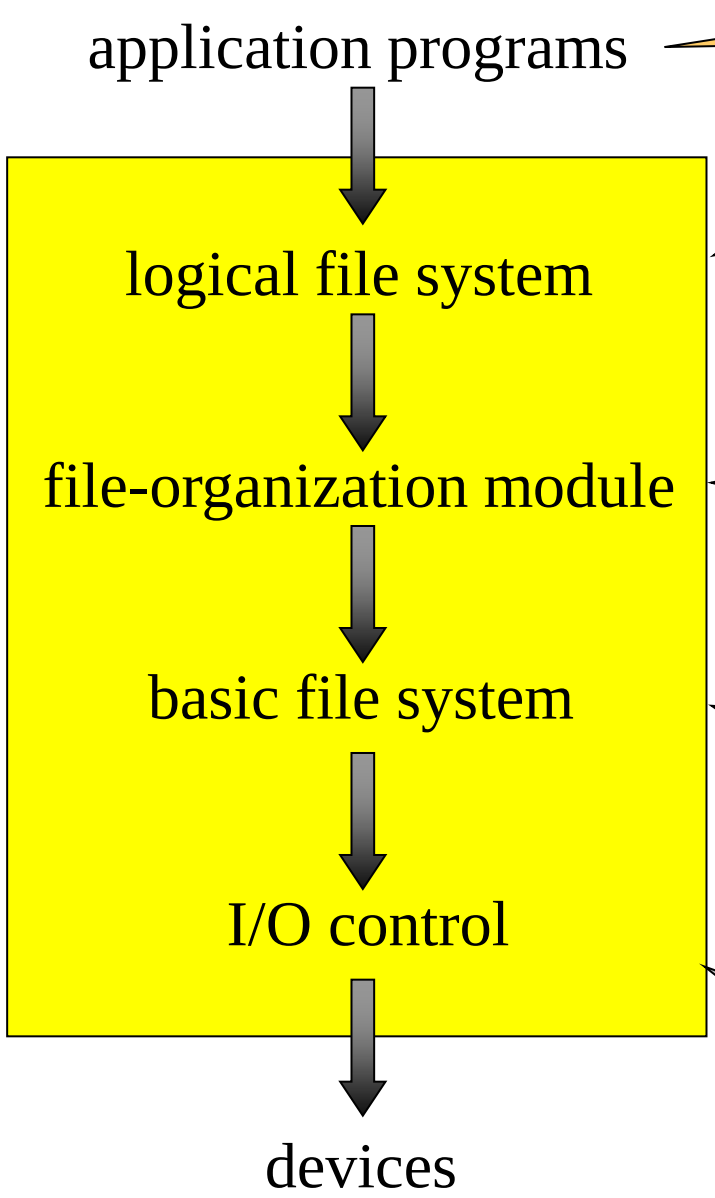
- File structure
  - Logical storage unit
  - Collection of related information
- Disk characteristics
  - A disk can be written in place.
  - A disk can access directly any given block of information.
  - I/O transfers between memory and disk in units of blocks.
- File system resides on secondary storage (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.

File-System Structure(Cont.)

- Main design problems
  - User Interface
    - Defining a file and its attributes
    - Operations on a file
    - Directory structure
  - Algorithms and data structure mapping the logical file system onto the physical secondary-storage device.
- File control block – storage structure consisting of information about a file.
- Device driver controls the physical device.
- File system organized into layers.

Layered File System

合理规划，功能清晰，责任明确
用户接口，硬件接口，分工合作



System calls for file operations

Manages metadata information,
Manages the directory structure,
Maintains file structure via FCB.
Responsible for protection and security.

Translate logical block addresses to physical block addresses.
Free-space manager.

Issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
I/O buffer, disk cache for directory / data.

Device driver and interrupt handlers.
Transfer information between the main memory and the disk system.

Layered File System(Cont.)

- **Logical file system**
  - Manages metadata information.
    - All of the file-system structure.
  - Translates file name into file number, file handle, location by maintaining file control blocks (inodes in UNIX)
  - Directory management
  - Protection
- **File organization module**
  - understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

Layered File System(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
- **Device drivers** manage I/O devices
  - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060”
  - outputs low-level hardware specific commands to hardware **controller**
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance.

Layered File System(Cont.)

- **Many file systems, sometimes many within an operating system**
  - **Each with its own format**
    - **CD-ROM is ISO 9660. standard format agreed by manufacturers.**
    - **Unix has UFS, based on the Berkeley FFS.**
    - **Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD.**
    - **Linux has more than 40 types.**
      Standard Linux file system -- extended file system.
      common ext3 and ext4, plus distributed file systems, etc.
  - **New ones still arriving**
    - **ZFS (Zettabyte File System), OpenZFS**
    - **Google GFS**
    - **Oracle ASM(Automatic Storage Management)**
    - **FUSE (Filesystem in User space), -- OMG(Omnicom Media Group)**

12.2 File System Implementation

- 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
  - individual files
- On-disk structures
  - A boot control block (per volume), typically 1st block of a volume.
    - In UFS -- boot block, In NTFS -- partition boot sector
  - A volume/partition control block (per volume)
    - Details, the number of blocks, block size, free-block count and free-block pointers, free-FCB count and free-FCB pointers.
    - In UFS -- superblock, In NTFS -- Master File Table

File System Implementation (Cont.)

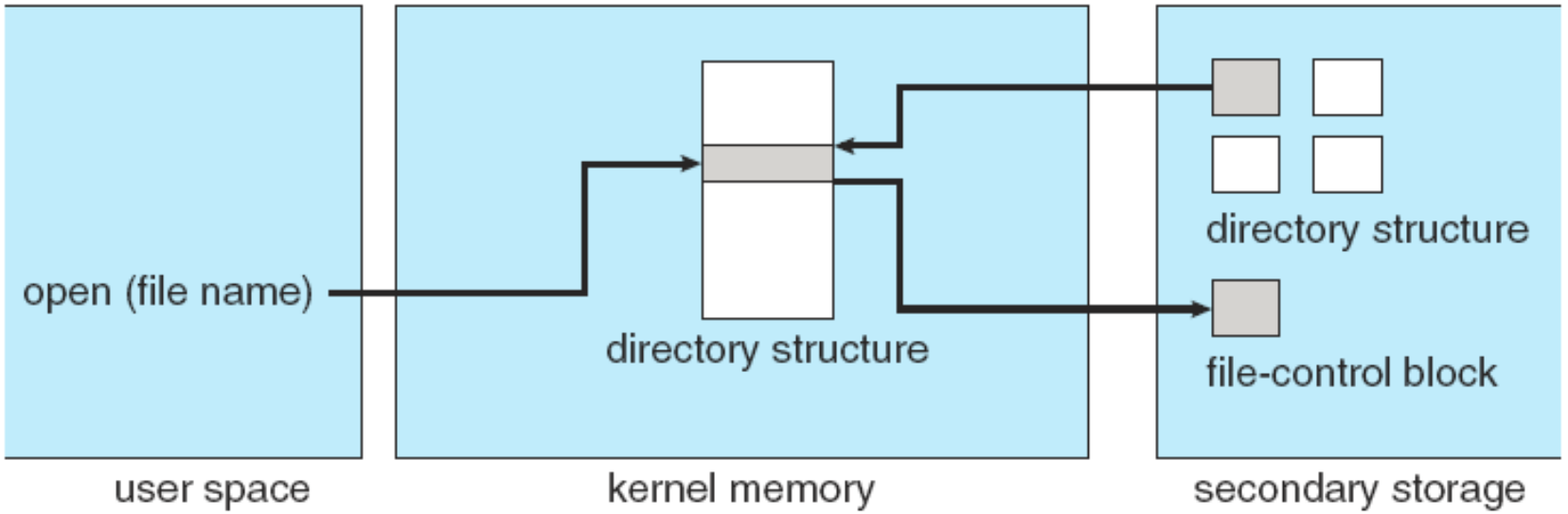
- On-disk structures(Cont.)
  - A directory structure (per file system)
    - In UFS -- file names and associated inode numbers
    - In NTFS -- in Master File Table
  - A FCB (per file)
    - storage structure consisting of information about a file, including ownership, permissions, and location of the file contents.
    - In UFS -- inode
      In NTFS -- Master File Table (relational database)
    - A Typical File-Control Block

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

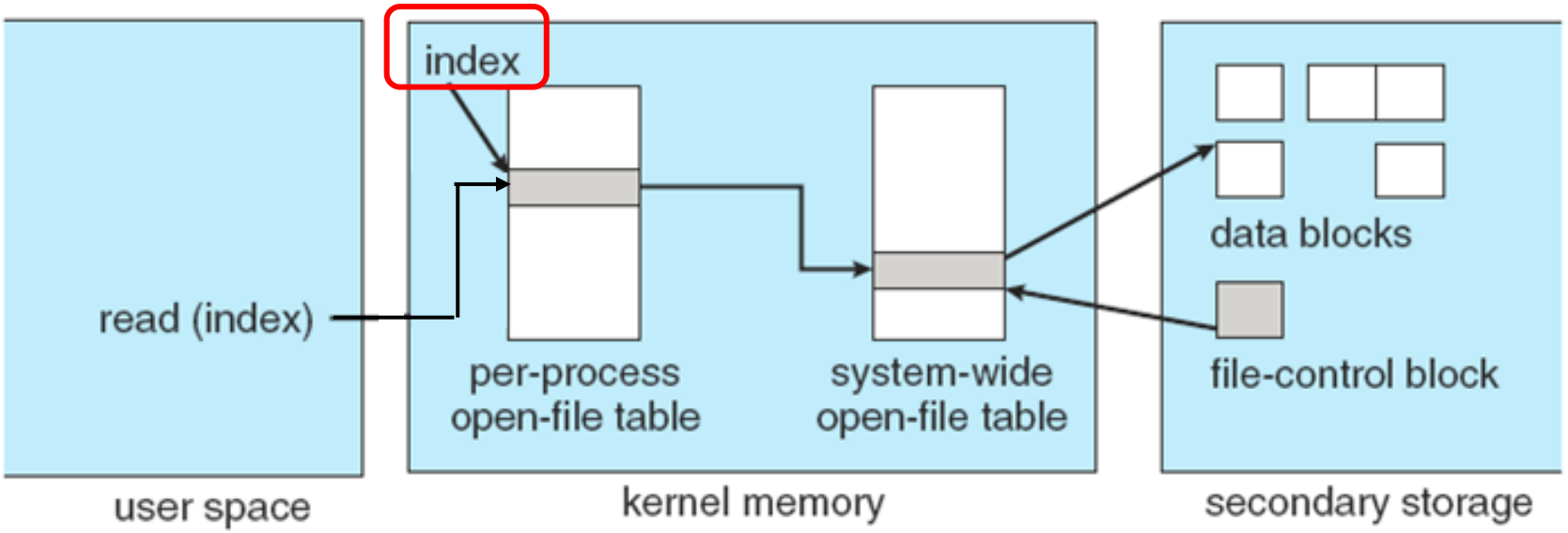
In-Memory File System Structures

- Used for:
  - file-system management
  - Performance improvement via caching
- In-memory structures
  - An in-memory mount table (partition table)
    - storing file system mounts, mount points, file system types.
  - An in-memory directory structure cache
  - The system-wide open-file table
    - Contains a copy of the FCB of each open file
    - One entry for each open file
    - Only one table for the system
  - The per-process open-file table
    - Contains a pointer to the appropriate entry in the system-wide open-file table.
    - One entry for each open file
    - One table per process
  - buffers hold data blocks from secondary storage.

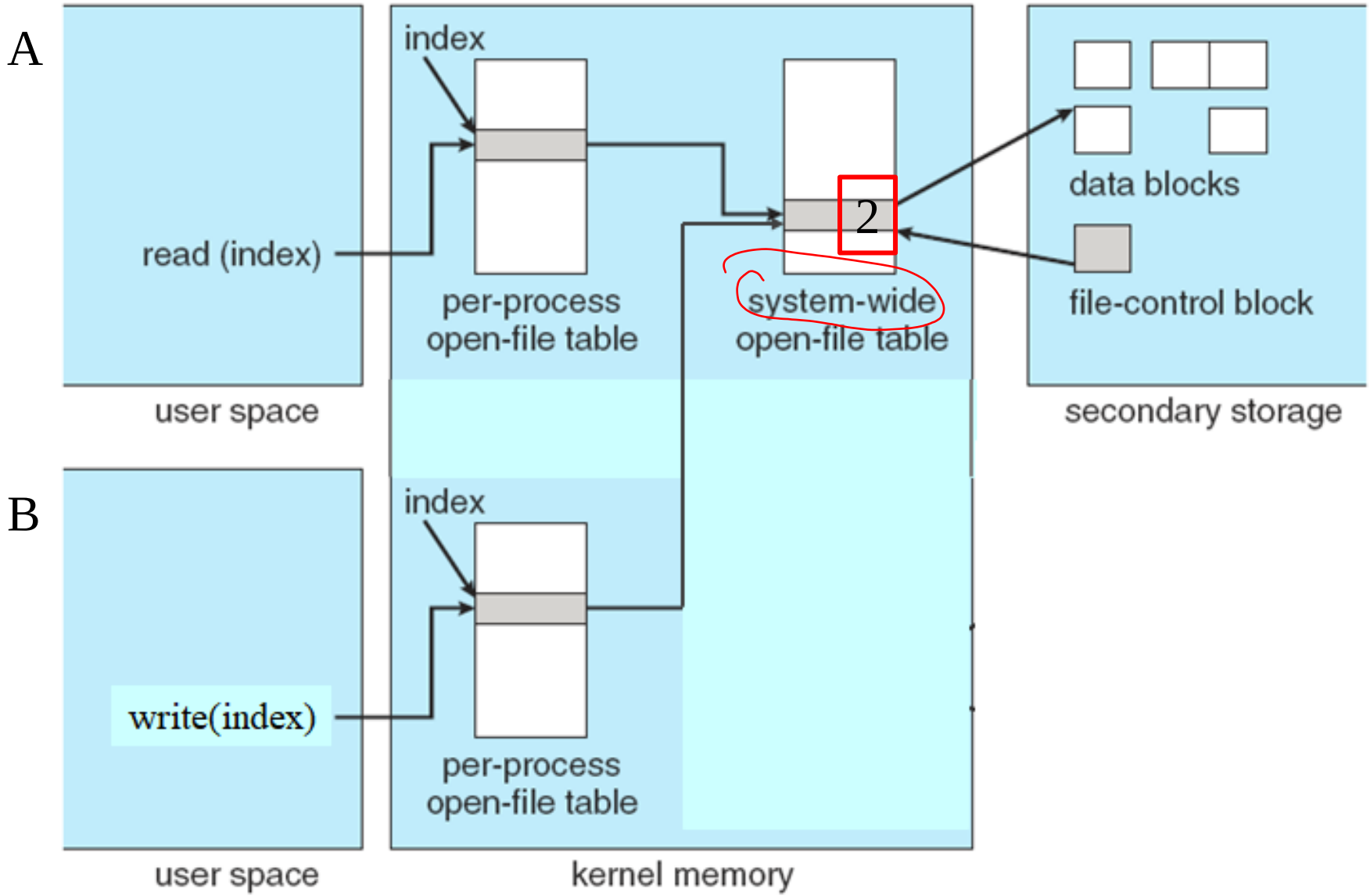
In-Memory File System Structures



A



In-Memory File System Structures（Cont.）



Partitions and Mounting

- A disk can be sliced into multiple partitions, or a partition can span multiple disks.
- Each partition can either be
  - "raw": containing no file system, just a sequence of blocks.
  - "cooked": containing a file system.
- Boot information can be stored in a separate partition, it has its own format, and is loaded into memory as an image, starts at a predefined location.
- A boot loader
  - knows enough about the file-system structure to be able to find and load the kernel and start it executing;
  - understands multiple file systems and multiple operating systems can occupy the boot space.

Partitions and Mounting(Cont.)

- The root partition, which contains the operating-system kernel and potentially other system files, is mounted at boot time.
  - Other partitions can be automatically mounted at boot, or manually mounted later.
- At mount time, file system consistency checked
  - By asking the device driver to read the directory and verifying that the directory has the expected format.
  - Is all metadata correct?
    - If not, fix it, try again.
    - If yes, add to mount table, allow access.
- Mount table – containing information about file systems that has been mounted.

Virtual File Systems

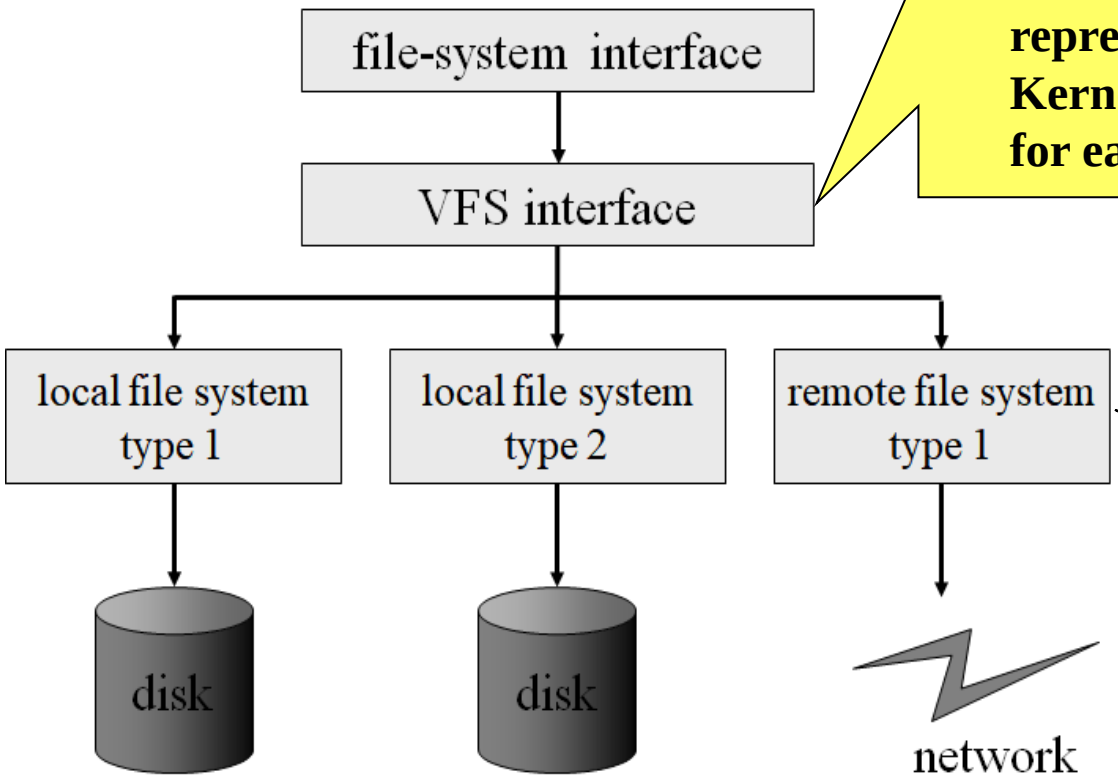
- modern OSes must concurrently support multiple types of file systems.
  - how does an operating system allow multiple types of file systems to be integrated into a directory structure?
  - how can users seamlessly move between file-system types as they navigate the file-system space?
- Virtual file system (VFS) on most OSes, including Unix, use object-oriented techniques to implement the file system.
  - Allow dissimilar file-system types to be implemented within the same structure, including network file systems.
  - simplify, organize, and modularize the implementation.
  - Data structures and procedures are used to isolate the basic system-call functionality from the implementation details.

Virtual File Systems(Cont.)

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

Virtual File Systems

based on the open(), read(), write() and close() calls and on file descriptors.



Two important functions:

1. **Separates file-system-generic operations from their implementation.**
2. **Provides a mechanism for uniquely representing a file throughout a network. Based on Vnode, a file-representation structure.**
   Kernel maintains one vnode structure for each active node.

Implementing the file-system type or the remote-file-system protocol

Virtual File Systems(Cont.)

- For example, Linux has four object types:
  - inode, file, superblock, dentry
- VFS defines a 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. e.g.
    - 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

11.3 Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
    - Linear search time
  - A **sorted list** allows a binary search and decreases the average search time.
  - A balanced tree
- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
    - Chained-overflow
  - Difficulties: fixed size, hash function

11.4 Allocation Methods

- An allocation method refers to how disk blocks are allocated for files.
  - Disk space is utilized effectively.
  - Files can be accessed quickly.

分配方式

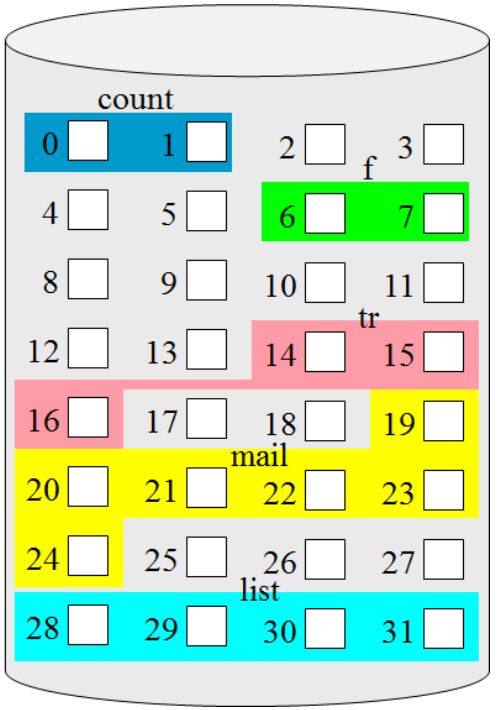
- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation

- Single set of blocks is allocated to a file at the time of creation
  - each file occupies a set of contiguous blocks on the disk.
- Simple
  - Only a single entry in the file allocation table.
  - only starting location (block #) and length (number of blocks) are required.

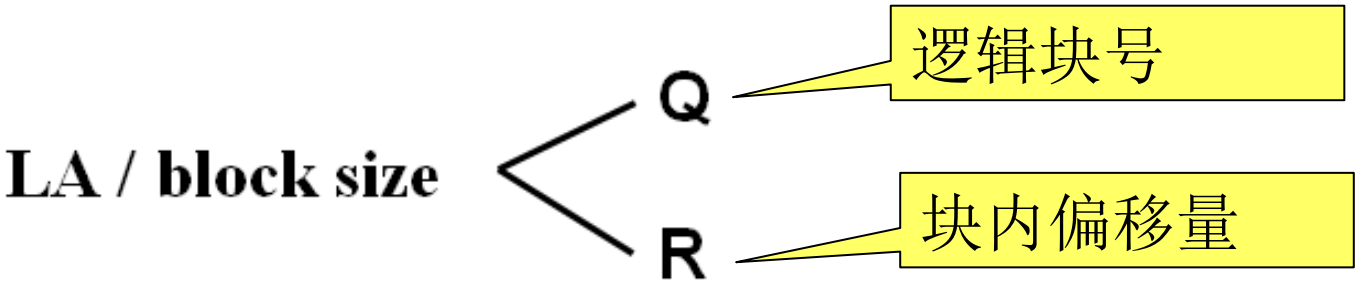
directory

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



- Both sequential and direct access can be supported.
- Best performance in most cases.
  - The number of disk seeks required is minimal.
  - The seek time when a seek is finally needed is minimal.

* Mapping from logical to physical



- Block to be accessed = Q + starting address
- Displacement into block = R
- Example
  - For text file
  - For record file

* 连续文件举例

例如：某文件系统，磁盘块大小为 512 B

- 字符流文件 A，长度为 1980 B
  - 文件A需要占用 4 个物理块。
  - 假如分配到30、31 、32和33四个相邻的物理块中。
  - 第33块中实际使用了444字节，剩余的68字节形成“内部碎片”。
- 记录文件 B，逻辑记录长 100 B，有23个记录
  - 假设：逻辑记录不能跨物理块存放。
  - 每个物理块中可以存放5个逻辑记录，需要5个物理块。
  - 假设分配到第 6、7、8、9、10 块
  - 前4块各有内部碎片12B，最后一块有212B没有使用。

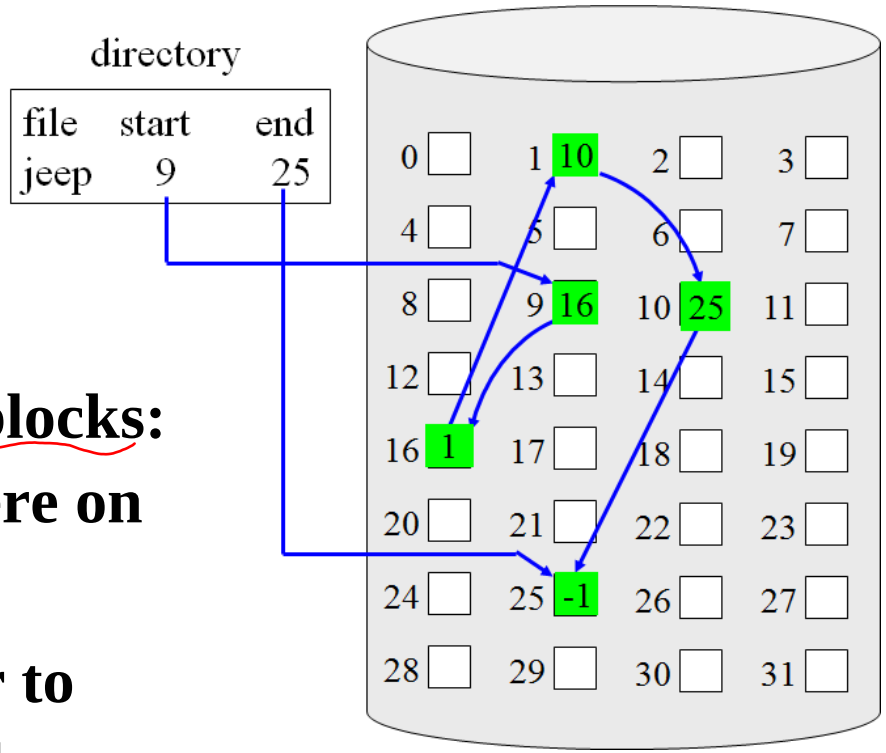
* 连续文件的地址转换

- 在文件说明信息中有关于存放位置的描述
  - 开始块号、总块数（物理文件的长度）
- 字符流文件中逻辑地址 L
  - L/物理块大小，商s：逻辑块号，余数d：块内地址
  - 物理地址：块号：开始块号+s，块内地址：d
- 记录文件中的逻辑记录号 n (记录不可跨块存放时)
  - 每个物理块中可以存放逻辑记录数 m
    - m=物理块大小/逻辑记录大小
  - n/m，商s：逻辑块号，余数w：块内记录号
  - 物理地址
    - 块号：开始块号+s，块内地址d=逻辑记录长度*w

Contiguous Allocation(Cont.)

- **Problems**
  - **Difficult to find space for a new file.**
    - **First-fit, best-fit**
  - **External fragmentation**
  - **Determine how much space is needed for a file**
    - **Initial size, file grows (find a larger hole, copy file, release previous space)**
    - **Knowing the final total amount of space**
  - **compaction off-line (downtime) or on-line**
- **Extent-Based Systems, a modified contiguous allocation scheme**
  - **An extent is a contiguous block of disks.**
  - **Extent-based file systems allocate disk blocks in extents.**
  - **A contiguous chunk of space is allocated initially, when that amount is not large enough, an extent is added.**
  - **A file consists of one or more extents.**

Linked Allocation

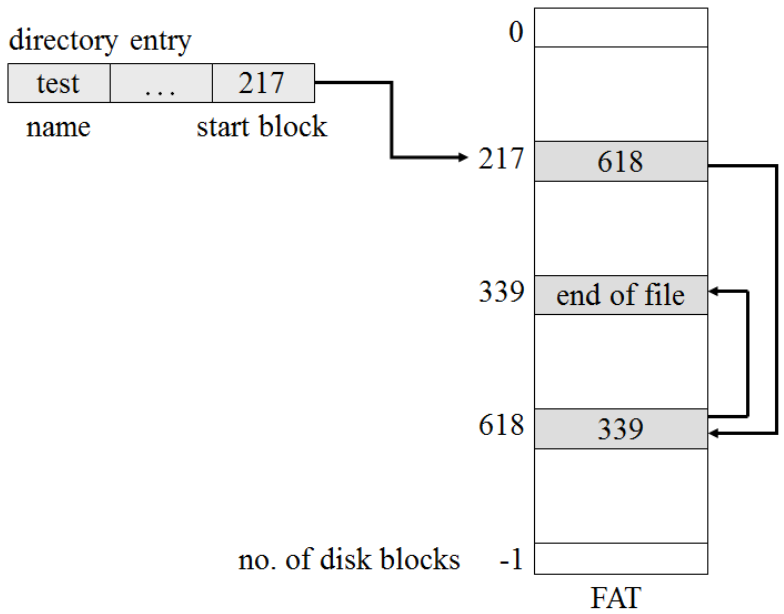


- Allocation on basis of individual block.
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.
- Each block contains a pointer to the next block in the chain.
- There is no external fragmentation.
- Any free block on the free-space list can be used to satisfy a request.
- **Disadvantages:**
  No random access, Space for pointers, Reliability

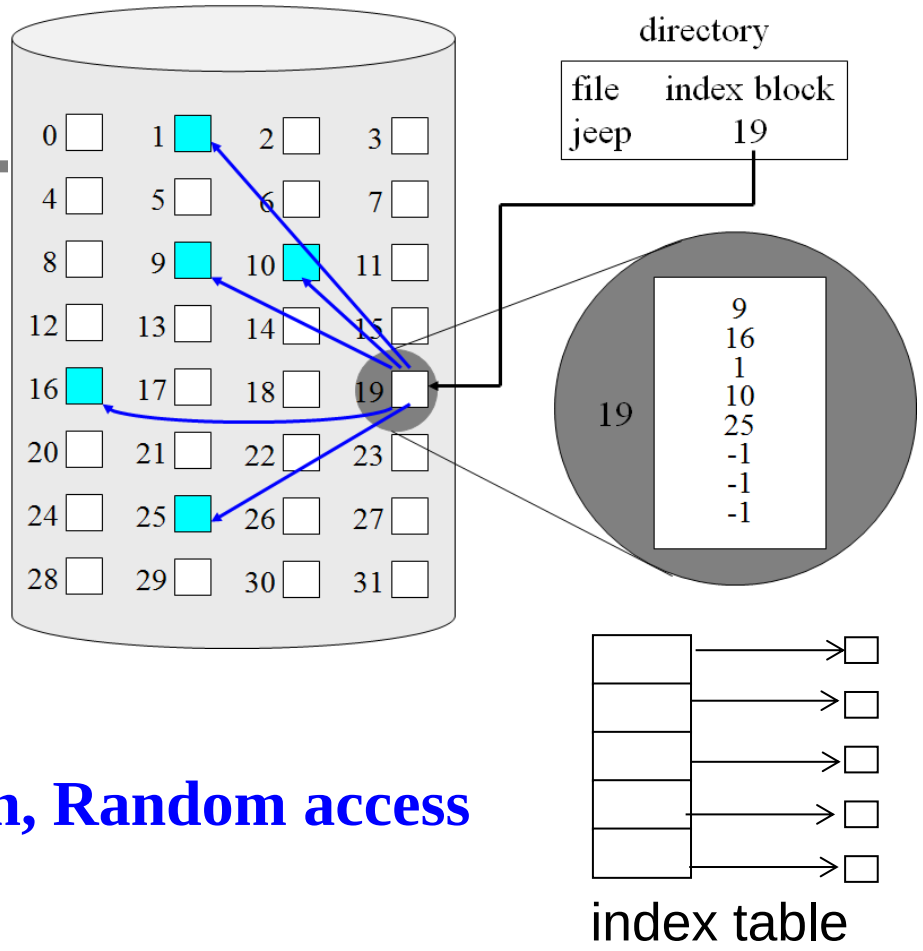
Linked Allocation (Cont.)

LA / (block_size - ptr_space) → Q, R

- Address mapping
  - Block to be accessed is the Qth block in the linked chain of blocks representing the file. 逻辑块号
  - Displacement into block = R + ptr_space 块内偏移
- Locating a block can take many I/Os and disk seeks.
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- File-allocation table (FAT)
  - Simple but efficient method used by MS-DOS and OS/2
  - Contained in a section at the beginning of each partition.
  - One entry per block.
  - Indexed by block number.
  - Cached in memory



Indexed Allocation

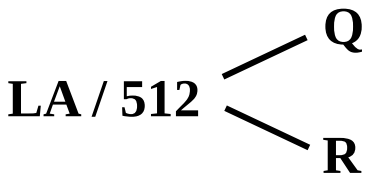


- Brings all pointers together into the index block.
- Each file has its own index block, which is an array of disk-block addresses.
- The directory contains block number for the index block.
- without external fragmentation, Random access
- Need index table / index block
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.

index table

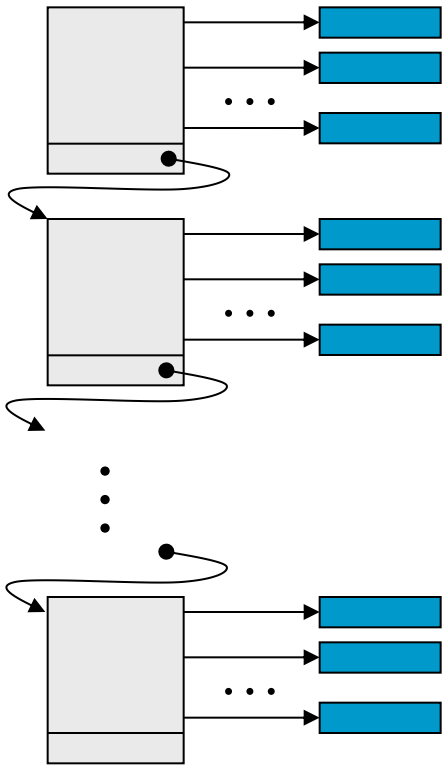
(1 word per block #)

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).



$LA / 512 \Rightarrow Q_1, R_1$

Q_1：逻辑块号

R_1：块内偏移

$Q_1 / 511 \Rightarrow Q_2, R_2$

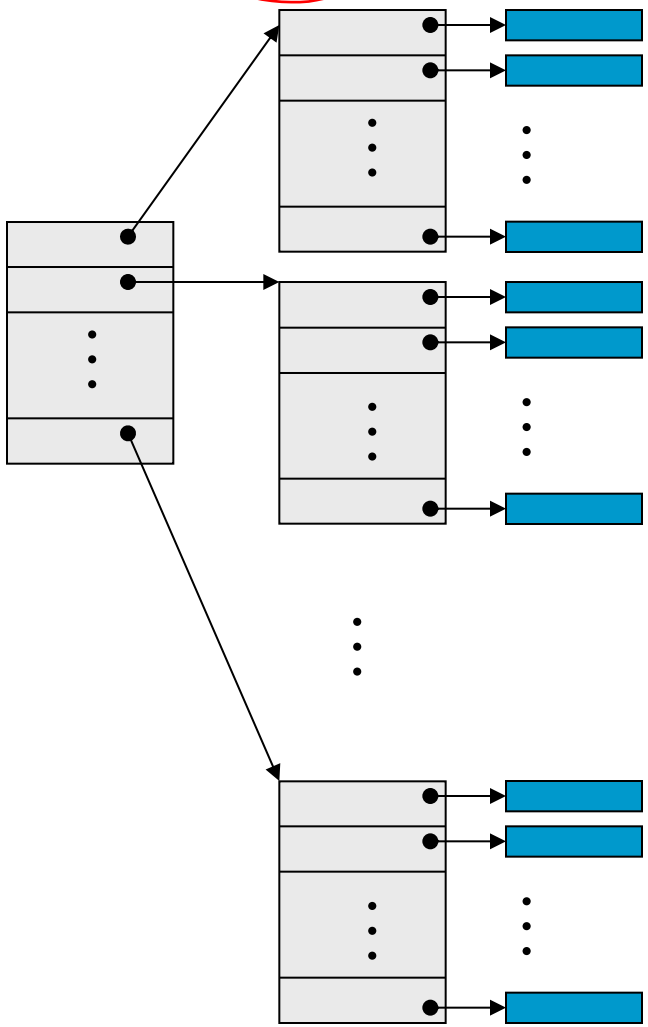
Q_2：索引块序号

R_2：索引块内偏移

根据 Q_2 在索引块链中找到索引块；
根据 R_2 在索引块内偏移，取出的就是相应于 Q_1 的物理块号。

Indexed Allocation – Mapping (Cont.)

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



$LA / 512 \Rightarrow Q_1, R_1$

Q_1 Q_1：逻辑块号

R_1 R_1：块内偏移

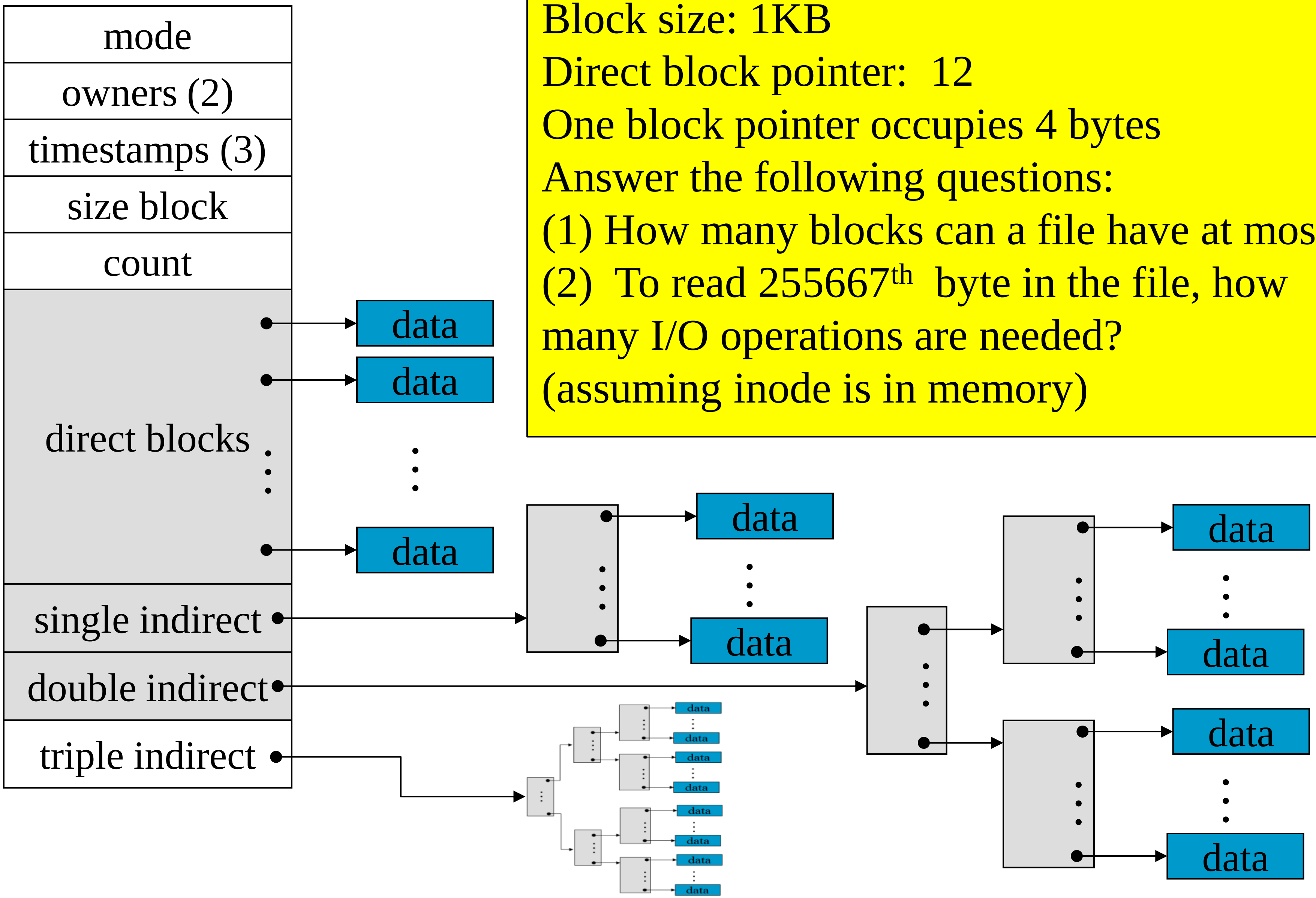
$Q_1 / 512 \Rightarrow Q_2, R_2$

Q_2 Q_2：索引块序号

R_2 R_2：索引块内偏移

根据 Q_2 在外索引块内偏移，取出的是索引块的物理块号；
根据 R_2 在该索引块内偏移，取出的是相应于 Q_1 的物理块号。

Combined Scheme: UNIX (4K bytes per block)



Block size: 1KB
Direct block pointer: 12
One block pointer occupies 4 bytes
Answer the following questions:
(1) How many blocks can a file have at most?
(2) To read 255667th byte in the file, how many I/O operations are needed?
(assuming inode is in memory)

Answer

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
  - Depends on the index structure, on the size of the file, and on the position of the block desired.
  - Single block access could require:
    - 2 I/Os, 1 for index block read, then 1 for data block read.
    - 3 I/Os, 2 for index block read, then 1 for data block read.
  - Clustering can help improve throughput, reduce CPU overhead
- combine contiguous allocation with indexed allocation by
  - using contiguous allocation for small files (up to three or four blocks)
  - automatically switching to an indexed allocation if the file grows large.

end 4

11.5 Free-Space Management

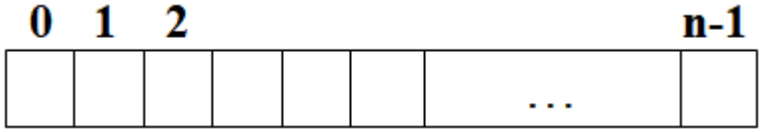
- File system maintains **free-space list** to track available blocks/clusters
  - **Free-space list** records all free disk blocks.

- Bit map / Bit vector
- Linked list
- Grouping
- Counting

空闲空间管理

Bit Map

- Each block is represented by 1 bit.
- For example, *n* blocks



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

word

bit

	31	30																											2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	1	1	0	0	1	0	1	1	0	0	1	1	0	1	0	1	1	1	1	1	1	1	1	0	0	1	1	0	0
3																															

- Calculation of the first free block number:
  (number of bits per word) *(number of 0-value words) + offset of first 1 bit
- CPUs have instructions to return offset within word of first "1" bit.

Bit Map(Cont.)

- **Bit map requires extra space.**
  - **Example:**

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

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

    n **= $2^{30}/2^{12}$ = 2^{18} bits (or 32K bytes)**

    256G ==> 8MB of memory

  - **Example:**

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

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

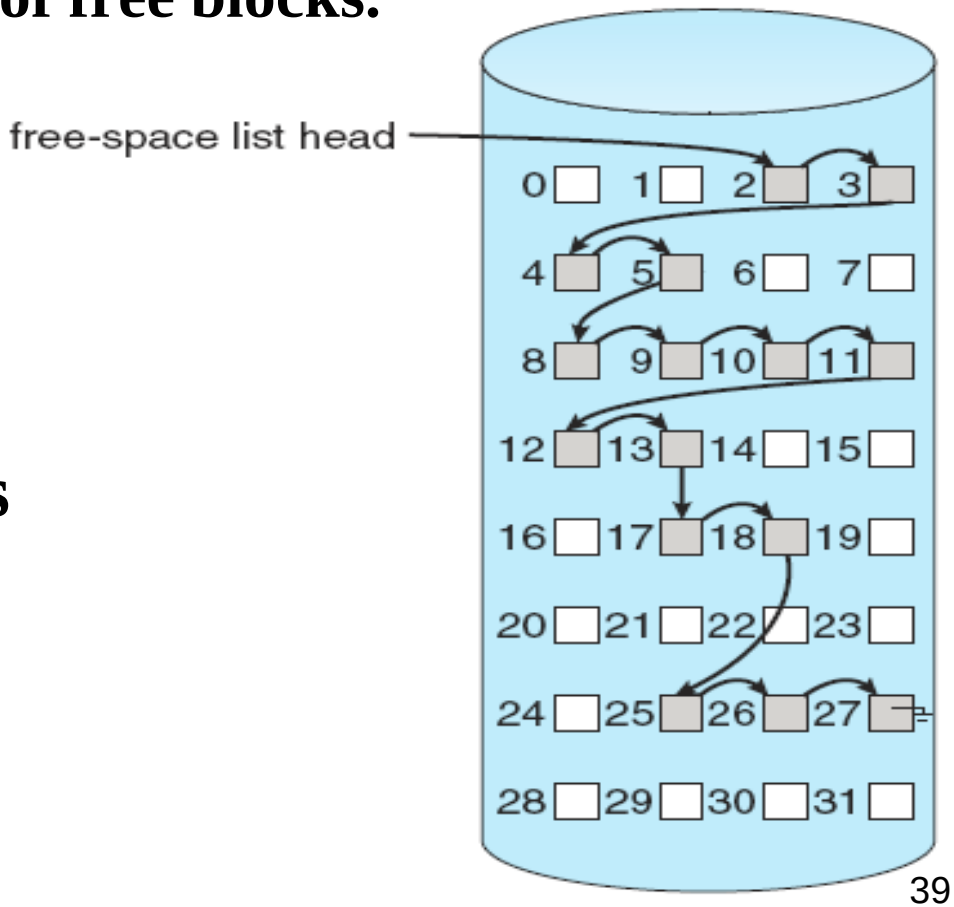
    n **= $2^{40}/2^{12}$ = 2^{28} bits (or 32MB)**

    if clusters of 4 blocks ==> 8MB of memory

  - **Easy to get contiguous files space.**

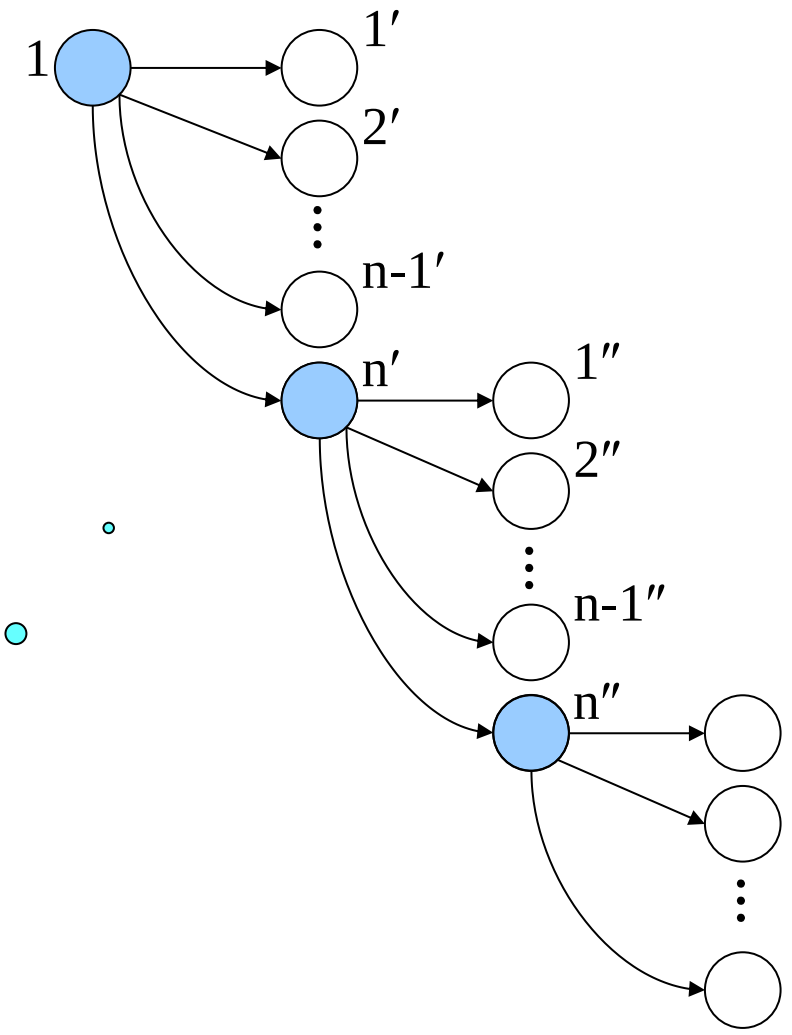
Linked List

- To link together all the free disk blocks.
  - keeping a pointer to the first free block in a special location on the disk and caching it in memory.
  - Records the total number of free blocks.
- No waste of space
- Cannot get contiguous space easily.
- No need to traverse the entire list (if # free blocks recorded).



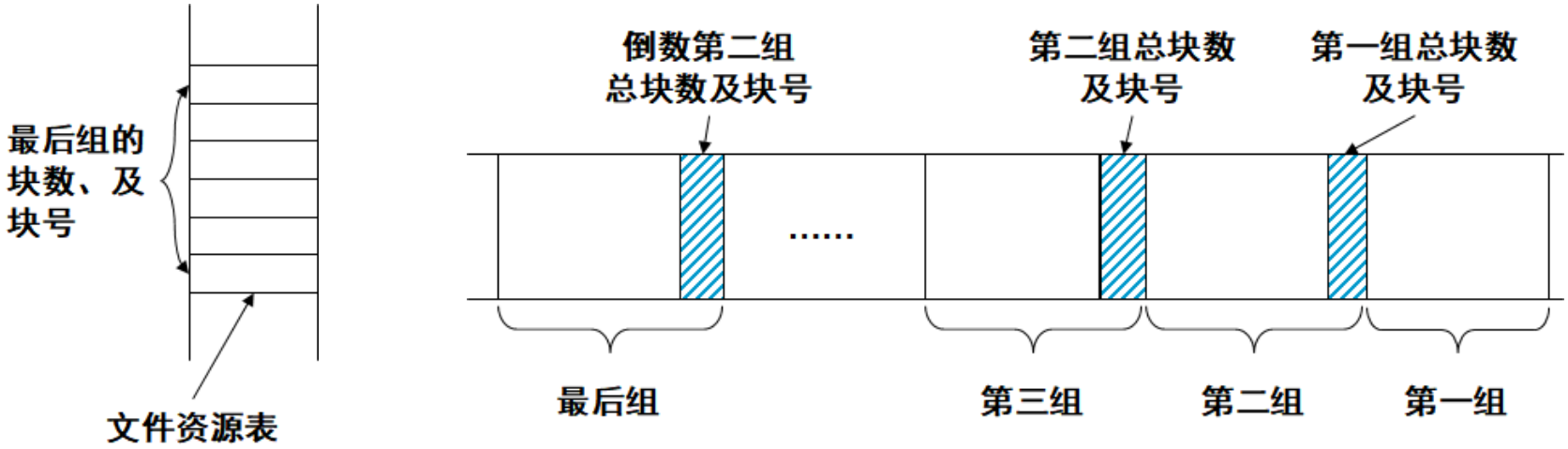
Grouping

- **Modify linked list to store address of next *n-1* free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)**



UNIX

* UNIX成组链接的方法



- 把文件存储设备中的所有空闲块从后向前，按50块为一组进行划分。
- 每组的第一块用来存放前一组中各块的块号和总块数。
- 由于第一组前面已无其他组存在，因此第一组为49块。
- 由于存储空间不一定正好是50的整数倍，所以最后一组有可能不足50块。
- 由于该组后边已经没有其他组了，所以最后一组的块号与总块数只能放在管理文件存储设备的文件资源表中。

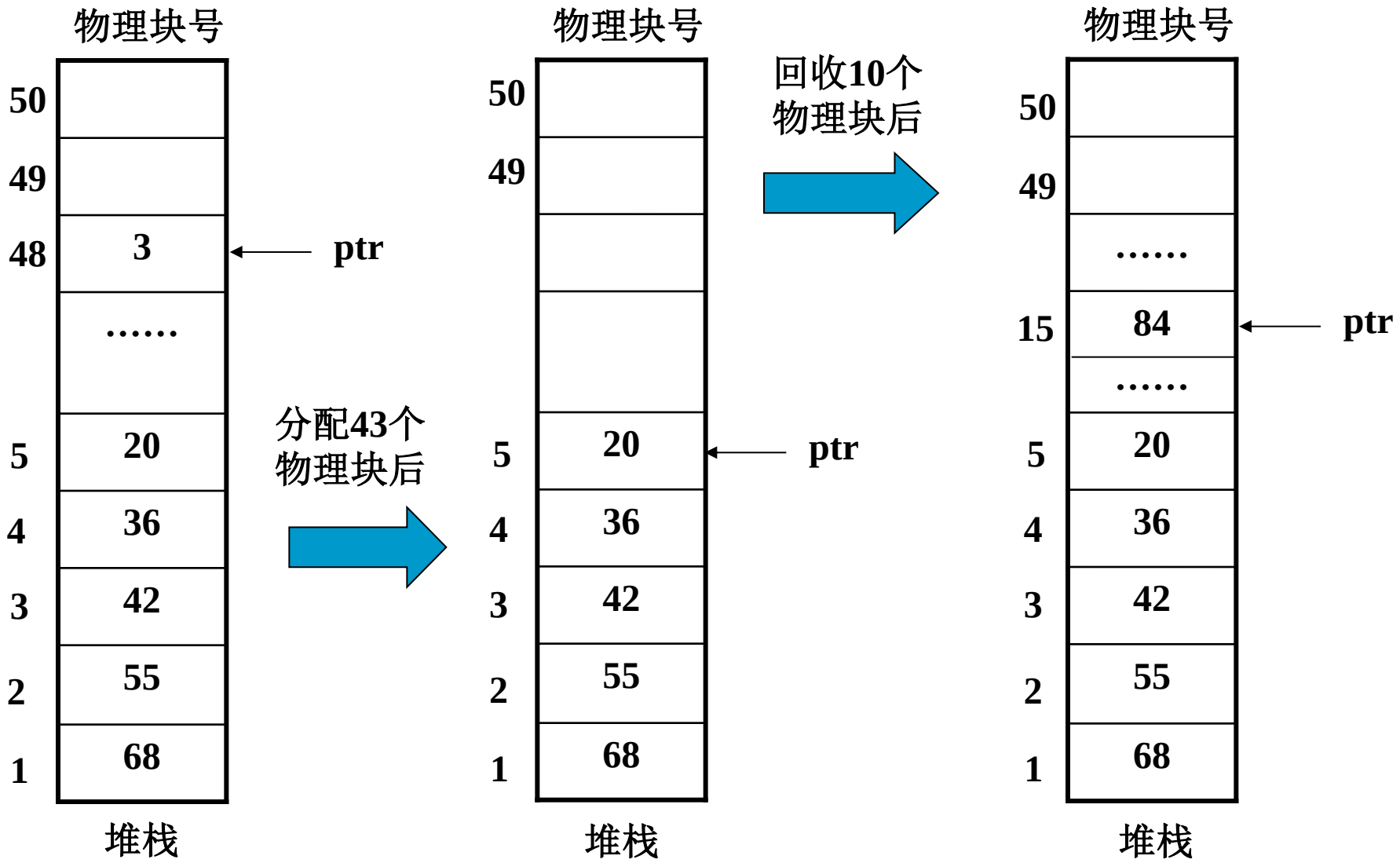
* UNIX成组链接的方法（续）

- 系统启动时把文件资源表复制到内存
  - 使文件资源表中的最后一组空闲块总数及块号的栈进入内存
- 设置一个用于空闲块分配与回收的栈
  - 栈指针ptr，ptr的初值等于该组空闲块的总块数。
- 成组链接法的分配方法
  - 申请者要求的空闲块数n
  - 按照后进先出的原则分配栈中的空闲块
    - 将ptr所指的块号分配给请求者，ptr-1 ==>ptr
  - 重复此操作，直到所要求的n块都已分配完毕、或栈中只剩下最后一个空闲块的块号。
    - 若栈中只剩下最后一个空闲块号
      - 弹出该块号，系统启动设备I/O，将该块中存放的下一组的块数和块号读入内存，压入栈中，将该块分配给请求者。
      - 重新设置ptr=下一组的块数。
      - 继续为申请者分配空闲块。

* 成组链接法的回收方法

- 成组链接法的分配方法（续）
  - 文件存储设备的最后一个空闲块中设置有尾部标识，指示空闲块分配完毕。
- 成组链接法的回收方法
  - 当用户删除某文件时，回收空闲块
  - 空闲块号入栈
    - ptr+1==>ptr
    - 把回收的物理块号放入ptr所指的位置
  - 如果ptr=50，表示该组已经回收结束
    - 如果还有空闲物理块F需要回收，回收该块并启动设备I/O，把栈中记录的50个块号与块数50写入块F中。
    - 设置ptr=1，将块F的块号入栈，开始新的一组空闲块的回收。
- 对空闲块的分配和释放过程对栈的操作必须互斥进行，否则会发生数据混乱。

* UNIX成组链接的方法示例



Counting

- Because space is frequently contiguously used and freed, with contiguous-allocation allocation.
  - Keep address of first free block and count of following free blocks.
  - Free space list then has entries containing addresses and counts.
- Similar to the extent or clustering method of allocating blocks.
- Entries can be stored in a balance tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Free-space list

addr1	count1
addr2	count2
addr3	count3
...	...

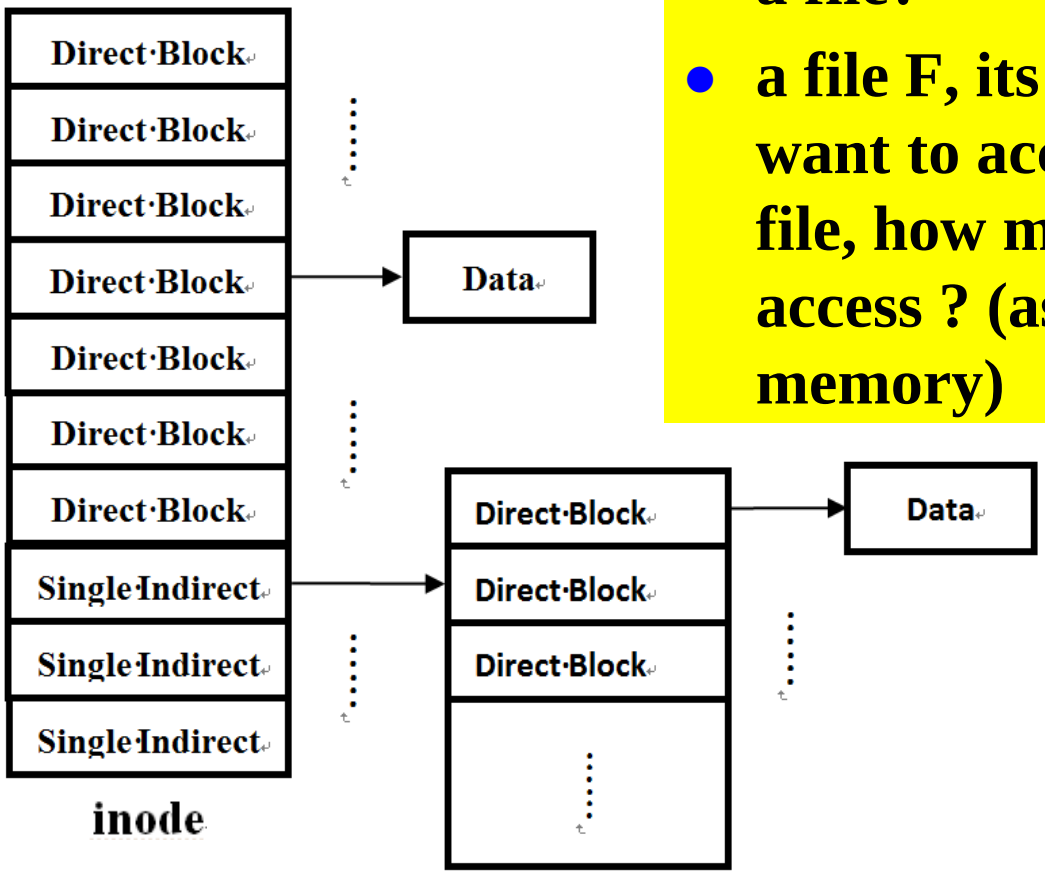
Free-Space Management (Cont.)

- **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*] = 0 in memory and bit[*i*] = 1 on disk.**
  - **Solution:**
    - **Set bit[*i*] = 0 on disk.**
    - **Allocate block[*i*]**
    - **Set bit[*i*] = 0 in memory**

exercise

- A file system, its allocation scheme is as follows:
- each file, only one block can be used as inode, in which, 10 block numbers can be hold at most, 7 point to data blocks and 3 Single indirect blocks.

- How many blocks is the largest size of a file?
- a file F, its size is 30 Blocks. If we want to access the third Block in this file, how many Blocks we need to access ? (assuming all Blocks is not in memory)



11.6* Efficiency and Performance

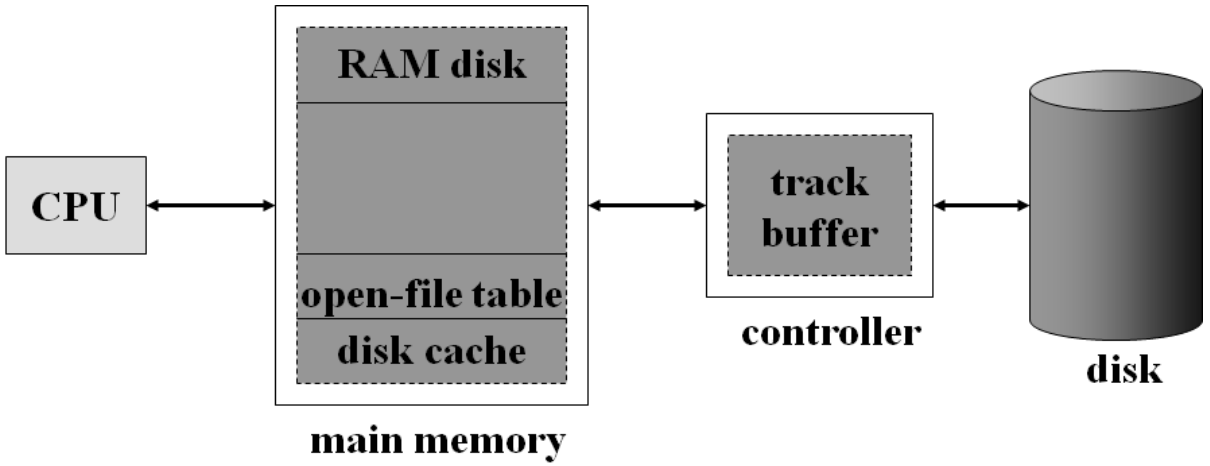
- **Efficiency dependent on:**
  - **disk allocation and directory algorithms**
    - **UNIX inodes are preallocated on a volume**
    - **UNIX allocation and Free-space algorithms try to keep a file's data blocks near that file's inode block to reduce seek time.**
  - **types of data kept in file's directory entry**
    - **Last write date, last access date**
  - **Pre-allocation or as-needed allocation of metadata structures**
  - **Fixed-size or varying-size data structures**
    - **size of pointers used to access data**
      - **32-bit pointers limits the size of a file to 2^{32}, or 4GB.**
      - **64-bit pointers allows file size 16 EB, but pointers require more space to store.**

Efficiency and Performance(Cont.)

- size of pointers used to access data(Cont.)
  - Choosing is difficult, the effect of changing technology.
    - MS-DOS, file system could support only 32MB.
      Each FAT entry was 12 bits, pointing to an 8-KB cluster.
    - Disk data structures and algorithms had to be modified to allow larger file system.
      Each FAT entry was expanded to 16 bits, and later to 32 bits.
    - Solaris's ZFS file system uses 128-bit pointers.
- Solaris OS
  - Originally, many data structures were of fixed length, allocated at system startup. E.g. Process table, open-file table, etc.
  - Later releases of Solaris, almost all kernel structures were allocated dynamically.

Efficiency and Performance(Cont.)

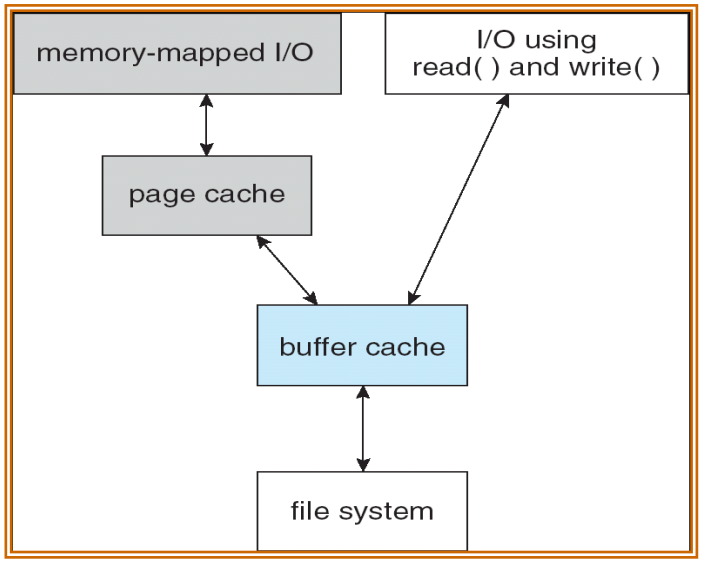
- **Performance**
  - most disk controllers include local memory to form an on-board cache that is large enough to store entire tracks at a time.

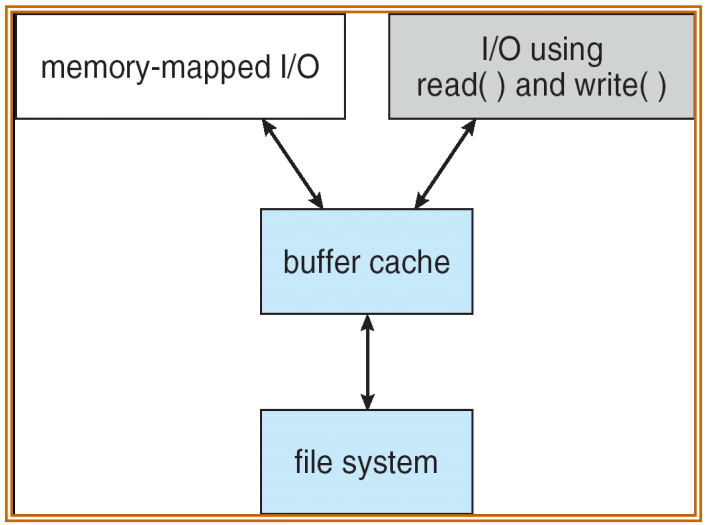


    - RAM disk: contents are controlled by user
    - Disk cache: controlled by OS
  - Keeping data and metadata close together.
  - Buffer cache – separate section of main memory for frequently used blocks.

Efficiency and Performance(Cont.)

- **Page Cache**
  - Caches pages rather than disk blocks using virtual memory techniques and addresses.
  - Memory-mapped I/O uses a page cache.
  - Routine I/O through the file system uses the buffer cache.
- **Unified Buffer Cache**
  - Uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching.





Free-behind and read ahead

- **Synchronous writes occur in the order in which the disk subsystem receives them, and the writes are not buffered.**
- **In an asynchronous write the data is stored in the cache and returns control to the caller.**
- **Allow a process to request that writes be performed synchronously.**
  - **Using a flag in the open system call.**
- **Free-behind removes a page from the buffer as soon as the next page is requested.**
- **Read ahead, a requested page and several subsequent pages are read and cached.**

11.7* 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.
- A typical backup schedule
  - Day 1: full backup
  - Day 2: incremental backup
  - ...
  - Day N: incremental backup. Go back to Day 1.

11.8* Log Structured File Systems

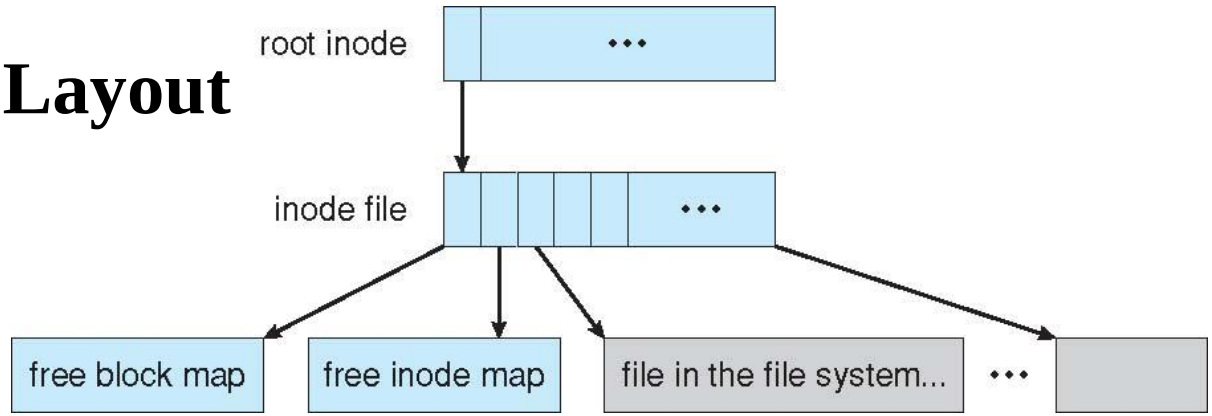
- **Log structured (or journaling) file systems record each update to the file system as a transaction.**
- **All transactions are written to a log.**
  - **A transaction is considered committed once it is written to the log.**
  - **Sometimes to a separate device or section of disk.**
  - **However, 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.**
- **Faster recovery from crash, removes chance of inconsistency of metadata.**

11.9* NFS

- 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.

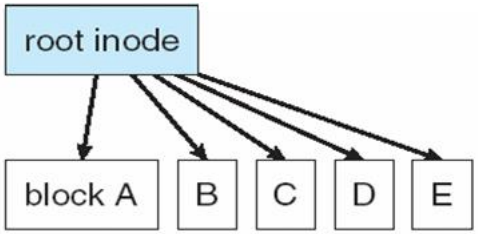
11.10* Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances.
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications.
- The WAFL File Layout

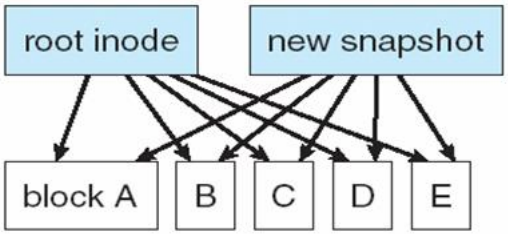


WAFL File System(Cont.)

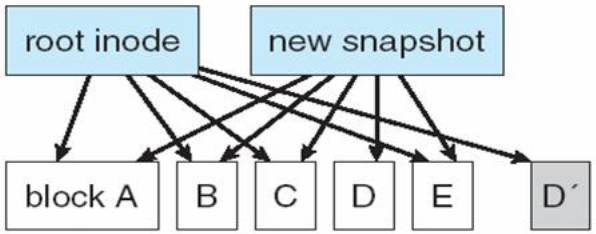
- **Snapshots in WAFL**



(a) Before a snapshot.



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



(c) After block D has changed to D´.

Homework

- 11.6
- Thinking about: 11.1 11.2 11.3

- **Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.**

  a. The block is added at the beginning.

  b. The block is added in the middle.

  c. The block is added at the end.

  d. The block is removed from the beginning.

  e. The block is removed from the middle.

  f. The block is removed from the end.