

Implementing File Systems

— Chapter 11

2022年11月

薛哲

xuezhe@bupt.edu.cn

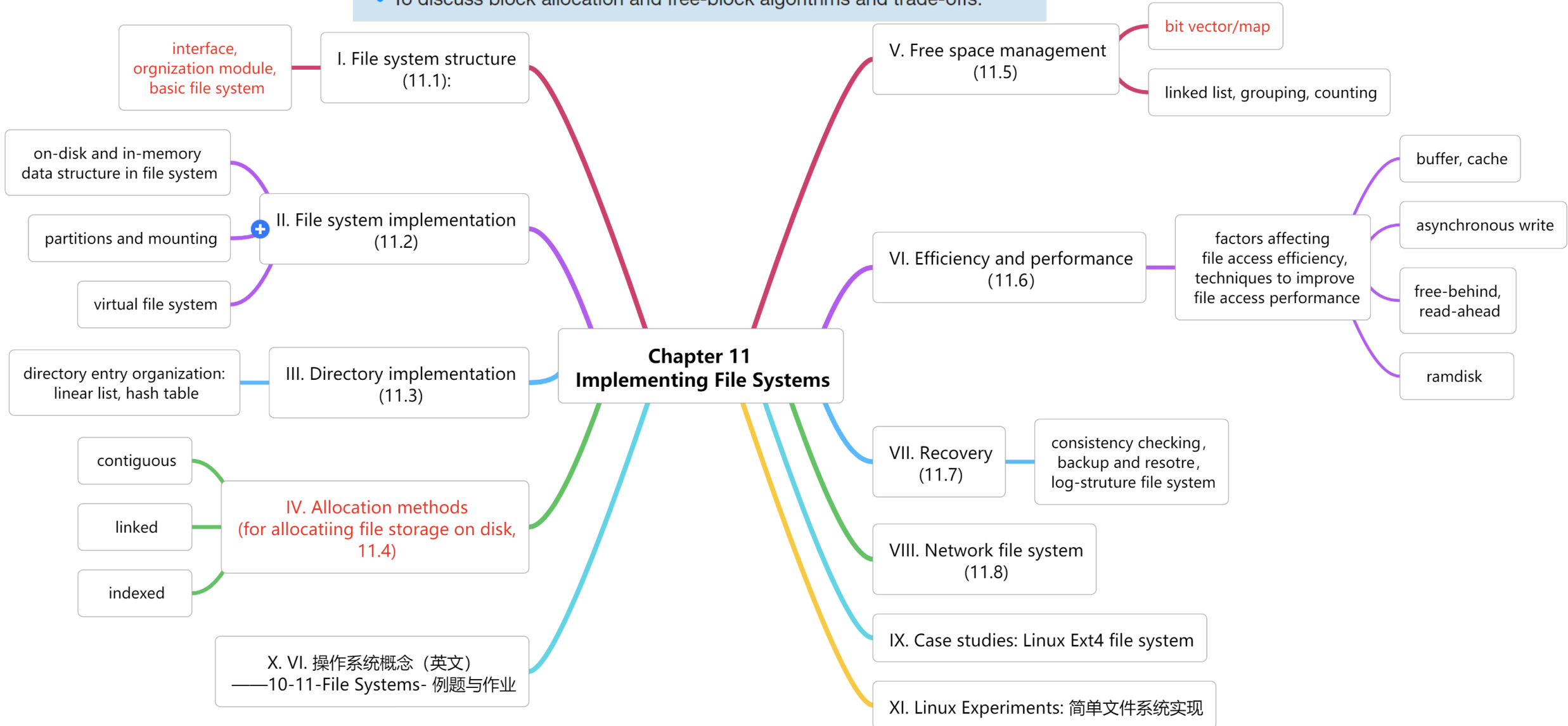
School of Computer Science (National Pilot Software Engineering School)




北京邮电大学

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.



11.1 File System Structure

- Disk's characteristics convenient for file storage 
- I/O transfer between memory and disk is performed in unit of **block**, which contains one or more **sectors**
 - being rewritten in places
 - read a block from disk, modify it , write it back to the same place
 - supporting direct/random access and sequential access, through moving read-write head and rotating disk-plate, feasible for accessing any block of information on disk

- Layered file system illustrated in Fig. 11.1

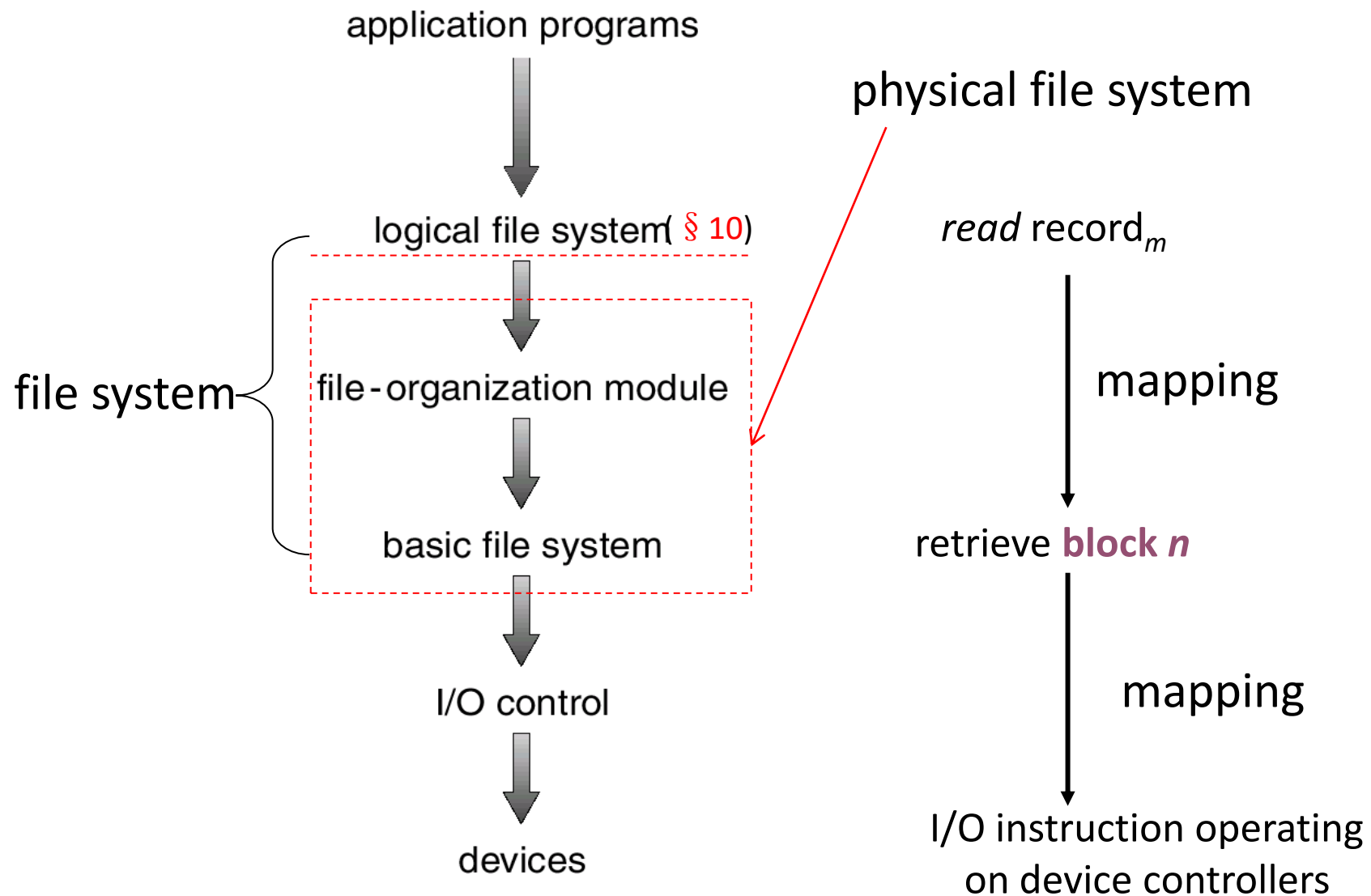


Fig.11.1 Layered File System

Layered File System

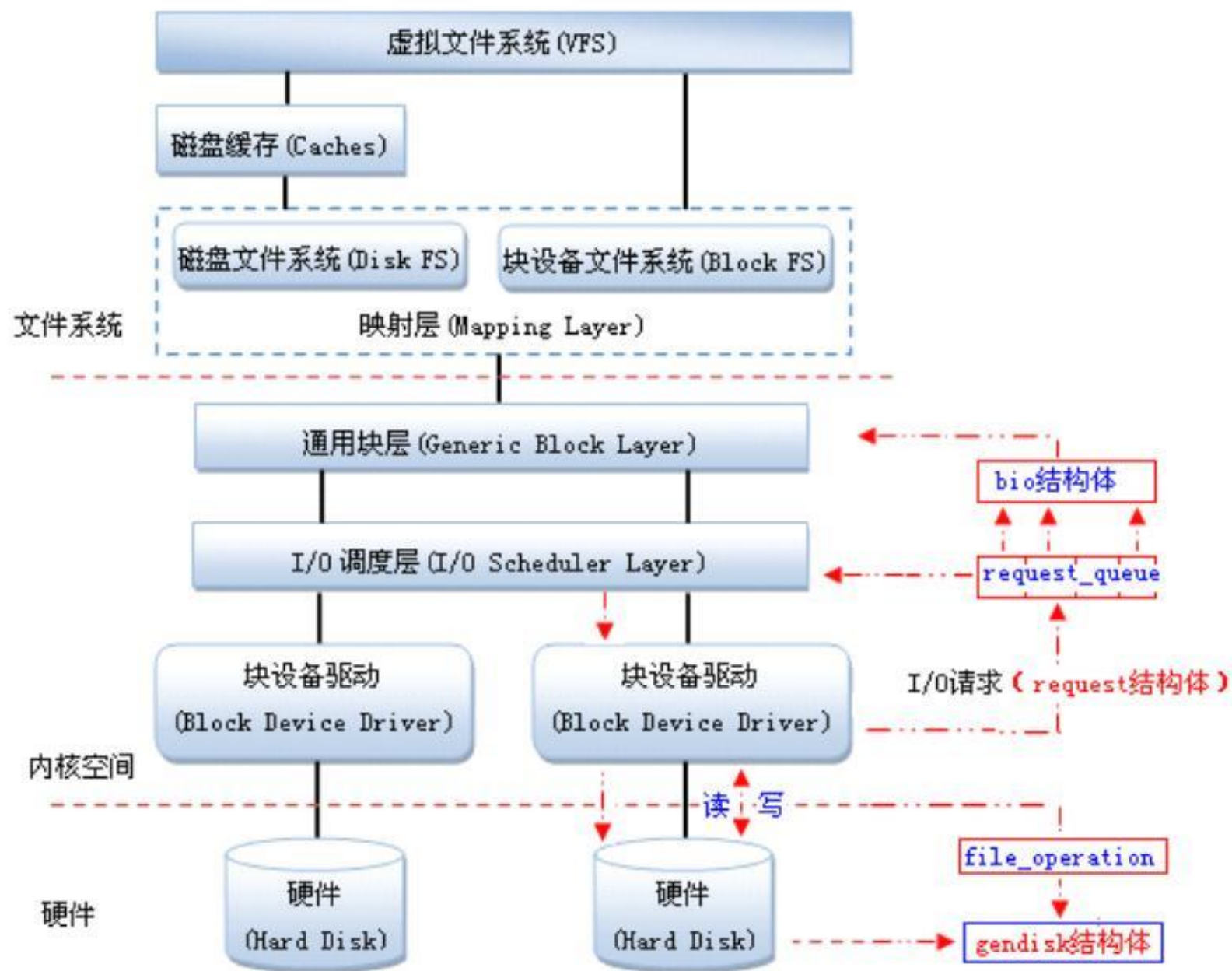
- Logical file system
 - ❑ file access interface for users, on the basis of *logical record number*
 - ❑ defining how the file system looks to users, including
 - file logical blocks/records and file attributes
 - allowed operations on files, e.g. *read, write, open, close, ...*
 - directories for file organization
 - ❑ managing metadata information about director structure, in terms of File Control Block (FCB)
 - ❑ providing file protection and security

Layered File System

- File organization module
 - ▣ mapping between ***logical block/record number*** and ***physical block address***
 - ▣ disk *free space* management and allocation
- Basic file system
 - ▣ generating **generic/high-level I/O commands** to device drivers to ***read*** or ***write*** physical blocks
 - ▣ on the basis of the ***block number*** or ***physical block address***
 - driver#, cylinder#, track#, sector#

Layered File System

- I/O control, i.e., I/O subsystem in OS kernel, including *device drivers* and *interrupt control*
 - ▣ conducting information transferring between memory and disk
 - ▣ device driver
 - input: **high-level I/O commands**, e.g. “retrieve **physical block *n***”, ***physical block address***
 - output: *CPU-specific I/O instructions*, operating directly on device controller, e.g. *LOAD port *k*, disk address*



Layered File System in Linux

■ 虚拟文件系统(VFS)

- ▣ 隐藏底层硬件、具体文件系统的具体细节，为用户访问多种类型的硬件、文件系统提供统一接口
 - open/close/write/read 等函数API
- ▣ 支持多种文件系统格式，比如EXT、FAT等

■ 映射层(mapping layer)

- ▣ 确定文件系统的block size，据此计算文件访问所请求的数据包含多少个block
- ▣ 调用具体文件系统接口函数访问文件的索引节点inode，确定所请求的数据在磁盘上的逻辑地址
 - 文件中的数据块可以存储在磁盘上的多个非连续块中，文件索引节点采用特定数据结构将文件中的每个数据块的块号映射到一个磁盘逻辑块的块号
 - 记录了文件数据块存储在哪个磁盘逻辑块上

Layered File System in Linux

■ 通用块层

- ▣ 针对所有不同类型的块设备，如disk、SSD，封装出块设备操作的各种接口
- ▣ 屏蔽了不同IO设备的硬件差异，对上兼容不同的VFS

■ IO调度层

- ▣ 将待处理的多个IO数据访问请求进行归类，如果两个数据访问请求→所访问的数据块在磁盘上的存放位置相邻，则将这些请求聚合在一起
- ▣ 对经过聚合后的访问请求，采用预先定义调度算法（如电梯算法），安排这些请求的执行顺序，以便快速执行全部访问请求
 - e.g. 1#, 200#, 60#, 201#, 2#, 63# → 1#, 2#; 60#, 63#; 200#, 201#

■ 块设备驱动层

- ▣ 设备驱动程序driver层
- ▣ CPU执行设备driver程序，直接访问IO设备，实现内存与外设间的数据传输

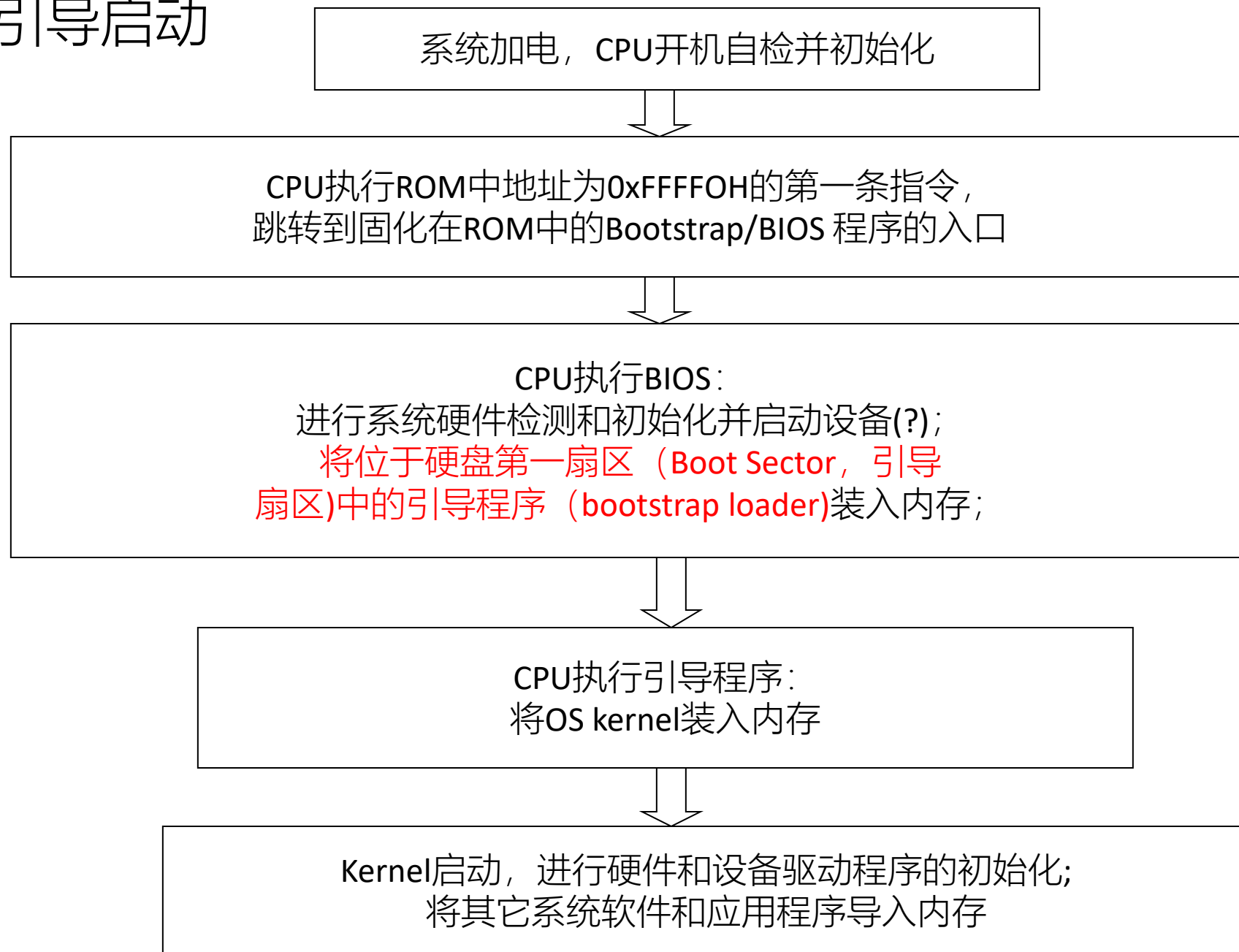
11.2 File System Implementation

- In this section, internal structures and operations for implementation of file operation

11.2.1 Overview: on-disk and in-memory structures

- On-disk structure
 - ▣ boot control block, volume/partition control block, directory structure, file control block(FCB)
- Boot control block (per volume)
 - ▣ contain information needed by the system to boot OS from the volume, also known as
 - *boot block* in **UFS**
 - *partition boot sector* in **NTFS**,
 - ▣ used for mounting OS from the corresponding partition,
e.g. one/two steps booting in chapter 3, **bootstrap loader in boot sector**

■ Two-step booting/引导启动



On-disk Structure

- Volume/partition control block
 - ▣ contain volume/partition details
 - the number of blocks in the partition,
 - size of the block,
 - free block count, free block pointer
 - free FCB count, free FCB pointer
 - ▣ also known as,
 - in UFS, called *superblock*
 - in NTFS, stored in the *master file table*

On-disk Structure

- Directory structure
 - used to organize file, also known as,
 - in UFS, it includes file names and associated inode number
 - in NTFS, stored in the master file table
- Per-file file control block(FCB)
 - contain many details about the file, including ownership, permission, location, etc.
 - also known as **inode**(index node) in Unix/Linux file systems
 - has an unique identifier to allow association with a directory entry
 - in NTFS, this details are stored in the master file table

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

Fig.11.2 A typical file control block

In-memory Structure

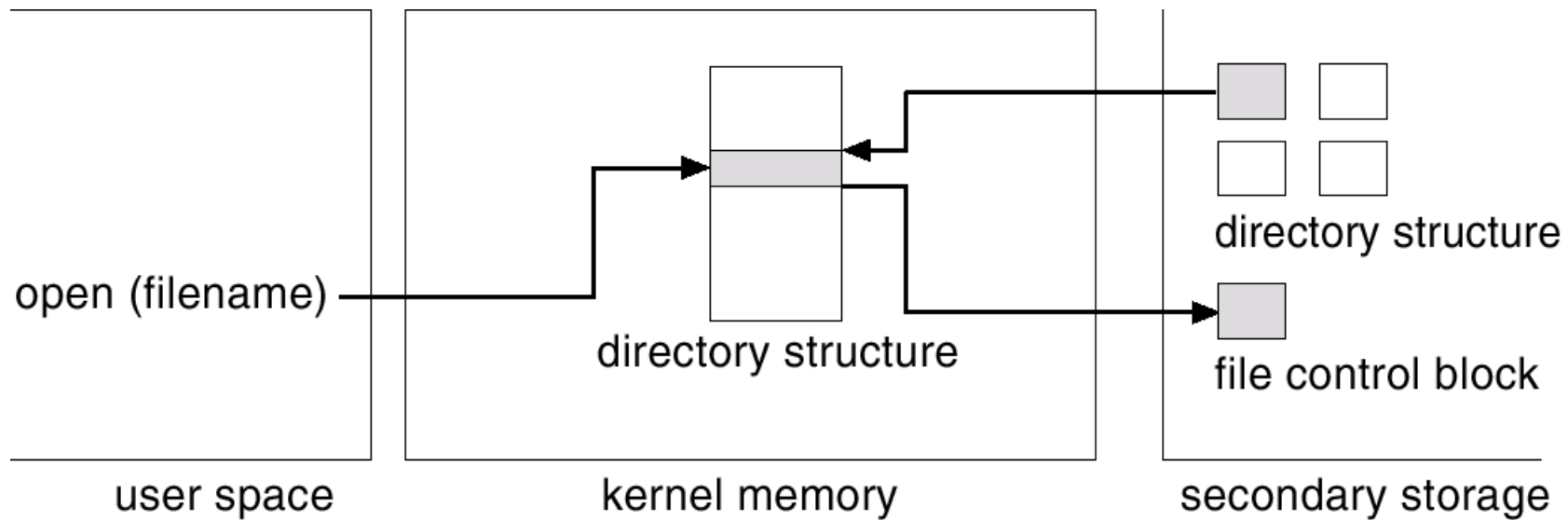
- The in-memory information is used for file-system management performance improvement via **caching**
 - cache some information in on-disk structures into in-memory structures
- Commonly-used in-memory structures are as follows
 - in-memory partition/mount table
 - information about mounted partition/volume
 - in-memory directory structure cache
 - about recently accessed directory
 - system-wide **open-file table**
 - FCBs of opened files
 - per-process **open-file table**
 - pointer to the entry in system-wide open-file table
 - opened files are viewed as the resources used by processes, recorded in PCB
 - buffer
 - hold file system blocks when they are read from disk or written back to disk

Open() vs on-disk and in-memory structures

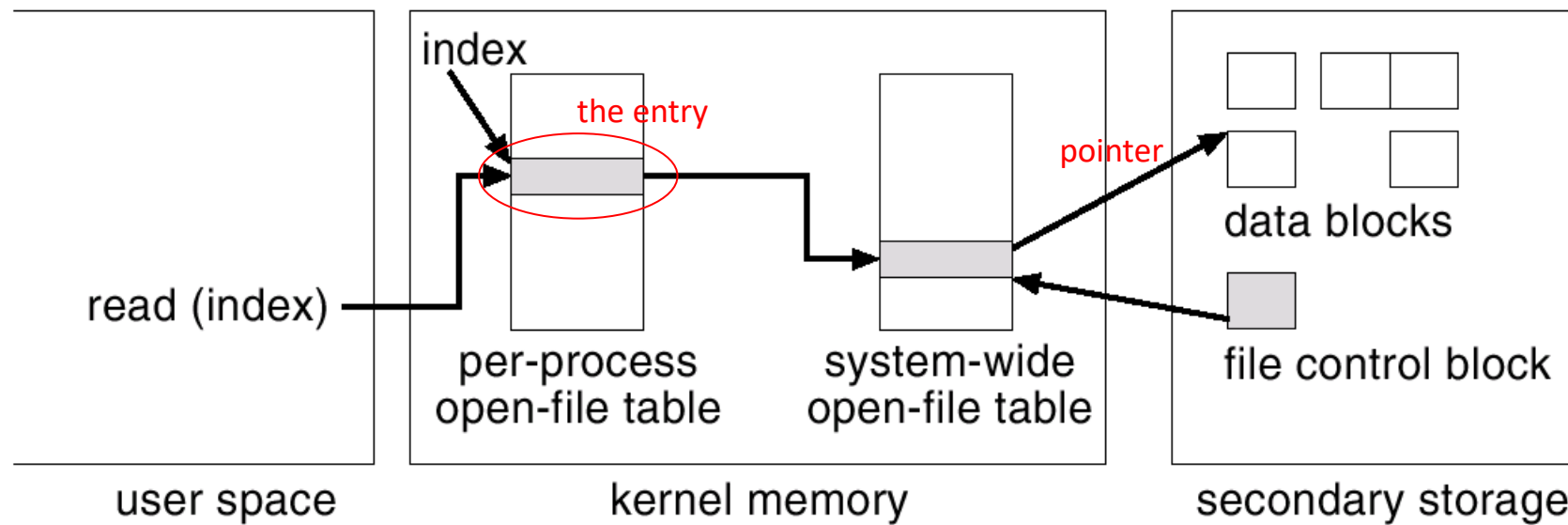
- On the basis of these data structures, the following file operation are provided
- Creating a new file
 - ▣ create the new file's FCB, and insert it into directory in the disk
- When a process opens a file by **Opening()**, OS will
 - ▣ search the system-wide open file table to see if the file is already in use by another process
 - ▣ if it is, OS makes an entry created in the **per-process open-file table** and this entry points to the system-wide open file, **with a pointer to the current location** in the file (for *read* or *write* operations)
 - refer to Fig. 11.3(a) for opening a file
 - ▣ if it is not, copy the FCB of the found file into the system-wide open file table in memory

Open() vs on-disk and in-memory structures

- ❑ open() returns a pointer to **the entry** in the per-process open-file table
 - all file operations are then performed via this pointer, and done on the open-file table
- ❑ the name given to the entry is called
 - **file descriptor** (文件描述符FD) in Unix/Linux
 - **file handle** in Windows
- ❑ refer to Fig. 11.3(b) for reading a file



(a)



(b)

Fig. 11.3 In-Memory File System Structures

Example

23. 若多个进程共享同一个文件 F，则下列叙述中正确的是：

- A、个进程只能用“读”方式打开文件 F；
- B、在系统打开文件表中仅有一个表项包含 F 的属性；
- C、各进程的用户打开文件表中关于 F 的表项内容相同；
- D、进程关闭 F 时系统删除 F 在系统打开文件表中的表项。

【答案】 B

【解析】

- A：各进程既可以用读方式打开文件 F，也可用写方式打开文件 F
- B：系统打开文件表只有一张，正确
- C：打开文件表关于同样一个文件的表项内容不一定相同
- D：进程关闭 F 时会使 F 的引用计数-1，引用计数=0 时才会删除表项

408-20

Partitions and Mounting

- A disk can be sliced into several partitions, and a partition can also span several disk (e.g. RAID)
- Partitions can be divided into
 - ▣ **raw** partition, or **raw disk**, containing no file system
 - e.g. Unix swap space, database-used raw disk
 - *users are not allowed to access raw partition directly by means of system calls*
 - ▣ **cooked** partition, containing a file system

Partitions and Mounting

- Boot partition
 - ▣ a separate partition is used to store **boot** information
 - ▣ containing the OS kernel, and being mounted at booting time
 - ▣ boot information has its own format, and is usually loaded as a sequential series of blocks into memory.
- When a partition or file system is mounted, OS verifies whether or not the partition is valid (e.g. format is right ?), and records information about the valid file system in the partition in a in-memory **mount table**

Example: 408-21

46. (8 分) 某计算机用硬盘作为启动盘, 硬盘第一个扇区存放主引导记录, 其中包含磁盘引导程序和分区表。磁盘引导程序用于选择要引导哪个分区的操作系统, 分区表记录硬盘上各分区的位置等描述信息。硬盘被划分成若干个分区, 每个分区的第一个扇区存放分区引导程序, 用于引导该分区中的操作系统。系统采用多阶段引导方式, 除了执行磁盘引导程序和分区引导程序外, 还需要执行 ROM 中的引导程序。请回答下列问题。

- (1) 系统启动过程中操作系统的初始化程序、分区引导程序、ROM 中的引导程序、磁盘引导程序的执行顺序是什么?
- (2) 把硬盘制作为启动盘时, 需要完成操作系统的安装、磁盘的物理格式化、逻辑格式化、对磁盘进行分区, 执行这 4 个操作的正确顺序是什么?
- (3) 磁盘扇区的划分和文件系统根目录的建立分别是在第(2)问的哪个操作中完成的?

46. 【答案要点】

- (1) 执行顺序依次是 ROM 中的引导程序、磁盘引导程序、分区引导程序、操作系统的初始化程序。
- (2) 4 个操作的执行顺序依次是磁盘的物理格式化、对磁盘进行分区、逻辑格式化、操作系统的安装。
- (3) 磁盘扇区的划分是在磁盘的物理格式化操作中完成的。文件系统根目录的建立是在逻辑格式化操作中完成的。

Virtual File Systems (VFS)

- Why VFS needed
 - ▣ an OS may contain multiple file systems
 - FAT32, UFS, NTFS, ext2, ...
 - ▣ to support currently multiple types of files, in a generic API
- VFS Functions
 - ▣ separate file system generic operations from their implementation by **VFS interface**, allowing **transparent** access to different type of file systems
 - ▣ refer to Fig.11.4 Schematic view of virtual file system
- VFS allows the same system call interface (i.e. POSIX API/system call) to be used for different types of file systems
 - ▣ the API is generic and not specific to any particular types of file system

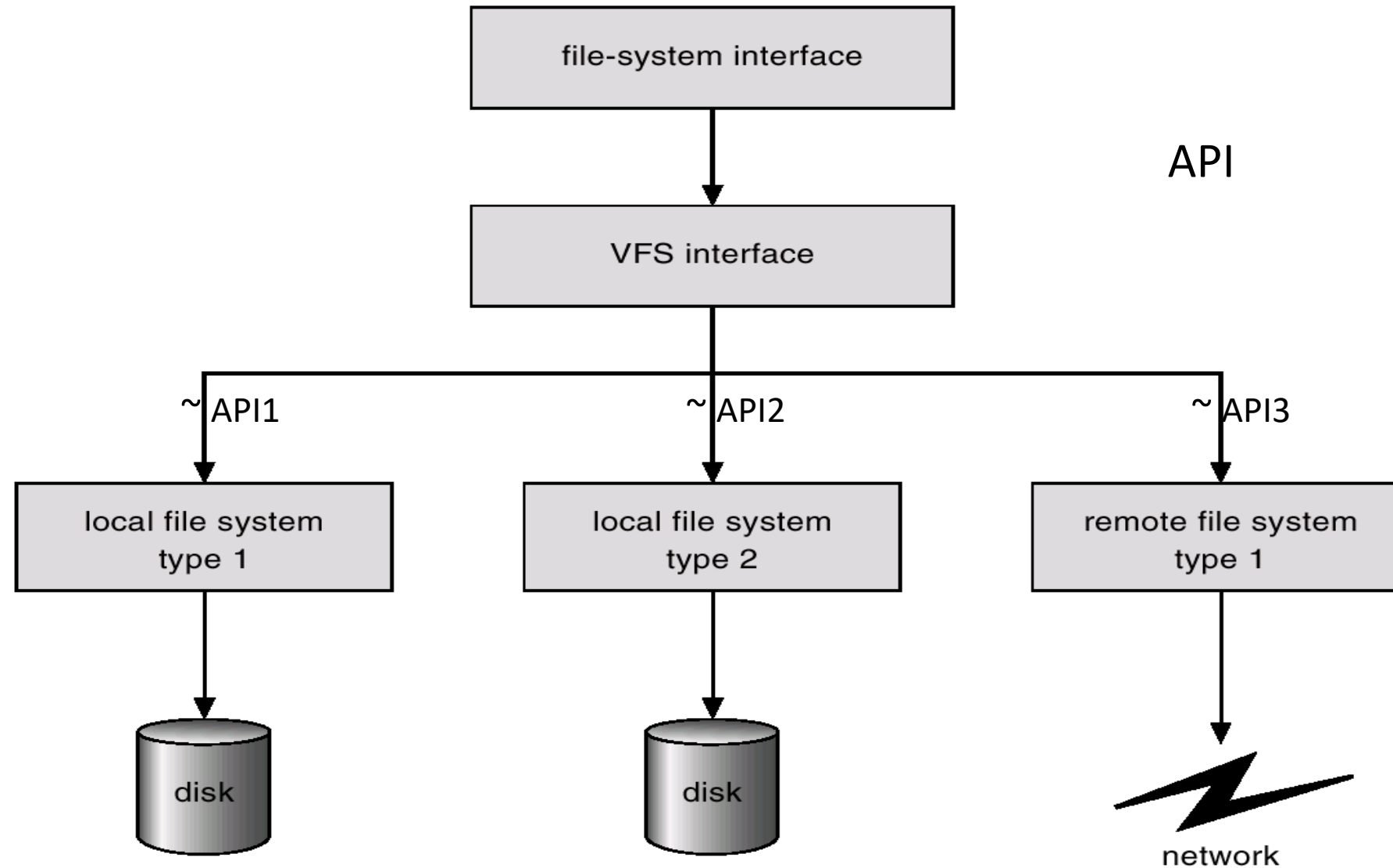
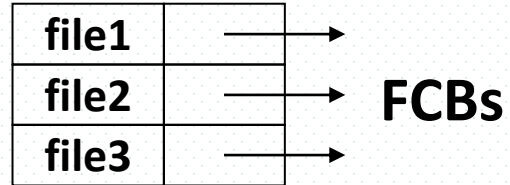


Fig. 11.4 Schematic view of virtual file system

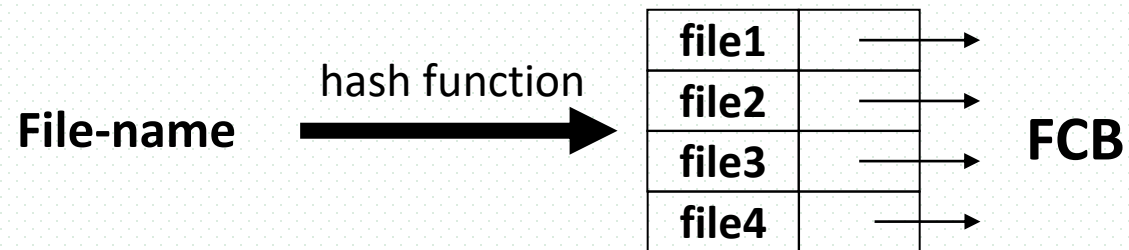
11.3 Directory Implementation

- Directory
 - ▣ set of entries
 - ▣ entry: file-name → FCB or *pointer to FCB*, e.g. Fig.11.3 (a)
- Directory implementation
 - ▣ how to organize the entries in the directory for rapid directory access
- **Linear list based implementation**
 - ▣ linear list of file names with pointer to the data blocks
 - ▣ simple to program, but time-consuming to execute: linear search to find a file
- **Hash-table-based implementation**
 - ▣ hash table – linear list with hash data structure
 - ▣ decreases directory search time
 - ▣ *may be collisions* – situations where two file names hash to the same location

Directory Implementation



(a) list-based implementation



(b) Hash-table-based implementation

■ inode

▣ index node,索引|节点

408-20

31. 某文件系统的目录由文件名和索引节点号构成。若每个目录项长度为 64 字节，其中 4 个字节存放索引节点号，60 个字节存放文件名。文件名由小写英文字母构成，则该文件系统能创建的文件数量的上限为：

A、 2^{26} B、 2^{32} C、 2^{60} D、 2^{64}

【答案】 B

【解析】

创建的文件数量上限=索引节点数量上限，索引节点为 4 个字节 32 位，故最多 2^{32} 个索引节点，即最多创建 2^{32} 个文件

11.4 Allocation Methods

文件物理结构

- Allocation methods refer to how disk blocks are allocated for files, that is, the data organization of the file on the disk
 - ▣ e.g. block size: 2^{10} bytes = 1KB, file size 1M = 2^{20} B = 1024 blocks
- Three major methods
 - ▣ contiguous allocation
 - ▣ linked allocation
 - ▣ indexed allocation

408-20

24. 下列选项中支持文件长度可变, 随机访问的磁盘存储空间分配方式是:
A、索引分配; B、链接分配; C、连续分配; D、动态分区分配。

【答案】 A

【解析】

B: 链接分配不支持随机访问

C: 连续分配不支持长度可变

D: 动态分区分配是内存管理方式

Contiguous Allocation (顺序文件)

■ Principle

- each allocated file occupies a set of contiguous blocks on the disk
- file address is defined by file's starting location (block #) and length (number of blocks)
- random/direct access is used
- refer to Fig. 11.5

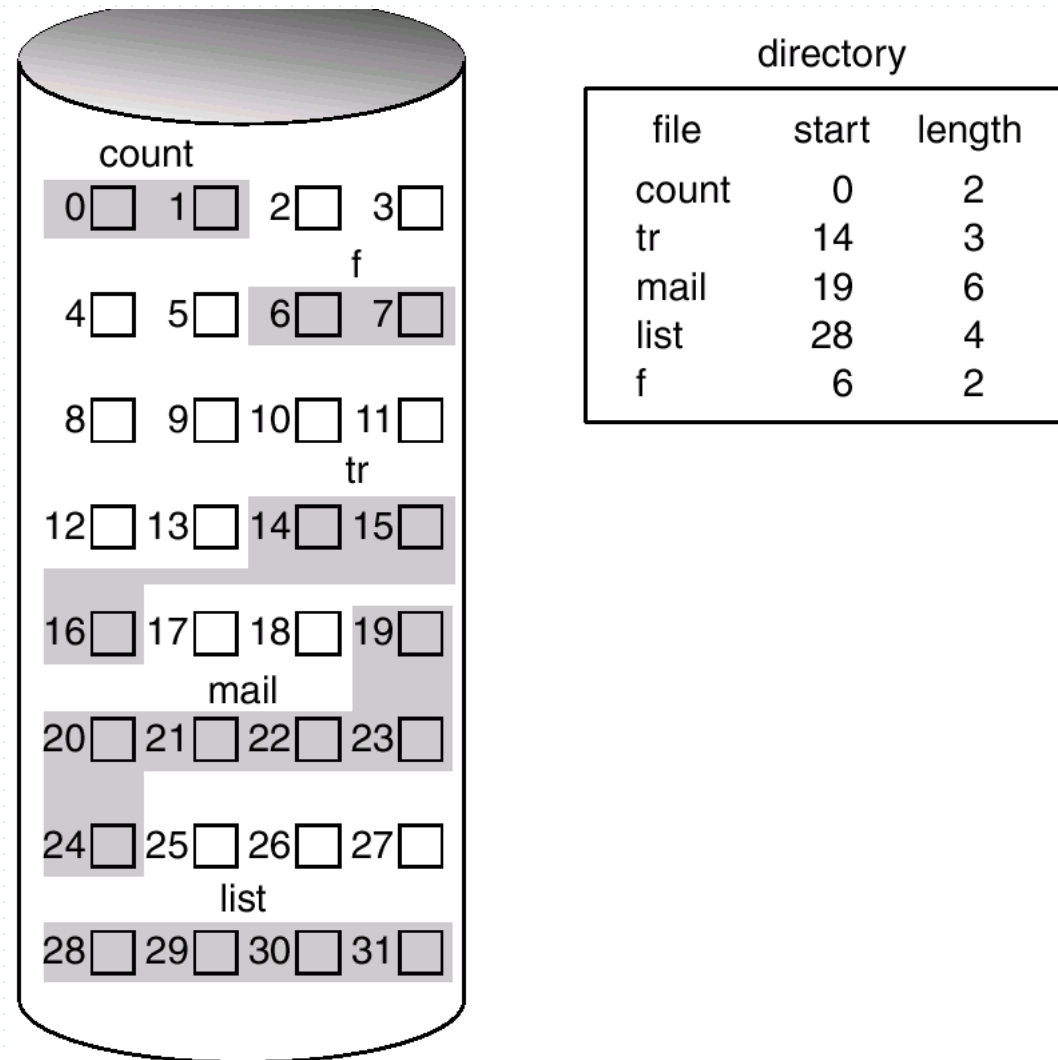


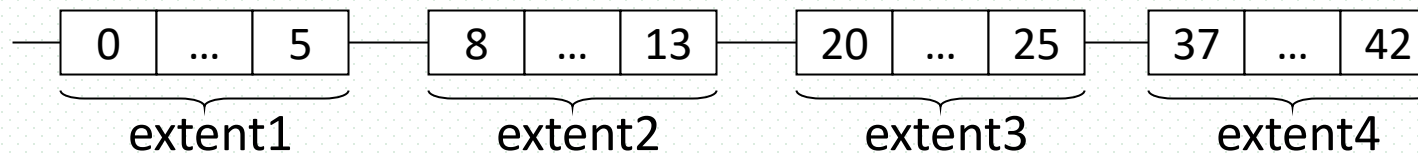
Fig. 11.5 Contiguous Allocation of Disk Space

Contiguous Allocation

- Contiguous allocation is a particular application of dynamic storage-allocation problem mentioned in §8.3
 - ▣ first fit, best fit
- Some older microcomputer systems used contiguous allocation of space on *floppy* disk
- Characteristics
 - ▣ wasteful of disk space, external fragmentation
 - ▣ when a file is created, the total amount of disk space the file needed must be allocated, so that file size may not grow dynamically

Contiguous Allocation

- An improvement of contiguous allocation, **extent-based** contiguous allocation
 - ▣ an **extent** is a contiguous block of disks, with fixed sizes
 - ▣ the file is allocated disk space in units of extents, not in the unit of blocks
 - ▣ a file consists of one or more extents, and these extents may not be contiguous
 - ▣ extent-based contiguous allocation is used in many newer file systems, such as **Veritas File System**
 - ▣ *e.g. a file with the size of 24 blocks, and 4 extents*



Linked Allocation (链接文件)

■ Principles

- each file is organized as a linked list of disk blocks
 - blocks may be scattered anywhere, i.e. noncontiguous on the disk
- file address: the initial address of the file
- *refer to Fig.11.6*

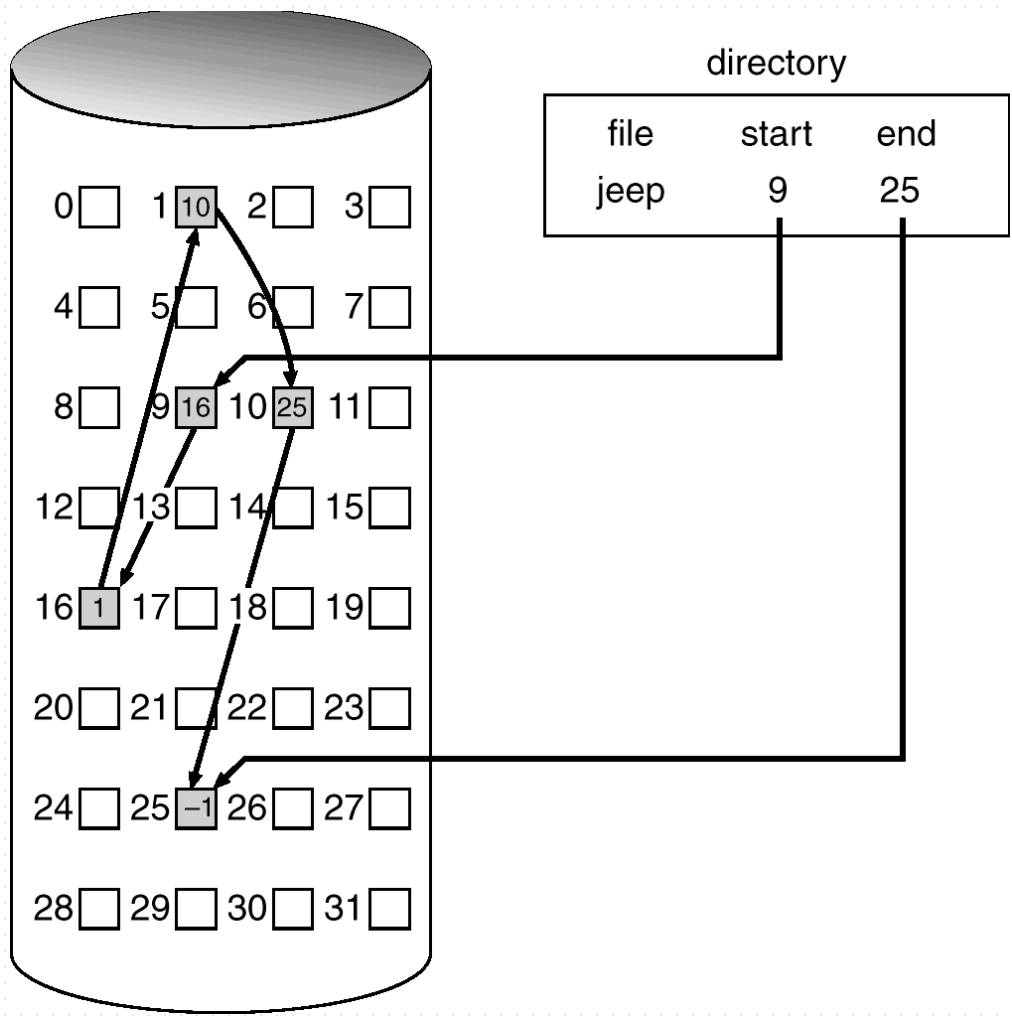
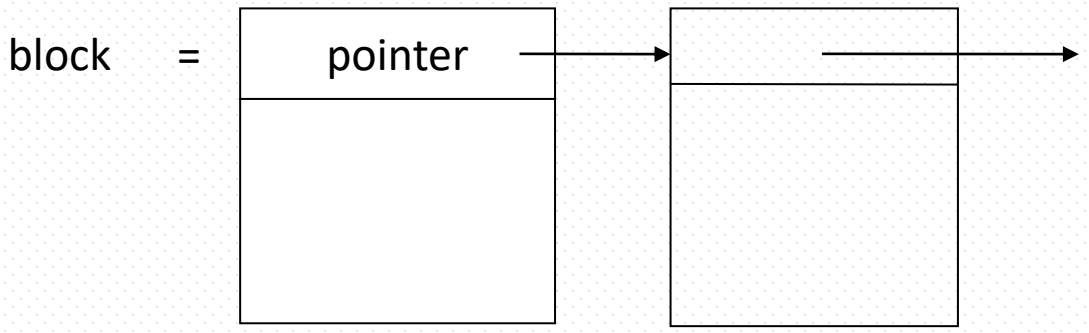


Fig.11.6 Linked Allocation

Linked Allocation

- Characteristics
 - ▣ merits
 - no waste of space, but may have internal fragmentation
 - ▣ demerits
 - sequential access, ***random access not allowed!!***
 - space for the pointer may be large
- An implementation— FAT-based linked allocation
 - ▣ FAT, File-allocation table
 - data structure for disk-space allocation
 - used by MS-DOS and OS/2

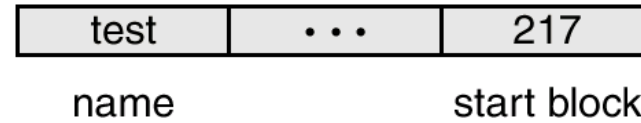
Linked Allocation

- FAT-based allocation

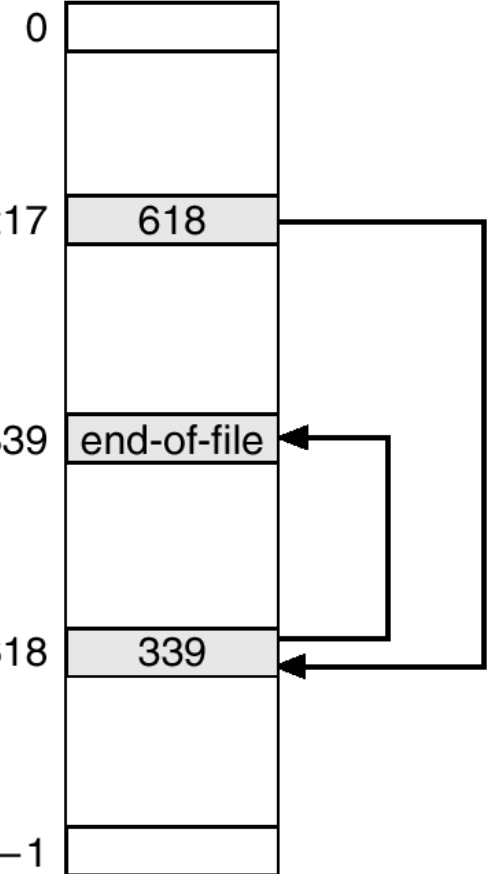
- refer to Fig.11.7
- each partition has one FAT, FAT is located at the beginning section of each partition
- **FAT has one entry for each disk block in the partition**, and is indexed by block number
- FAT is used as a linked list for the files allocated for this partitions
- the directory entry for a file contain the first block's block# of this file, the FAT entry indexed by that block number contains the next block's block# in the file
- **unused blocks are indicated by a 0 table value**

the file **test** and its block **217; 618; 339**

directory entry



partition k



(similar to *inverted page tables*)

例题：
根据指针长度，确定文件最大
size，
e.g. 指针长度为4bytes=32 bits，
可以指向最多 2^{32} 个blocks，文件
最大size为 2^{32} blocks

no. of disk blocks

FAT

Fig. 11.7 File-Allocation Table

Indexed Allocation

■ Principles

- use pointers to indicate the addresses of file blocks, and brings all pointers together into the **index block**, or **index table**
- each file has one or more index blocks, which is an array of disk-block address, the i th entry in the index block points to the i th block of the file
- e.g. Fig. 11.8

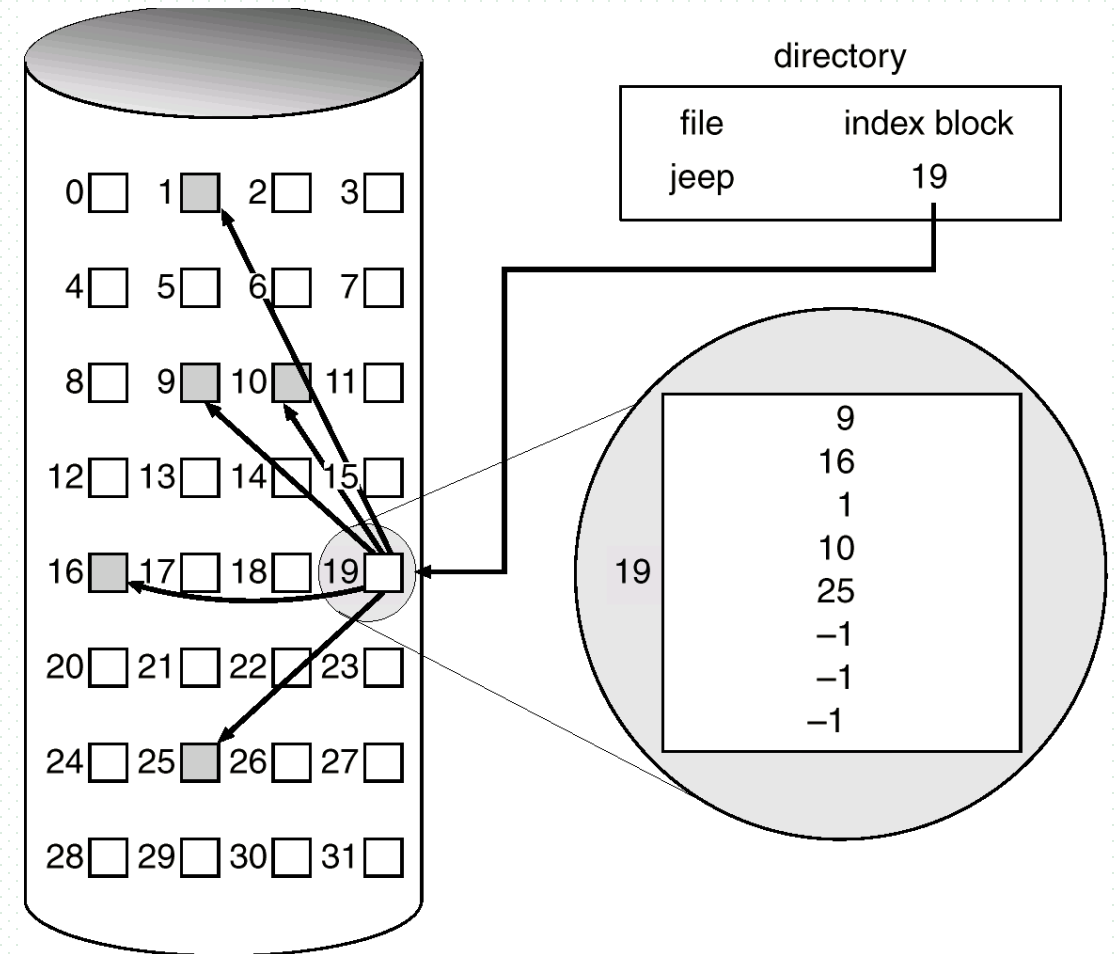
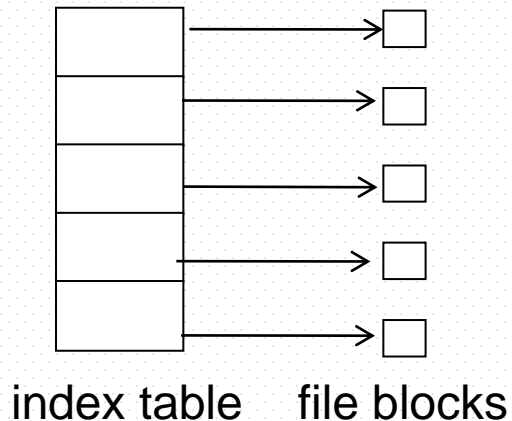


Fig. 11.8 Example of indexed allocation

Indexed Allocation

■ Merits/demerits

- ❑ random access
- ❑ dynamic access without external fragmentation
- ❑ overhead of index block, e.g.
 - maximum size of file: 256K words
 - block size: 512 words
 - index table: $256K \div 512\text{words} = 512\text{ words} = 1\text{ block}$

■ Improvement

- ❑ when the file is large, multi-level indexes can be used
- ❑ combined scheme can be used
- ❑ refer to Fig. 11.9.0 and Fig. 11.9

Indexed Allocation

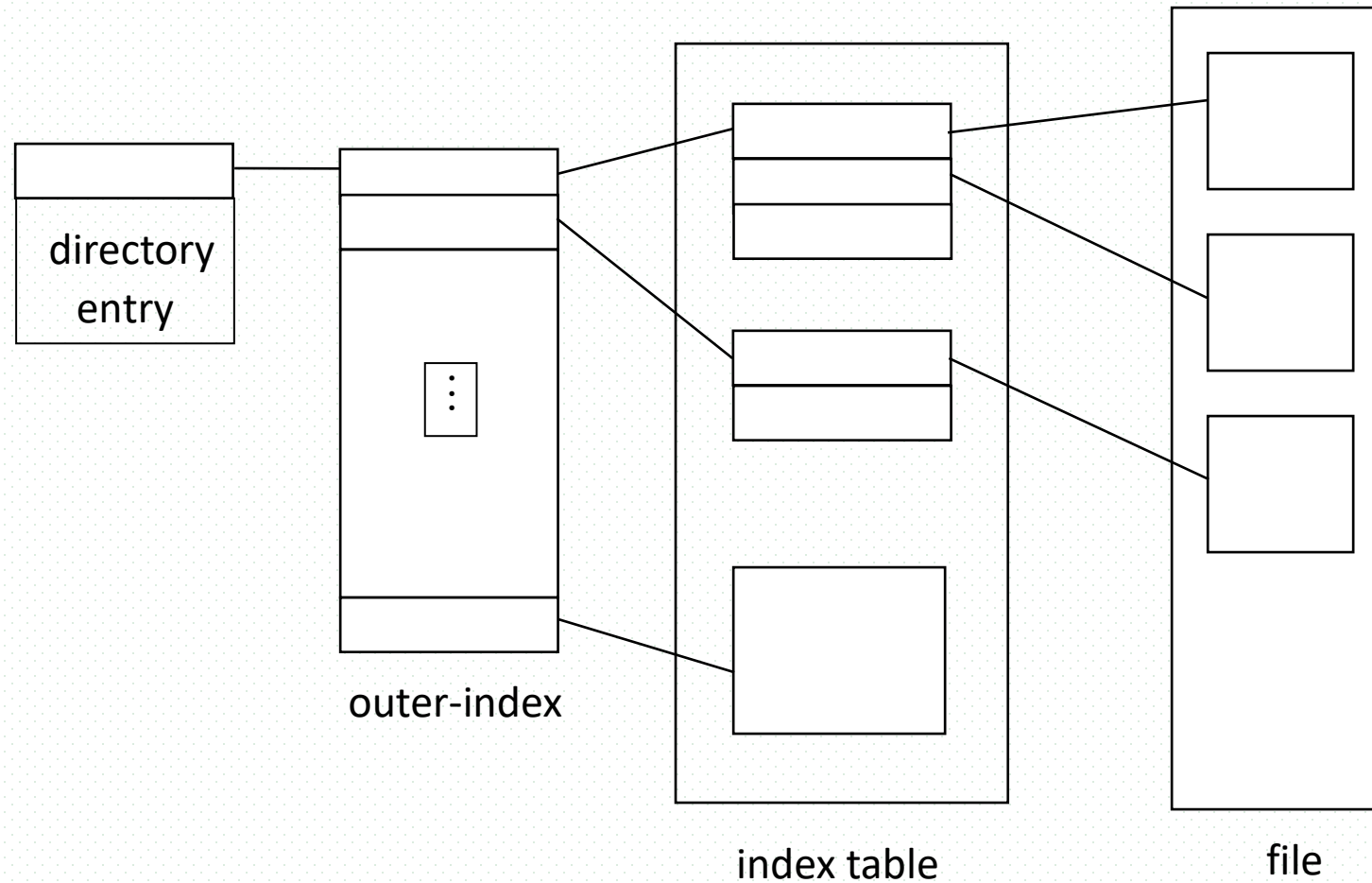


Fig. 11.9.0 Multi-level index

Indexed Allocation

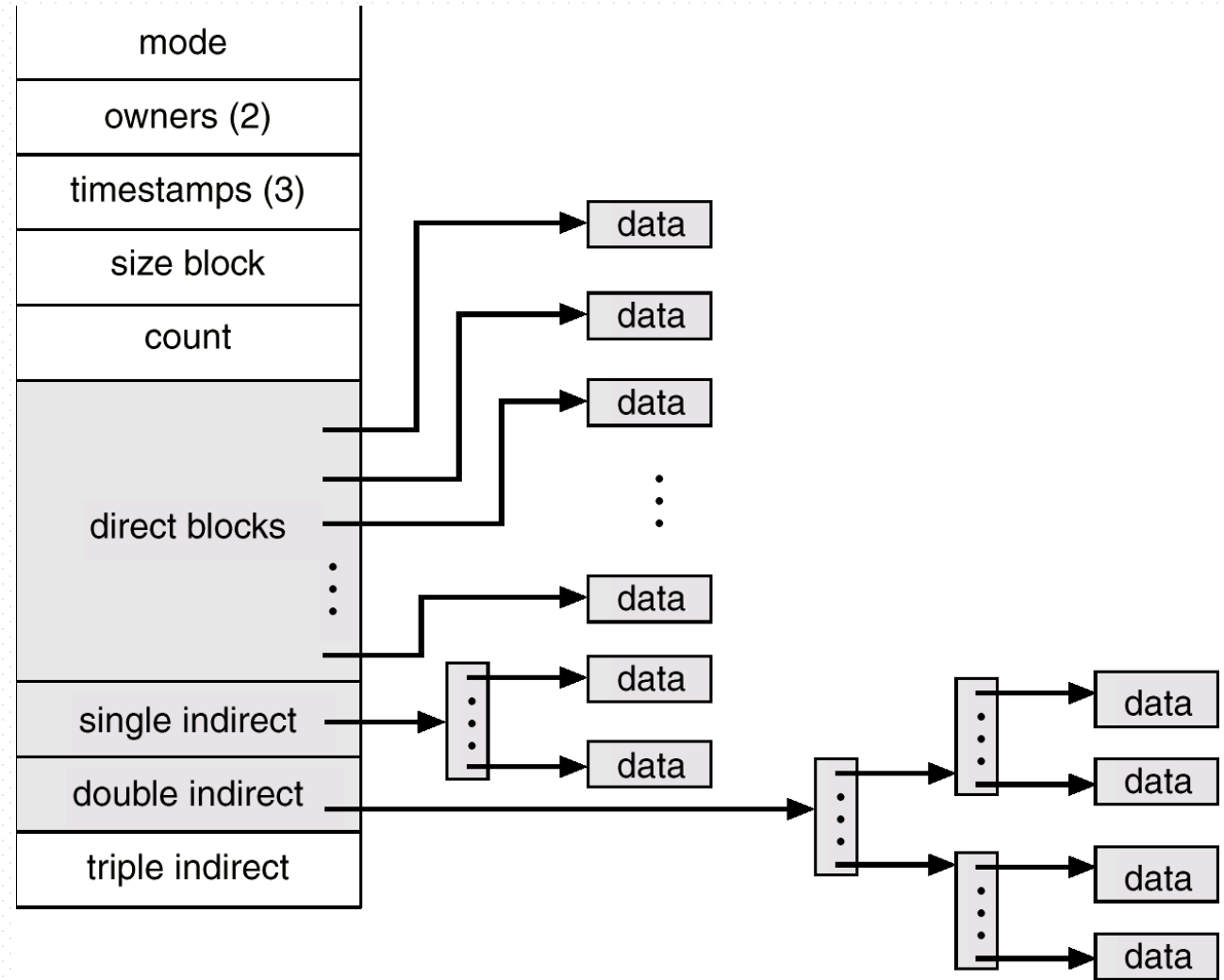


Fig. 11.9 Combined scheme in UNIX (4K bytes per block)

11.4.4 Performance

- Evaluating allocation methods mentioned above, considering
 - ▣ data-block access time
 - ▣ storage efficiency, i.e. fragmentations, especially external fragmentations

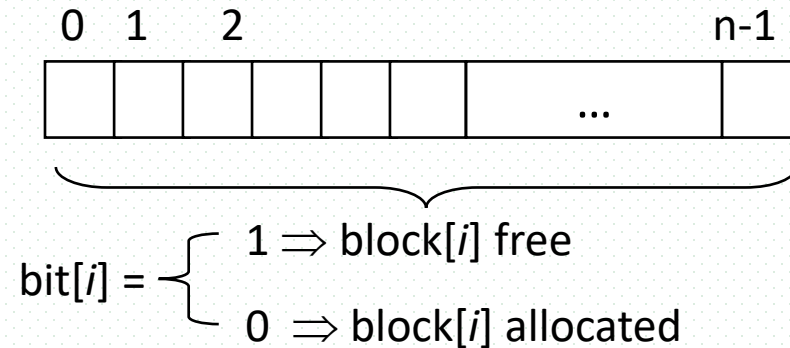
11.5 Free-Space Management

- Free-space management
 - ▣ recording free blocks in disk space by ***free-space list***
e.g. mount table
 - ▣ modifying free-space list when files are created or deleted
- To implement *free-space list* based free-space management, four major mechanisms can be used
 - ▣ **bit vector/map**
 - ▣ linked list
 - ▣ grouping
 - ▣ counting

Bit Vector/Map

■ Principles

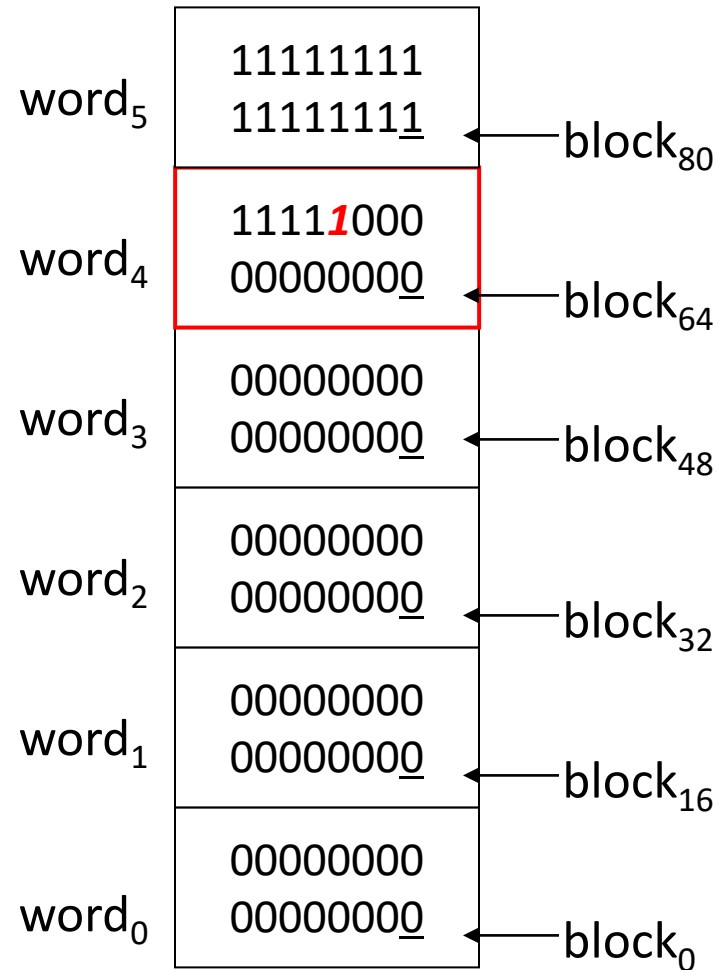
- supposed that there are n blocks on disk, the vector/map consists of a set of ***contiguous bytes/words***, with n -bits in length
- each block on the disk is represented by one bit in the vector



- to find the first free block, OS check sequentially in bit map/vector to find the first non-0 word, the block number of founded free block is:

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

■ Fig.11.5.1 Example of bit vector/map



disk size: total number of blocks
= $6 * 16 = 96$

number of bits per word = 16

size of bit map: 6 words

number of 0-value words = 4

first non-0-value word: word₄

offset of first 1 bit in word₄: 11

first free block is **block₇₅** :
 $16 * 4 + 11$

bit vector/map: block₀ ~ block₉₅

Bit Vector/Map

- Used in Macintosh OS, also in DBMS
- Easy to get contiguous files
- Bit map requires extra space

□ e.g.

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

disk size = 2^{40} bytes (1024*1 gigabyte)=1T

$n = 2^{40}/2^{12} = 2^{28}$ bits= 2^{25} bytes

= $2^5 * 2^{20}$ bytes (or 32 M bytes)

so, the size of bit map is 32M, occupies $32\text{M}/4\text{KB}=8\text{K}$ blocks

Linked list (free list)

■ Principles

- ❑ linking together all the free disk blocks
- ❑ keeping a pointer to the first block in a special location on the disk and caching it in memory
- ❑ the first block containing a pointer to the free next block, and so on
- ❑ refer to Fig. 11.10

■ Demerits

- ❑ getting contiguous space is not easy
- ❑ traversing the list may be time-consuming

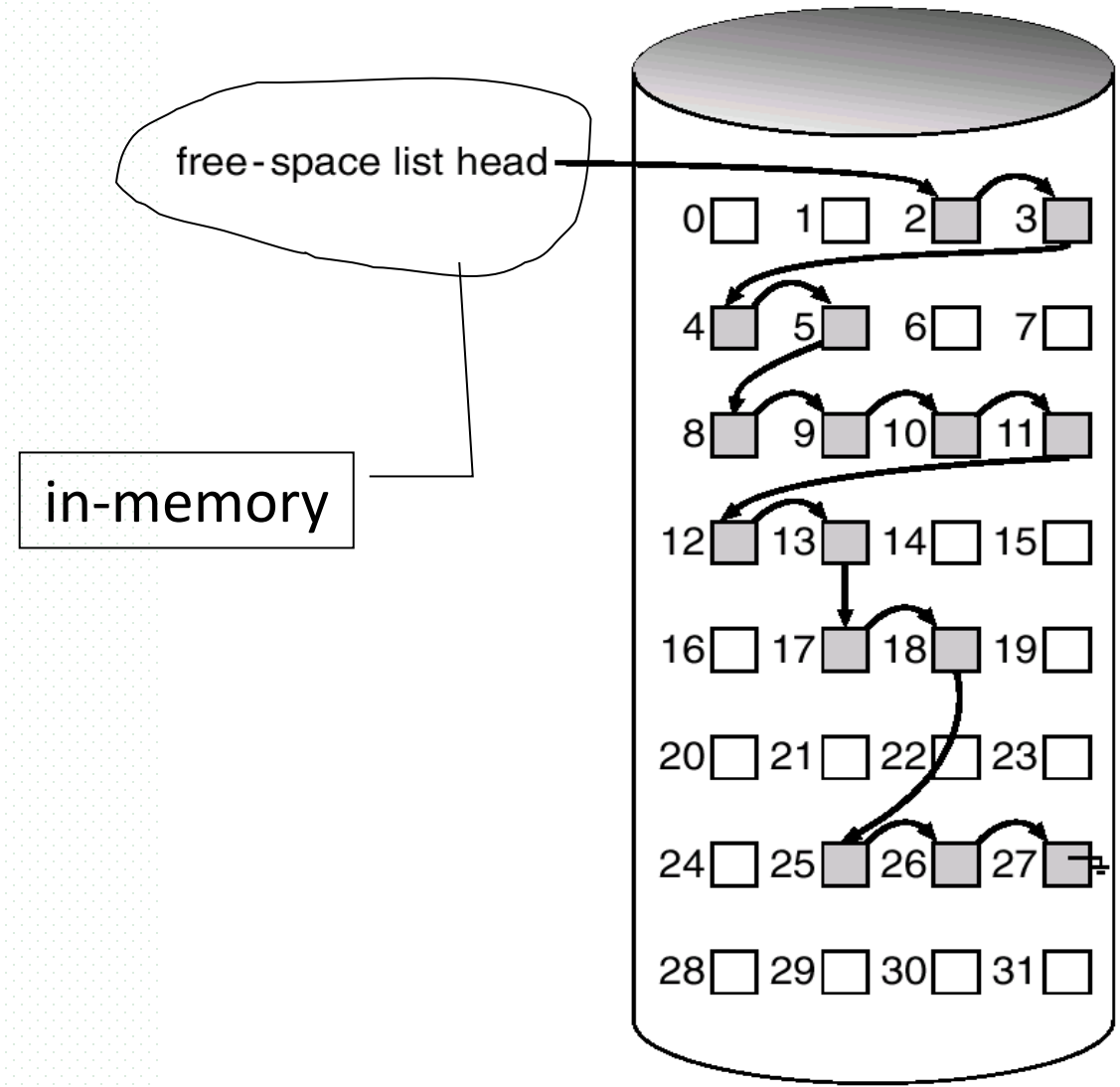
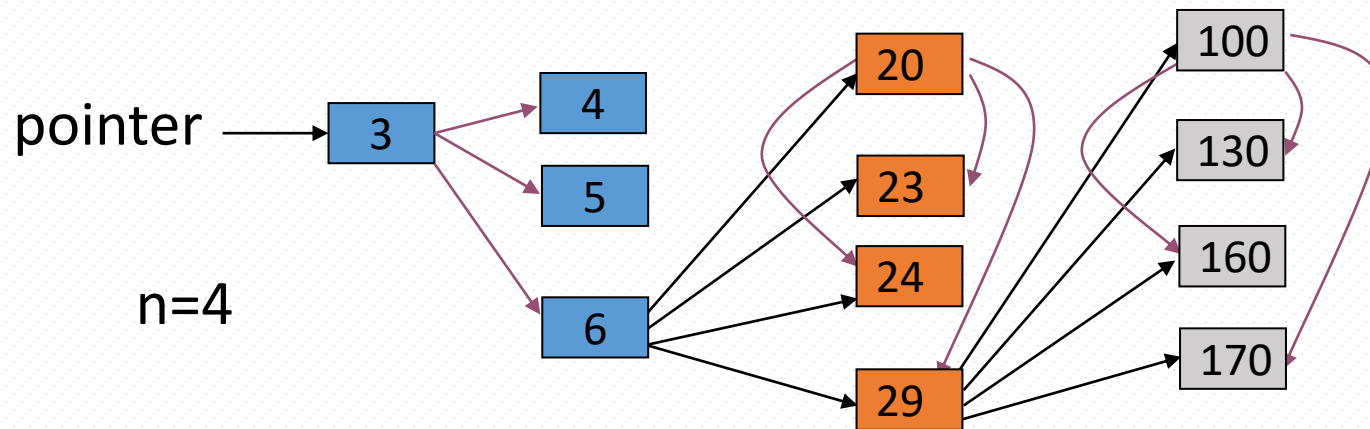


Fig. 11.10 Linked free space list on disk

Grouping

■ Principles

- ❑ n free blocks are grouped, which may be contiguous, or noncontiguous but adjacent
- ❑ the first block store the address of the other $n-1$ free blocks in its group
- ❑ the last free block in one group records the addresses of the blocks in another group

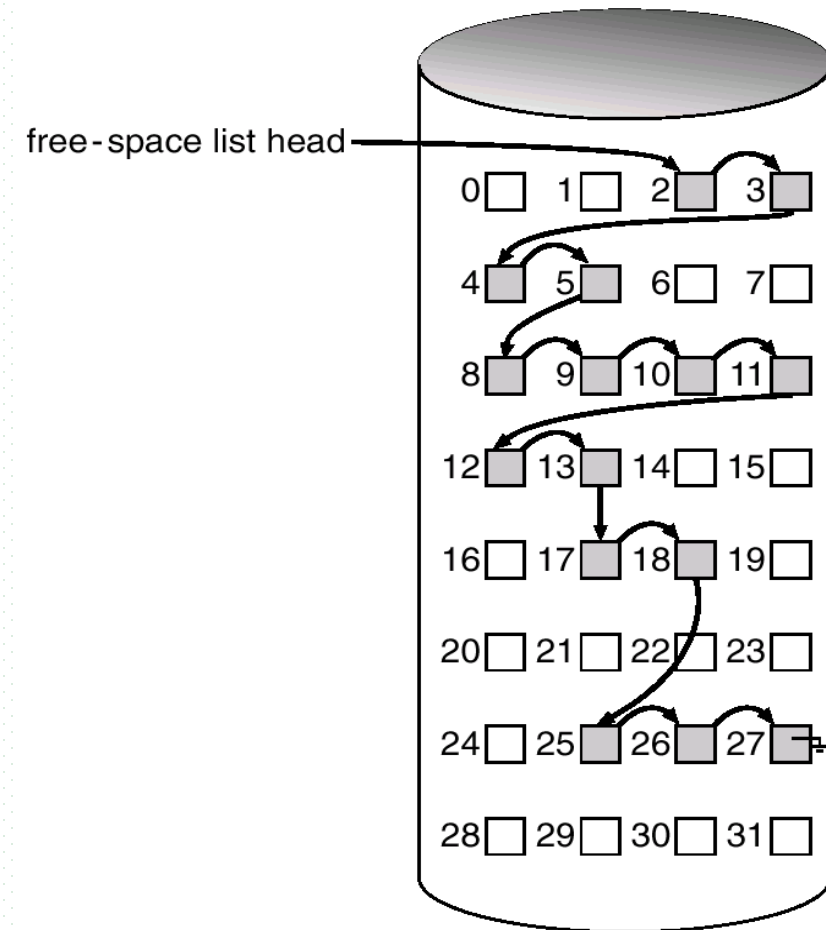
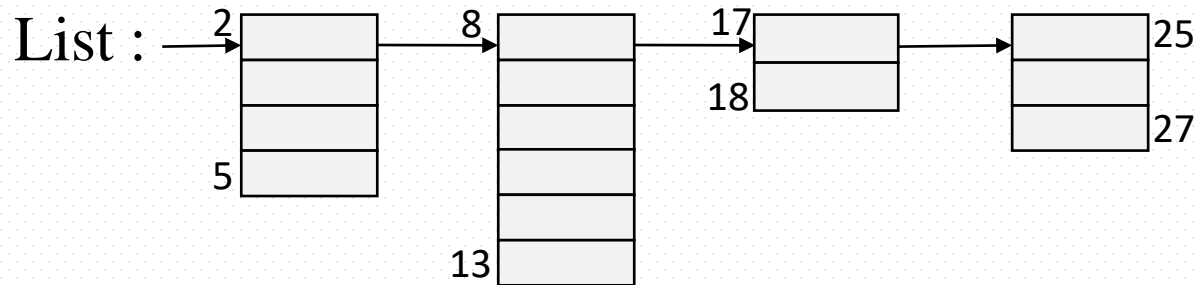


Counting

■ Principles

- several contiguous blocks in one group are allocated or freed simultaneously
- each entry in free space list records **the address of** the first block in the group and **the number** of free blocks in this group
- allocation methods
 - best fit, first fit, worst fit

2: 4	8: 5	17: 2	25: 3
------	------	-------	-------



11.6 Efficiency and Performance

- The disk is a major bottleneck in system performances
- Various disk allocation and directory management mechanisms make effects on efficiency and performance of disk usage
- Techniques used to improve the efficiency and performance

408-18, D

11.6.1 Efficiency

- Efficient usage of disk space ***depends on***

- disk allocation and directory algorithms

- e.g.1 contiguous allocation and disk fragments
- e.g.2 in Unix, **inode** (索引|节点, i.e. FCB) is preallocated on a partition, even for a empty disk; allocation algorithm and free-space algorithm keep a file's data block near that file's **inode** block

waste of space, but efficient

31. 下列优化方法中，可以提高文件访问速度的是_____。

I . 提前读

II . 为文件分配连续的簇

III . 延迟写

IV .采用磁盘高速缓存

A . 仅 I 、 II

B . 仅 II 、 III

C . 仅 I 、 III 、 IV

D . I 、 II 、 III 、 IV

Efficiency

- types of data kept in file's directory entry
 - e.g. "*last access date*" attribute in directory entries

whenever a file is read, this attribute (or the whole directory entry) should be read into memory, be changed and then written back to disk;

inefficient for frequently accessed file



11.6.2 Performance

- How to improve performances for the selected algorithms
- The approaches include
 - cache
 - object: to reduce the number of access on disks
 - asynchronous write
 - optimizing for sequential access
 - free-behind
 - read-ahead
 - RAM/Virtual disk

Cache

- Method 1— *on-board cache in disk controllers*
 - ▣ track buffer **in device controller** for storing entire track at a time
 - ▣ once a seek is performed, *the track starting at the sector under the disk head* is read into the disk cache
 - ▣ the disk controller then transfers any request of the disk sector in this track to OS, and OS may cache the block in the **block buffer**

on-board cache: set of blocks

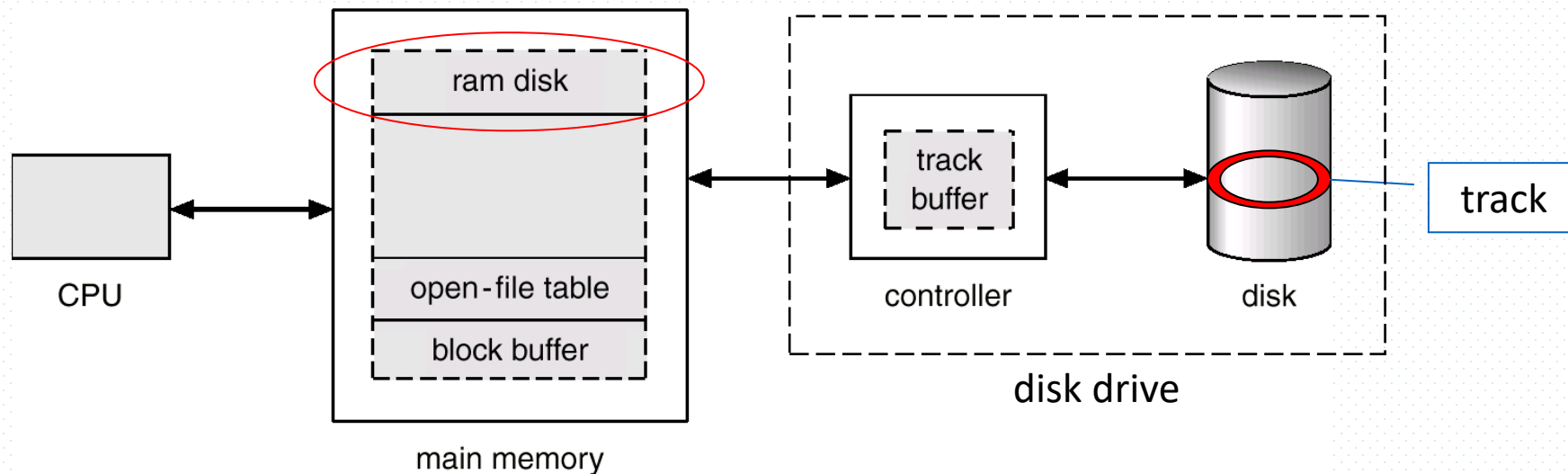


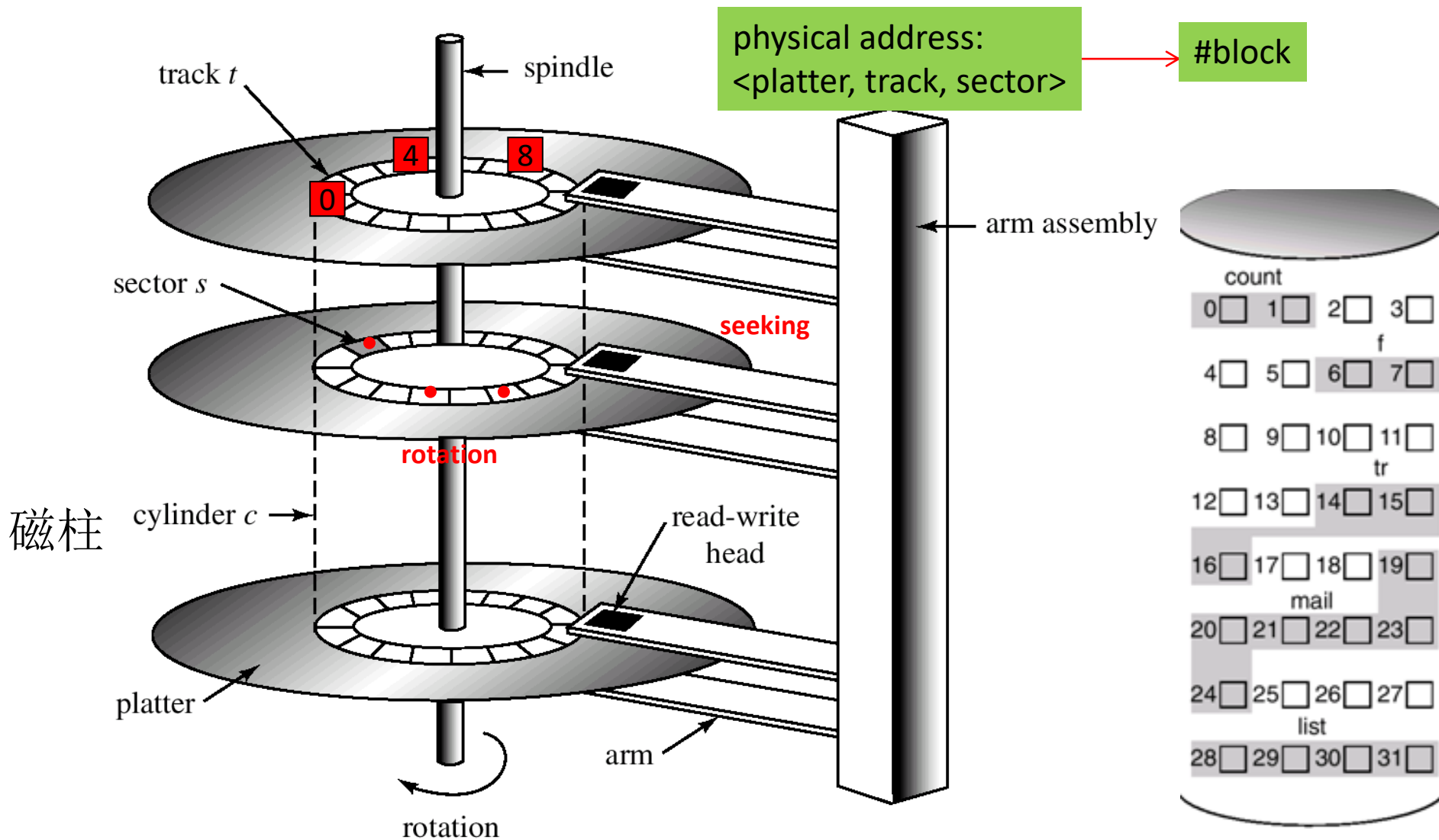
Fig. 11.11.0 on-board in device controller

Fig. Moving-head disk mechanism

1-dimension block-based organization:
block0, block1, ..., block n

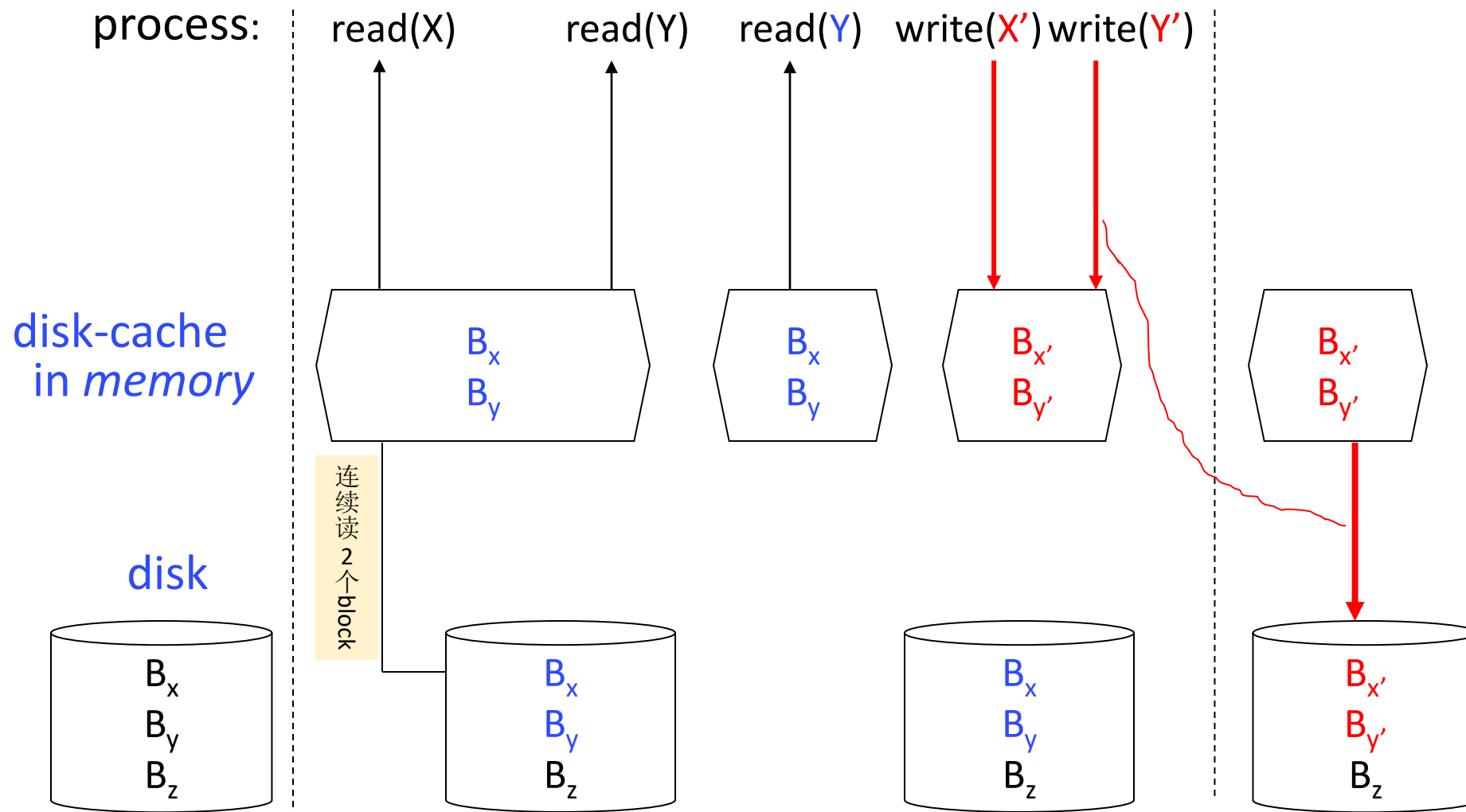
0 1 2 3 4 5 6 7 8

parallel access to the record 0, 1, 2 by the three r/w heads



■ Method 2 — disk cache in main memory

- ❑ a separate section in main memory, called ***disk cache or block cache***, is set for frequently used blocks, assuming that the blocks in buffer will be used shortly
- ❑ e.g. refer to Fig. 11.11.1
- ❑ a process (e.g. DBS users) writing to disk simply writes into the cache, and OS asynchronously writes the data to disk when convenient
 - this scheme is similar to that in database systems
- ❑ caching is in units of blocks



B_x : the block in which data item X resides

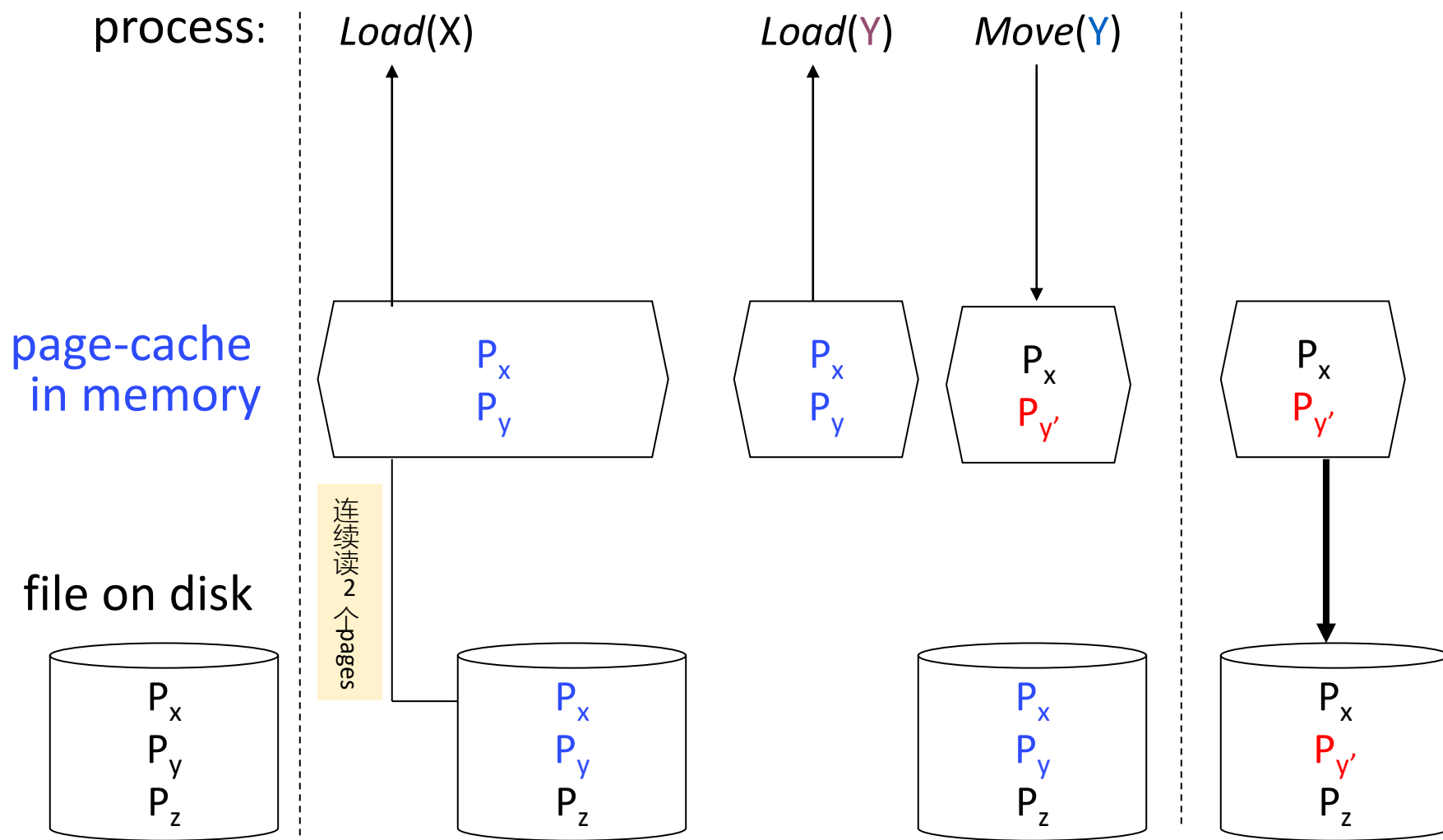
Fig. 11.11.1 Disk cache and *asynchronous* writes

■ Method 3— page cache

- ❑ on the basis of virtual memory techniques and **memory-mapped file**, file data is cached as **pages** rather than disk **blocks**, through **page cache**
- ❑ page cache, *refer to* and Fig.11.11.2 *in next slide* and Fig.11.11
 - a set of pages/frames in main memory
- ❑ the file is mapped into the process' virtual address space, e.g. **memory-mapped files** in §10.3.2, and the virtual address is used to cache file data

■ Unified virtual memory

- ❑ page caching is used to cache both process pages and file pages
- ❑ e.g. Solaris, Linux, Windows NT and 2000



P_x : the page in which data item X resides

Fig. 11.11.2 Page cache and asynchronous writes

Cache

- **Memory-mapped I/O** operations can be implemented by means of page cache
 - ▣ memory-mapped I/O
 - the addresses of **device registers** in device controllers (i.e., corresponding to disk blocks on disks) are mapped into main memory address space, the access on these memory addresses by **memory-access instructions** causes I/O operations on file data through these registers
 - ▣ the number of data buffering in memory are reduced
/*减少数据在内存中的缓存次数，提高I/O访问速度
 - ▣ whereas the normal I/O operations, i.e., routine I/O is implemented by I/O system calls (e.g. write() and read()) on the file system using the **buffer (disk) cache**

淘宝平台双11:
用户态I/O?

■ Method 4 — Non-unified buffer cache

- ❑ *refer to Fig. 11.11*
- ❑ buffer cache/block cache is in units of blocks, and is not visible to processes
- ❑ page cache is in units of **pages**, and is in process virtual address space, accessed by memory access instructions
- ❑ I/O using write() and read() system calls go through buffer cache, and is in unit of **blocks**

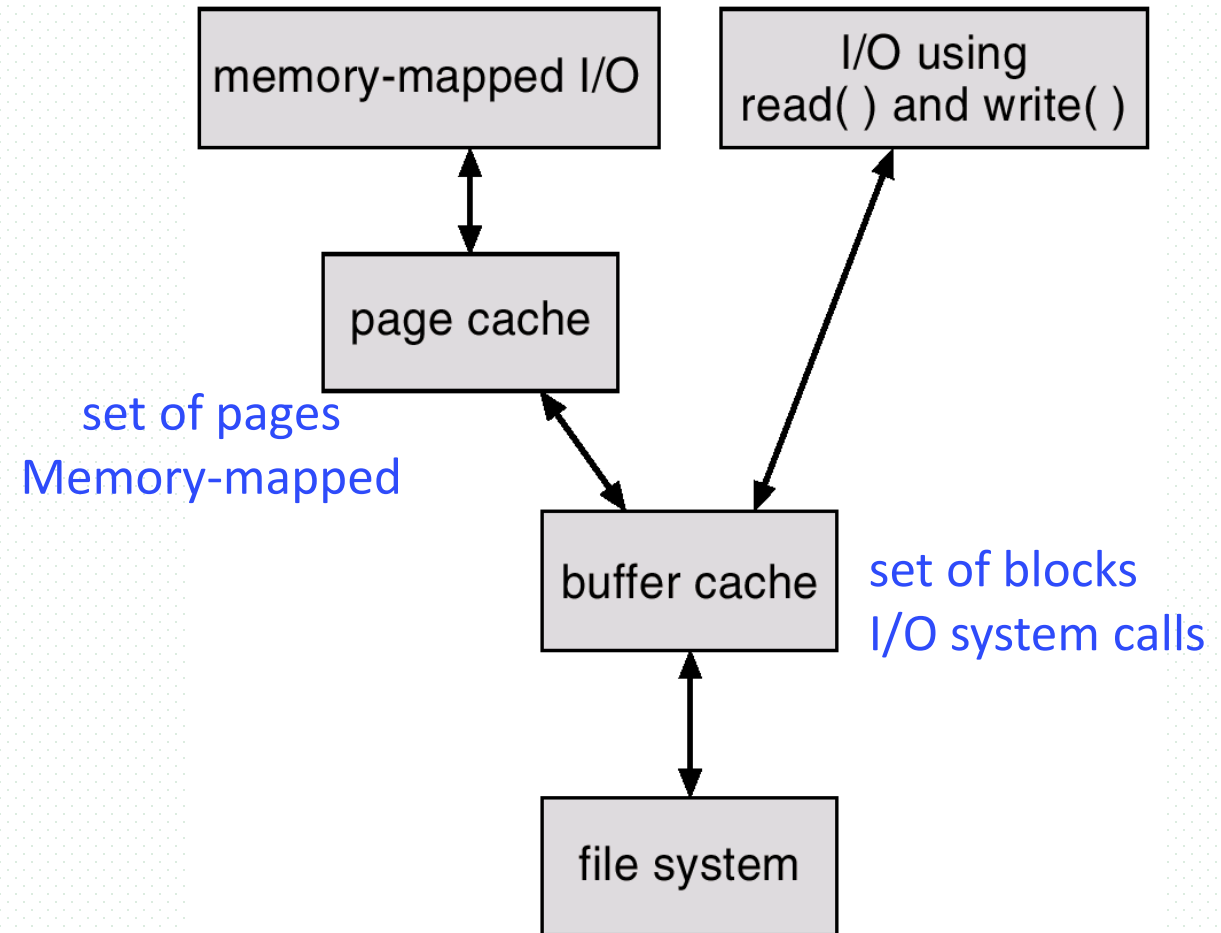


Fig. 11.11 I/O Without a Unified Buffer Cache

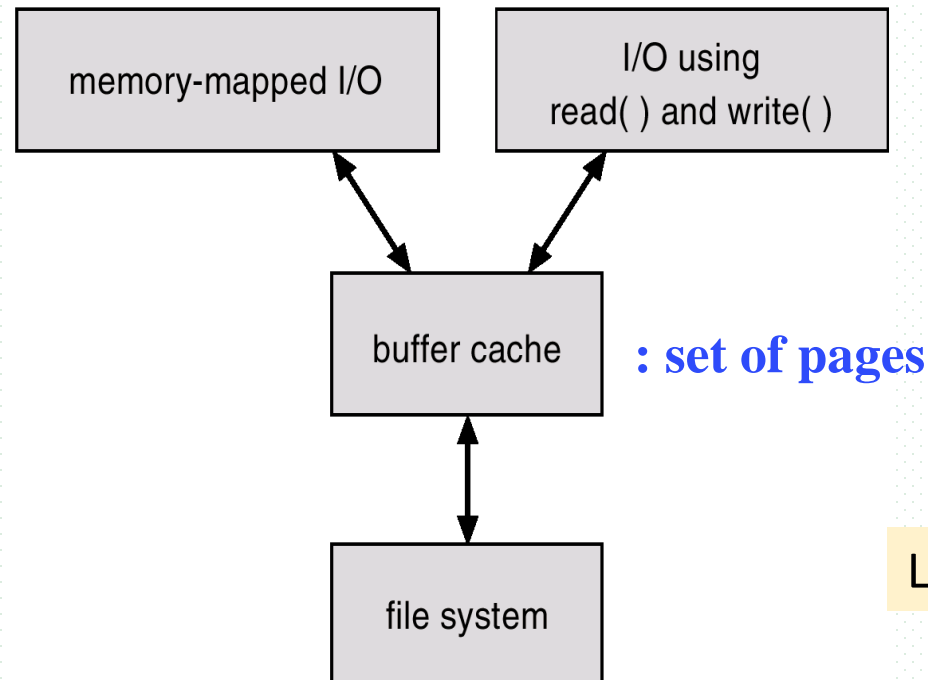
Cache

- ❑ memory-mapped I/O is on the basis of page cache, in units of pages, proceeding by
 - reading in-disk file blocks into buffer cache
 - because VM can not interface with buffer cache, file blocks in buffer cache are then copied to page cache for processes to access
- ❑ demerits
 - double caching, i.e. caching file data twice, can not be avoided, resulting in
 - wasteful of memory and CPU resources
 - inconsistency between two caches

Cache

■ Method 5 — Unified buffer cache

- ❑ *refer to Fig. 11.12*
- ❑ page cache is in units of pages, and is visible to both processes and I/O system calls (e.g. write() and read())
- ❑ a unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O




Linux 2.4及其后各版本采用

Fig. 11.12 I/O Using a Unified Buffer/**Page** Cache

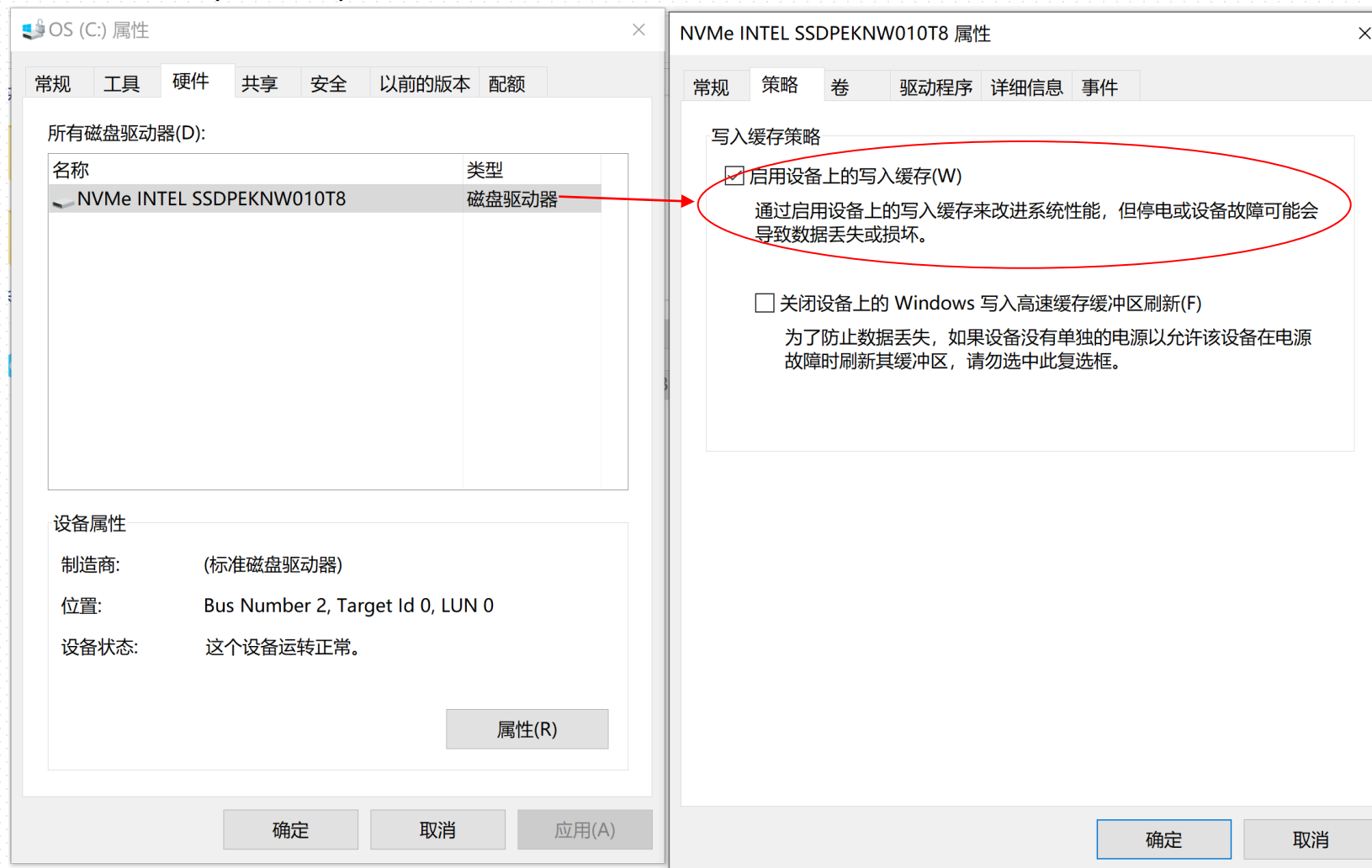
Asynchronous Writes

- A process writing to disks simply writes data into the cache in memory, and the system asynchronously writes the data to disk (刷盘) when convenient

- e.g. 磁盘写入缓存

- e.g. DBMS  in Fig.11.1.1

- The process can have very fast data writing



Asynchronous Writes

- When the data cached in memory is written back to the disk?
 - when users, e.g. do the 'save' operation on the PPT documents
 - in DBS, at the times of **checkpoints** (检查点), conducted by DBMS



Optimizing for Sequential Access

- Free-behind

- remove the **i-th page** from the buffer as soon as the next page (i.e. **(i+1)-th** page) is requested, because it is supposed that **the i-th page** may not be used later

- Read-ahead

- a request page i.e., **i-th page**, and several subsequent pages , e.g. **(i+1)-th** page, are read and cached

RAM/Virtual disk (内存盘)

- Improve PC performance by dedicating a section of memory as virtual disk, or RAM disk
 - ▣ e.g. RAM disk in Linux
- Fast disk operations are provided for users
- Linux内核自动支持ramdisk
 - ▣ 默认创建16个ramdisks设备，分别是 /dev/ram0 -- /dev/ram15，但未启用，不占用内存空间
 - ▣ 使用命令“ls -al /dev/ram*”可查看ramdisk设备

```
[root]# ls -l /dev/ram*
lrwxrwxrwx 1 root root 4 Jun 12 00:31 /dev/ram -> ram1
brw-rw---- 1 root disk 1, 0 Jan 30 2003 /dev/ram0
brw-rw---- 1 root disk 1, 1 Jan 30 2003 /dev/ram1
brw-rw---- 1 root disk 1, 10 Jan 30 2003 /dev/ram10
brw-rw---- 1 root disk 1, 11 Jan 30 2003 /dev/ram11
brw-rw---- 1 root disk 1, 12 Jan 30 2003 /dev/ram12
brw-rw---- 1 root disk 1, 13 Jan 30 2003 /dev/ram13
brw-rw---- 1 root disk 1, 14 Jan 30 2003 /dev/ram14
brw-rw---- 1 root disk 1, 15 Jan 30 2003 /dev/ram15
brw-rw---- 1 root disk 1, 16 Jan 30 2003 /dev/ram16
brw-rw---- 1 root disk 1, 17 Jan 30 2003 /dev/ram17
brw-rw---- 1 root disk 1, 18 Jan 30 2003 /dev/ram18
brw-rw---- 1 root disk 1, 19 Jan 30 2003 /dev/ram19
brw-rw---- 1 root disk 1, 2 Jan 30 2003 /dev/ram2
brw-rw---- 1 root disk 1, 3 Jan 30 2003 /dev/ram3
```

查看创建的内存盘

```
1 | ls /dev/ram*
```

磁盘设备格式化

```
1 | mkfs.ext4 /dev/ram0
```

创建挂载设备，进行挂载

```
1 | mount /dev/ram0 /log
```

后续卸载

```
1 | 1. umount
2 | umount /log
3 | umount /dev/ram0
4 |
5 | 2. 移出内核
6 | modprobe -r brd
```




Thanks for your
attention



北京邮电大学