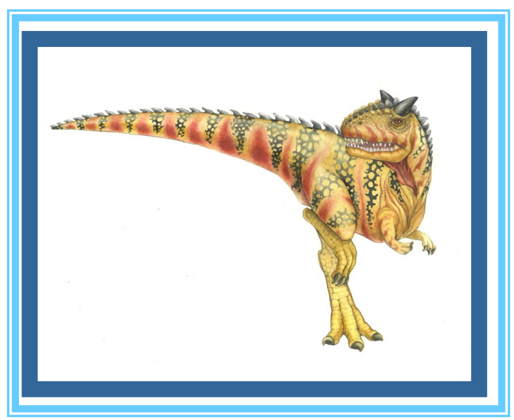
Chapter 11: File System Implementation



Chapter 11: File System Implementation

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery
- NFS
- Example: WAFL File System

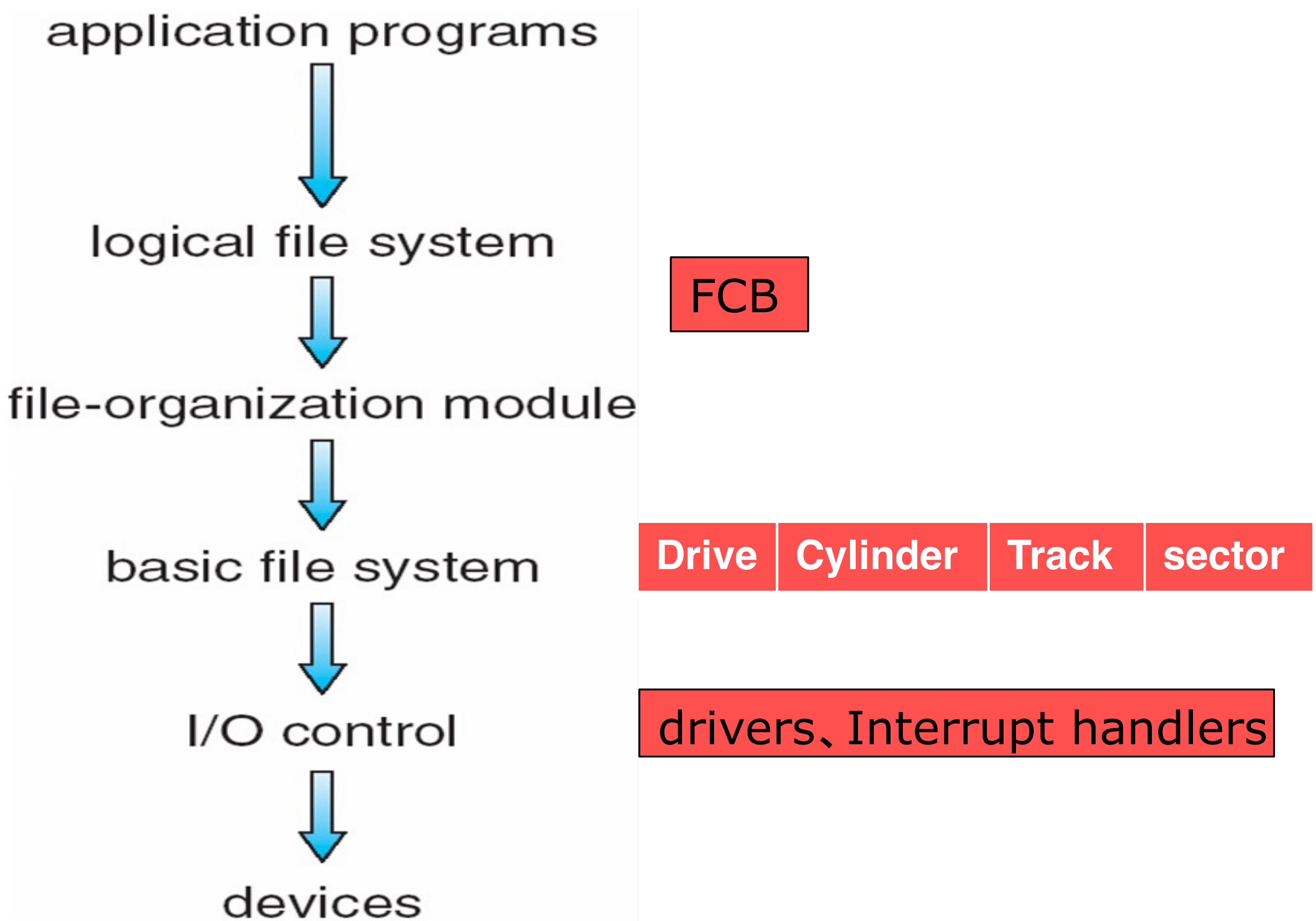
Objectives

- 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

File-System Structure

- File structure
  - Logical storage unit; Collection of related information
- File system organized into layers
- File system resides on secondary storage (disks)
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- File control block – storage structure consisting of information about a file
- Device driver controls the physical device

Layered File System



File-System Implementation

- Boot control block contains info needed by system to boot OS from that volume
- Volume control block contains volume details
- Directory structure organizes the files
- Per-file File Control Block (FCB) contains many details about the file

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

A Typical File Control Block

Implementing File Operations

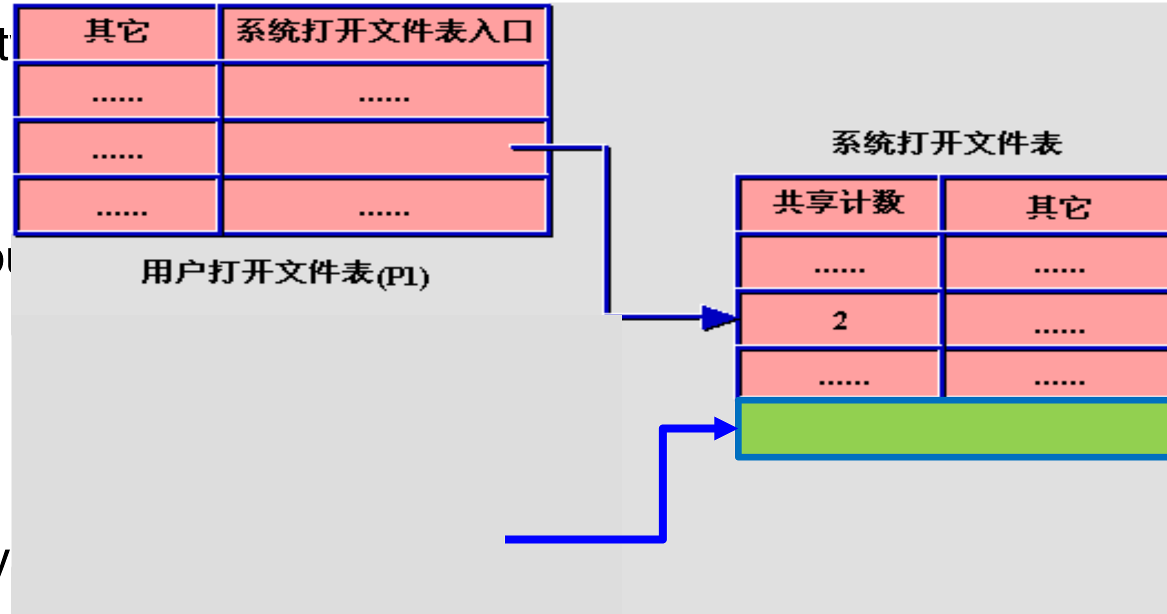
- Create a file:
  - Find space in the file system, add directory entry.
- Open file
  - System call specifying name of file. system searches directory structure to find file.
  - 根据文件号查系统打开文件表，看文件是否已被打开；
    - ✓ 是→共享计数加1
    - ✓ 否→将外存中的FCB等信息填入系统打开文件表空表项，共享计数置为1；
  - 在用户打开文件表中取一空表项，填写打开方式等，并指向系统打开文件表对应表项
  - System keeps ***current file position pointer*** to the location where next write/read occurs
  - System call returns ***file descriptor*** (a handle) to user process
- Writing in a file:
  - System call specifying file descriptor and information to be written
  - Writes information at location pointed by the files current pointer

Implementing File Operations

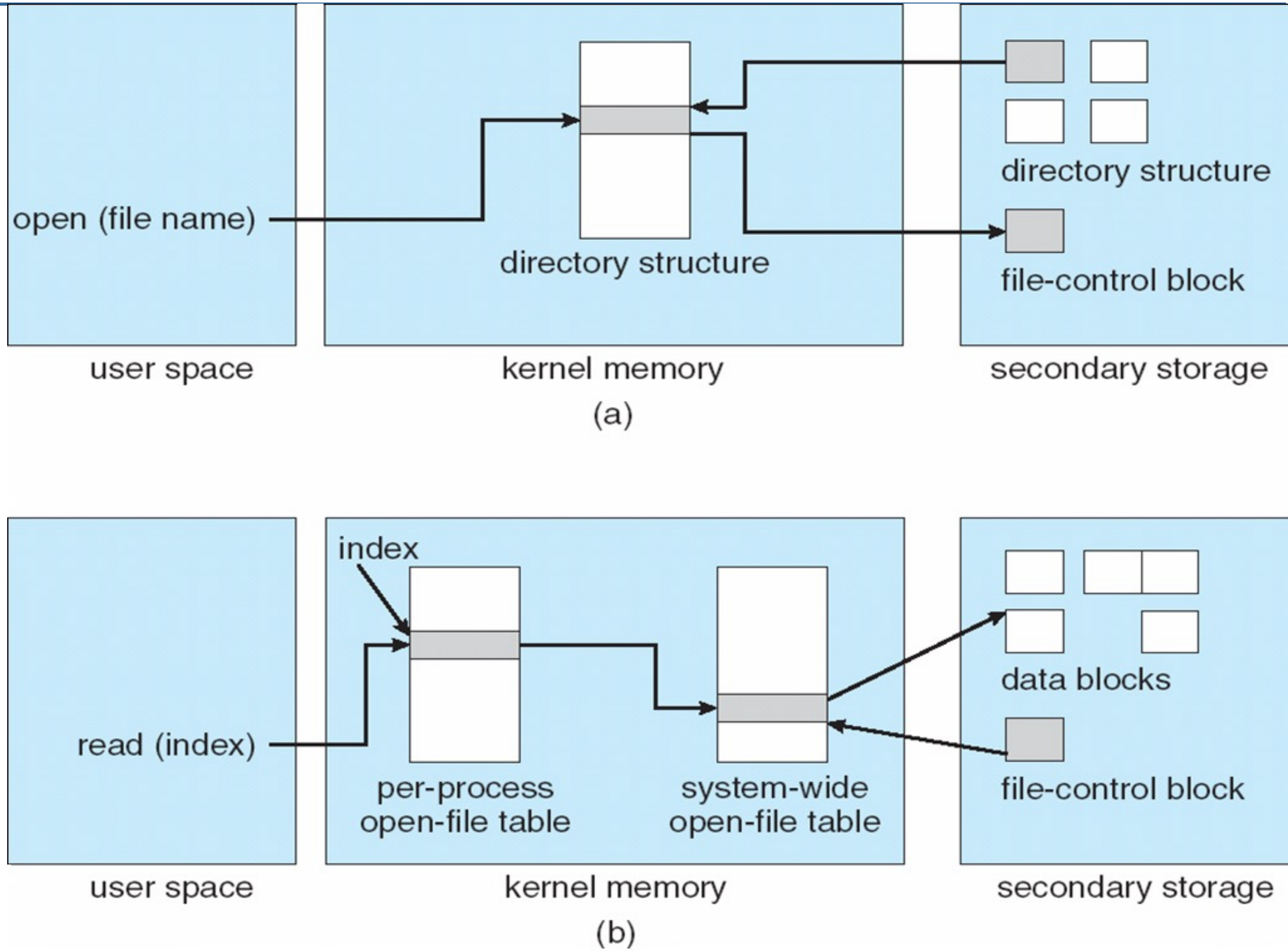
- Reading a file:
  - System call specifying file descriptor and number of bytes to read (and possibly where in memory to stick contents).
- Repositioning within a file:
  - System call specifying file descriptor and new location of current pointer
  - (also called a file ***seek*** even though does not interact with disk)
- Closing a file:
  - System call specifying file descriptor
  - Call removes current file position pointer and file descriptor associated with process and file(**打开文件表**)
  - 若在文件打开期间，该文件作过某种修改，则应将其写回到辅存。
- Deleting a file:
  - Search directory structure for named file, release associated file space and erase directory entry
- Truncating a file:
  - Keep attributes the same, but reset file size to 0, and reclaim file space.

Multiple users of a file

- OS typically keeps t
- Per-process table
  - Information abou … h
    pointer)
- System wide table
  - Gets created by
  - Location of file on disk
  - Access dates
  - File size
  - Count of how many processes have the file open (used for deletion)

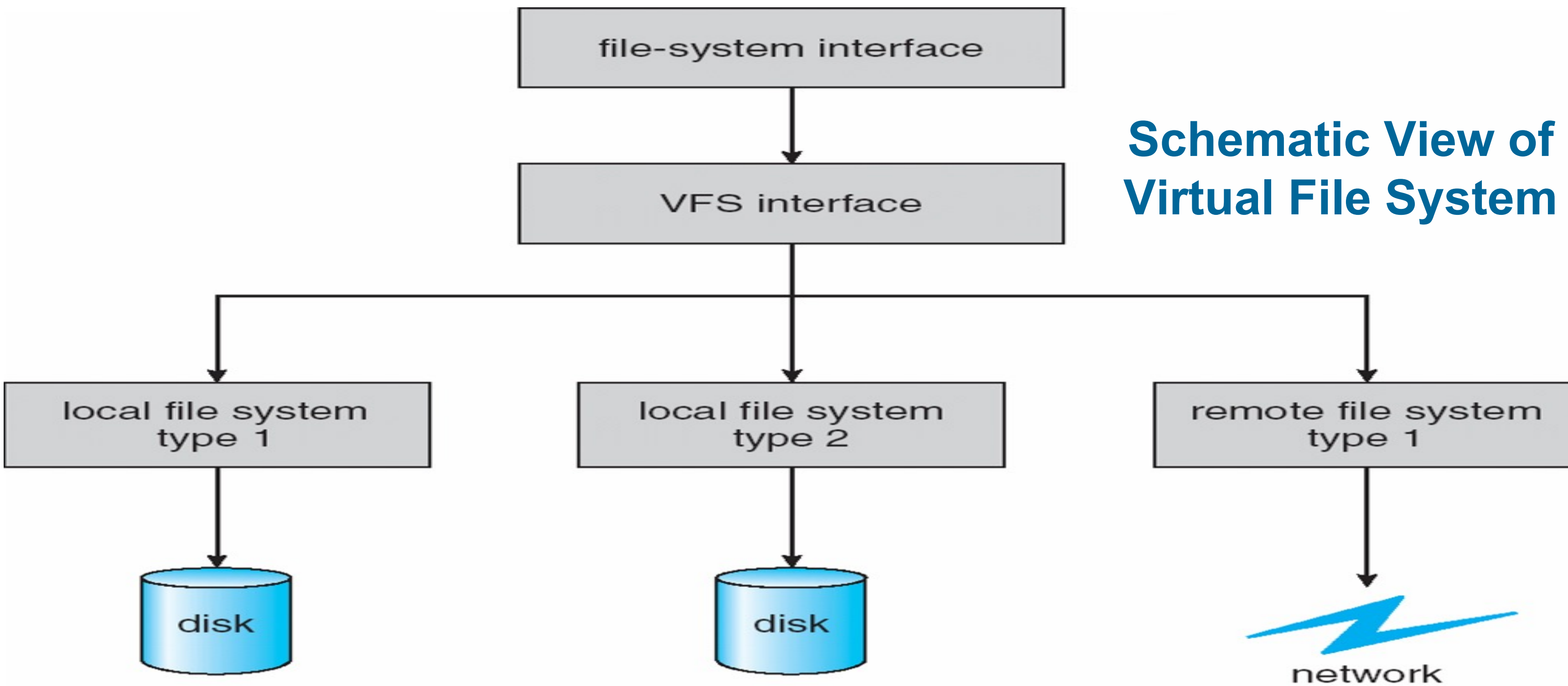


In-Memory File System Structures



Virtual File Systems

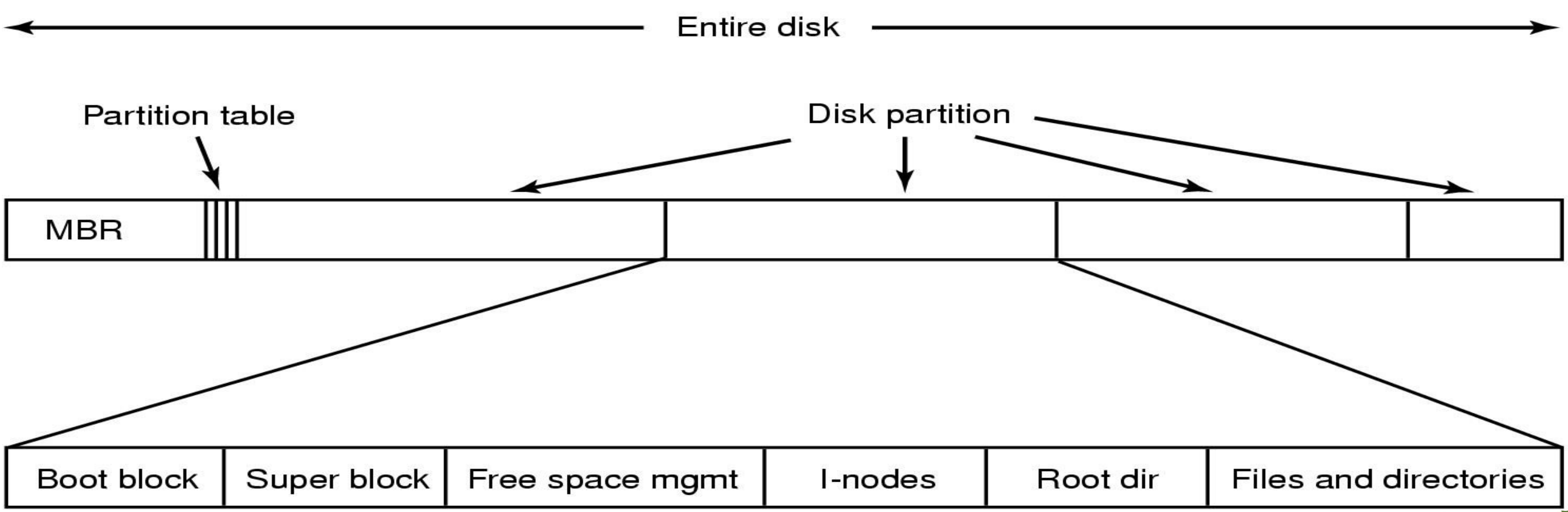
- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.



Schematic View of Virtual File System

File System Layout

- File System is stored on disks
  - Disk is divided into 1 or more partitions
  - Sector 0 of disk called Master Boot Record
  - End of MBR has partition table (start & end address of partitions)
- First block of each partition has boot block
  - Loaded by MBR and executed on boot



Storing Files

Files can be allocated in different ways:

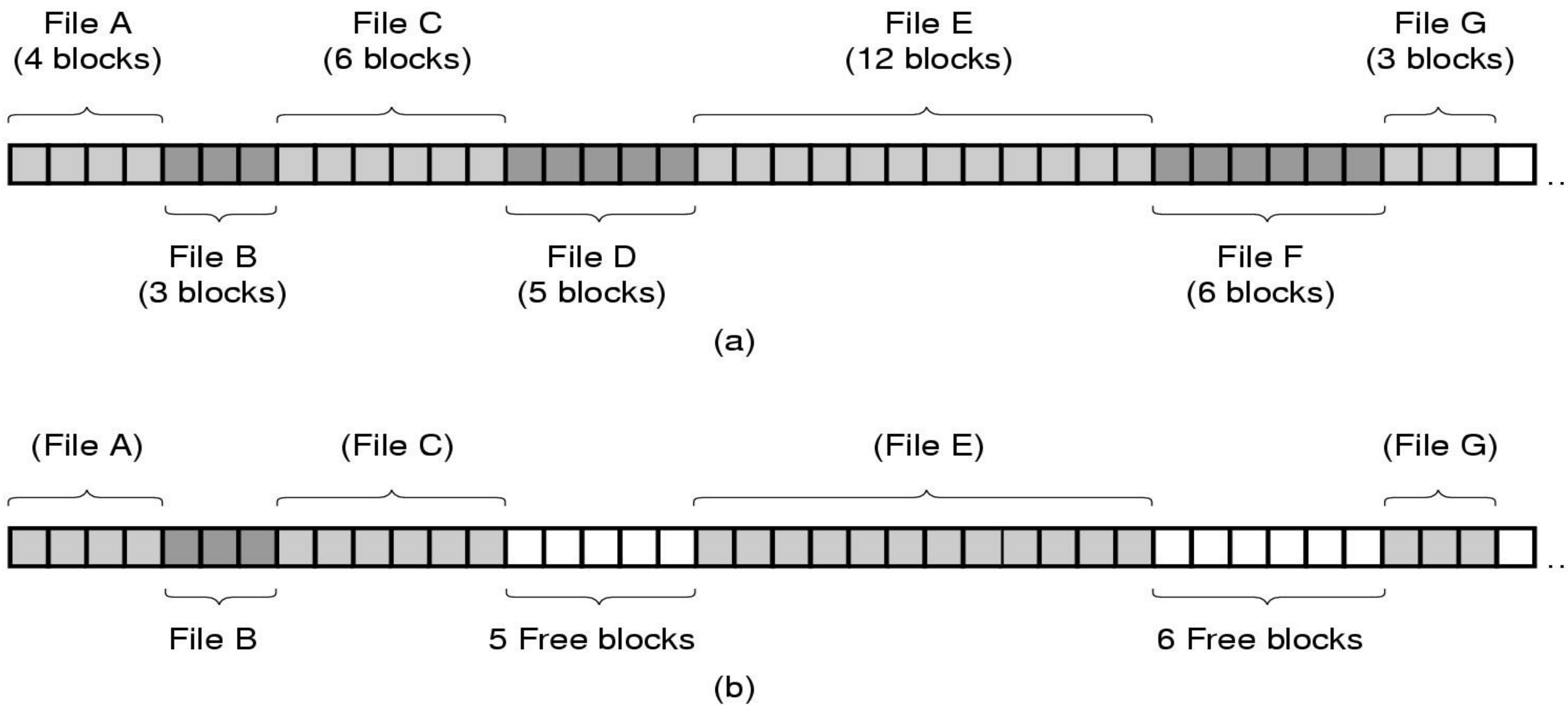
- Contiguous allocation
  - All bytes together, in order
- Linked Structure
  - Each block points to the next block
- Indexed Structure
  - An index block contains pointer to many other blocks

Rhetorical Questions -- which is best?

- For sequential access? Random access?
- Large files? Small files? Mixed?

Implementing Files

- Contiguous Allocation: allocate files contiguously on disk



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

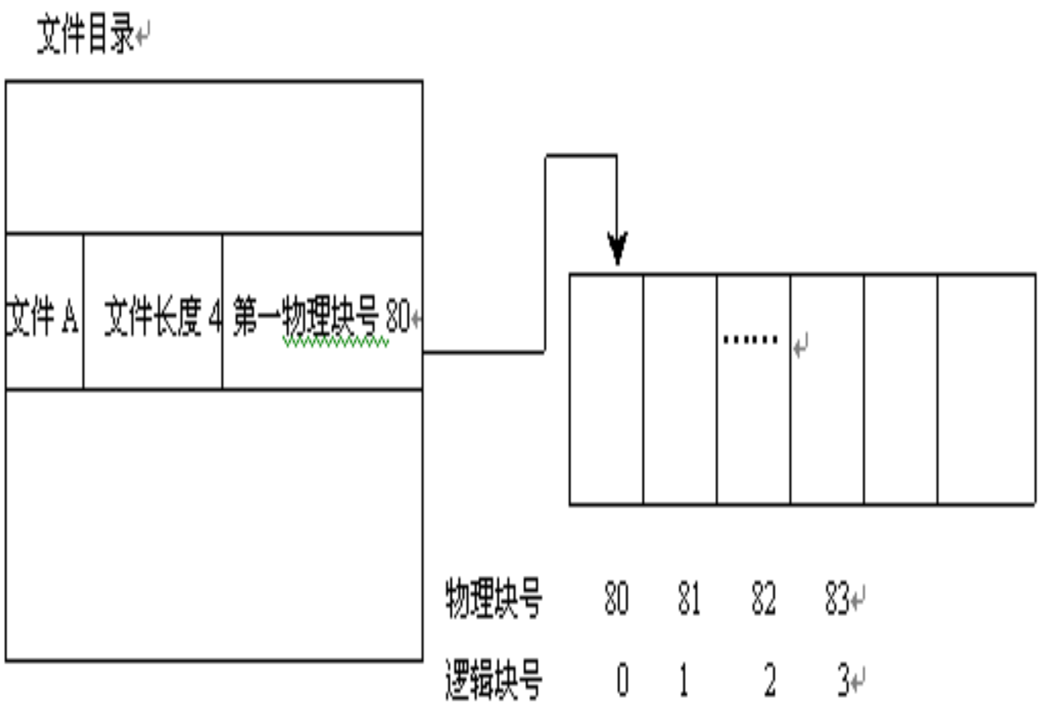
Contiguous Allocation

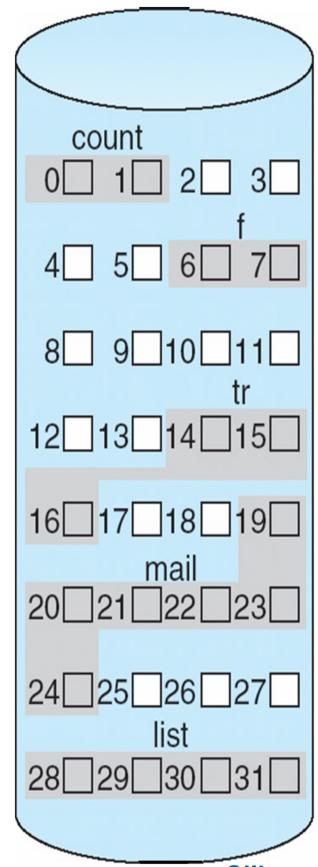
- Mapping from logical to physical

LA/512 → Q块, R位移

Block to be accessed = Q + starting address

Displacement into block = R





directory

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

Contiguous Allocation of Disk Space

Contiguous Allocation

- Pros:
  - Simple: state required per file is start block and size
  - Performance: entire file can be read with one seek
- Cons:
  - Fragmentation: external is bigger problem
  - Usability: user needs to know size of file
- Used in CDROMs, DVDs

Extent-Based Systems

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

Linked Allocation

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

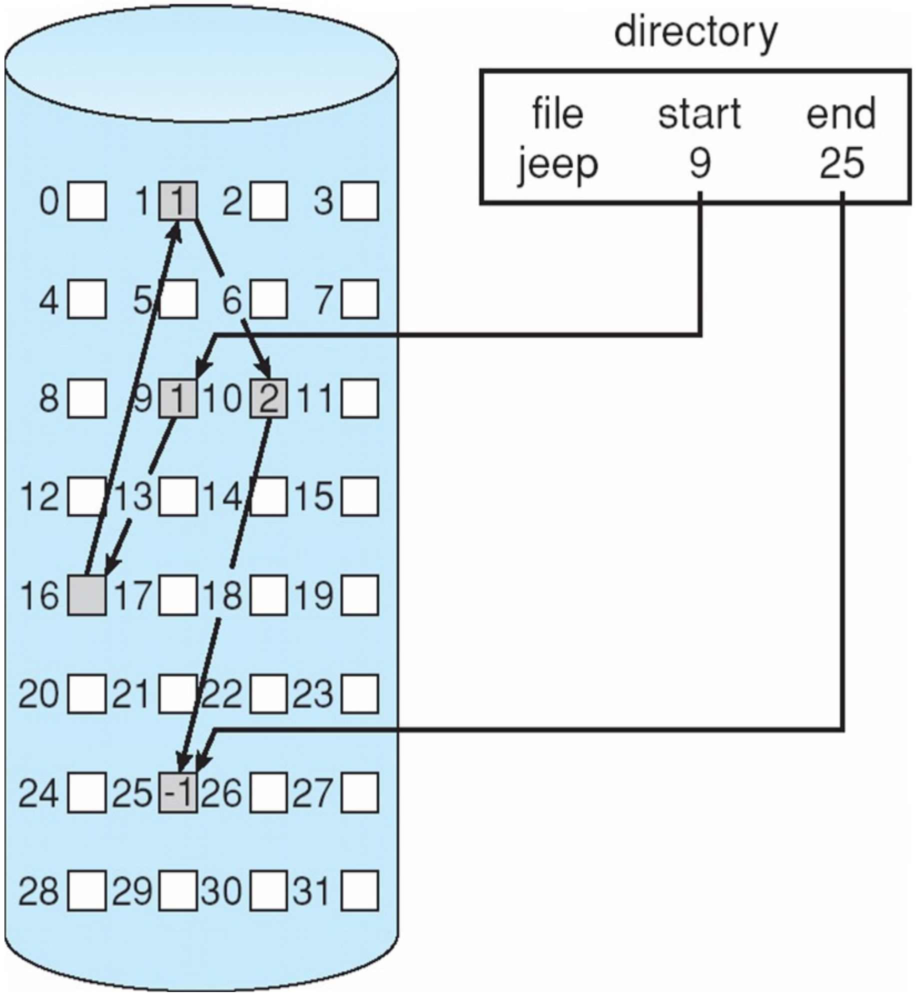
pointer

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

$$LA/511 \begin{cases} Q \\ R \end{cases}$$

- Block to be accessed is the Qth block in the linked chain of blocks representing the file.
- Displacement into block = R + 1(指针占一字节)

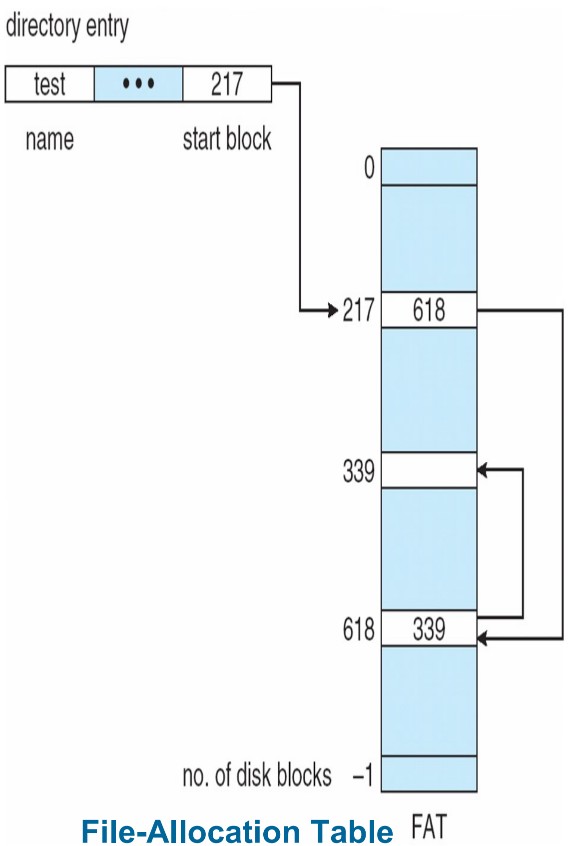
Linked Allocation



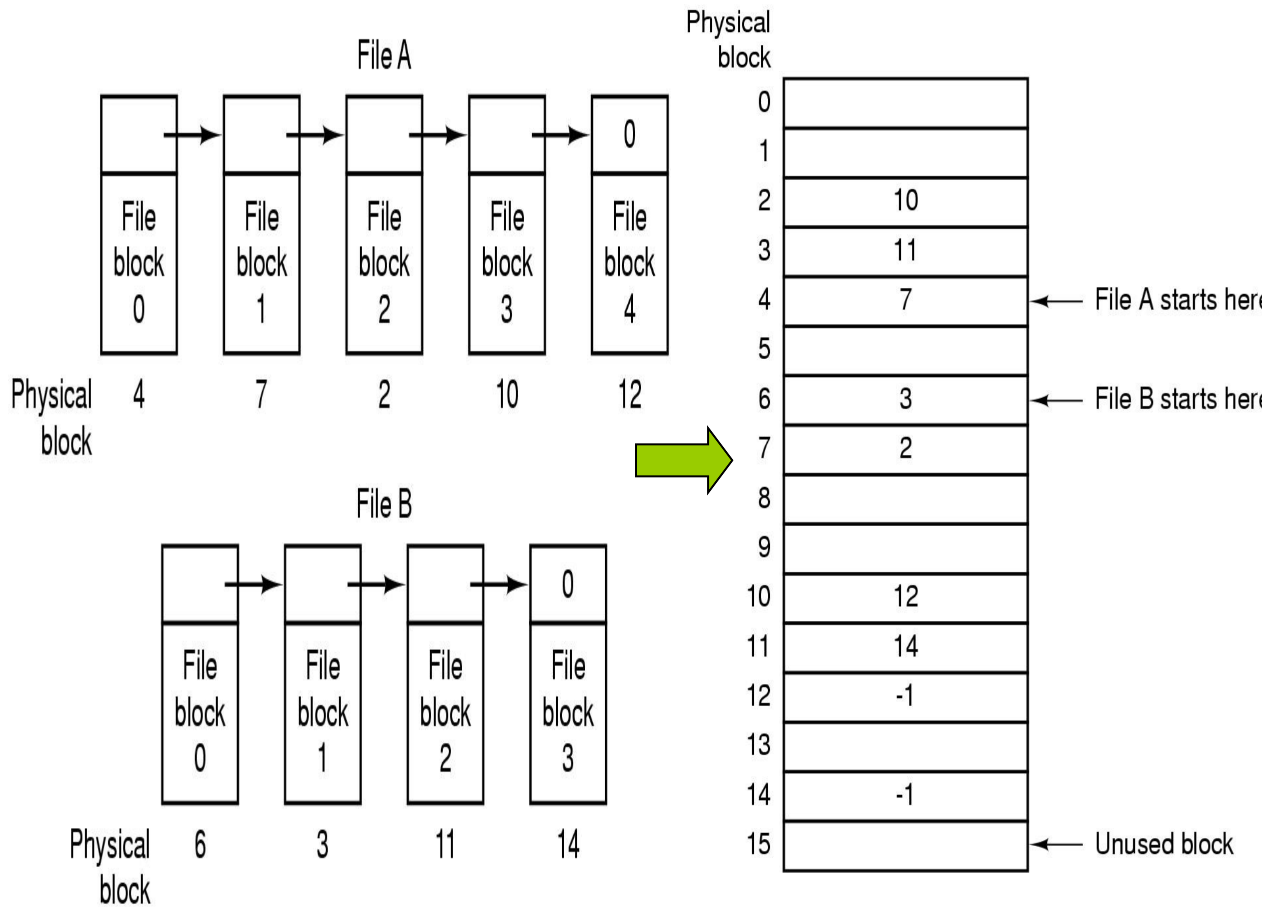
- Pros:
  - No space lost to external fragmentation
  - Disk only needs to maintain first block of each file
- Cons:
  - Random access is costly
  - Overheads of pointers

MS-DOS(OS/2) File System

- Implement a linked list allocation using a table
  - Called File Allocation Table (FAT)
  - Take pointer away from blocks, store in this table
  - Can cache FAT in-memory



File-Allocation Table

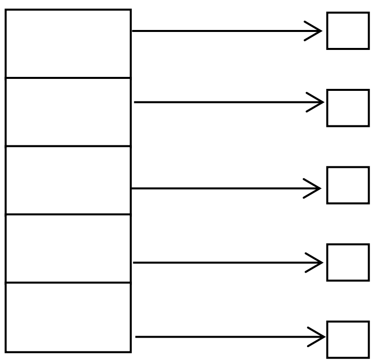


FAT Discussion

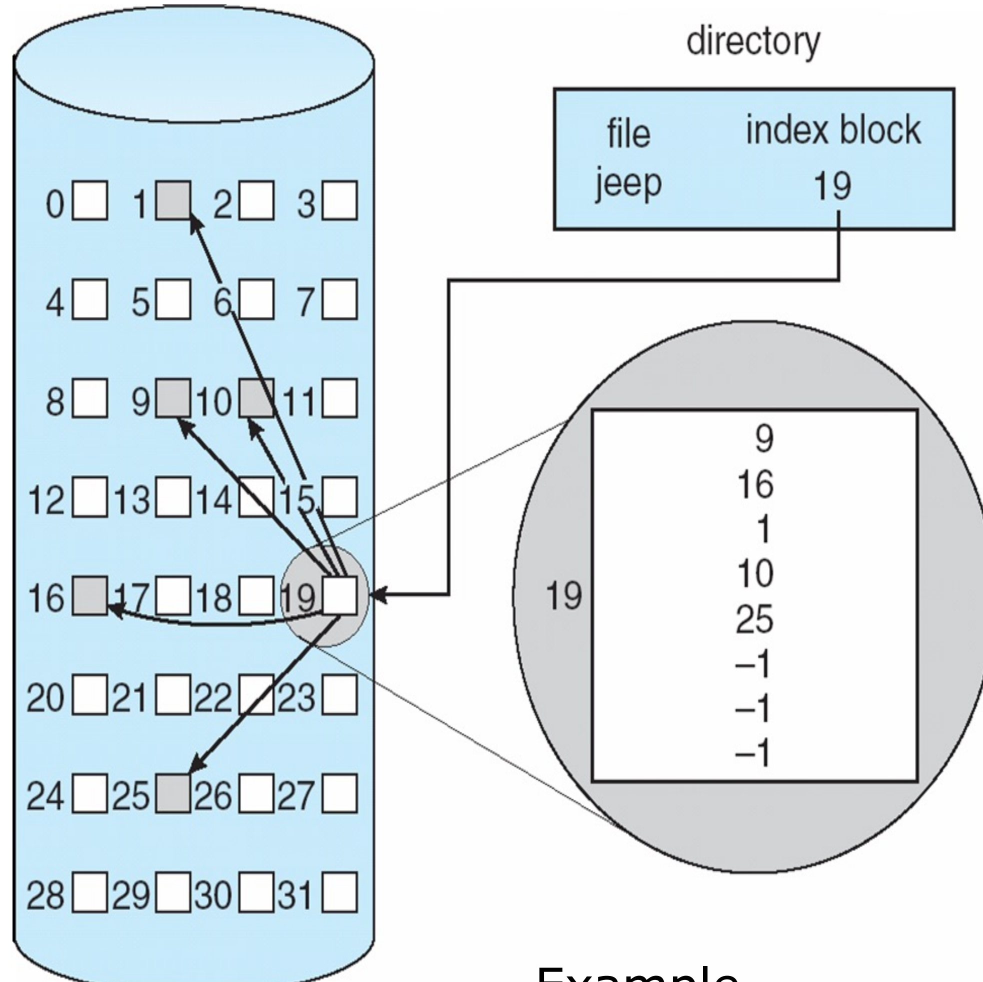
- Pros:
  - Entire block is available for data
  - Random access is faster since entire FAT is in memory
- Cons:
  - Entire FAT should be in memory
    - For 20 GB disk, 1 KB block size, FAT has 20 million entries
    - If 4 bytes used per entry $\Rightarrow$ 80 MB of main memory required for FS

Indexed Allocation

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



index table



Example

Indexed Allocation (Cont.)

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table

$$
LA/512 \begin{cases} Q \\ R \end{cases}
$$

Q = displacement into index table

R = displacement into block

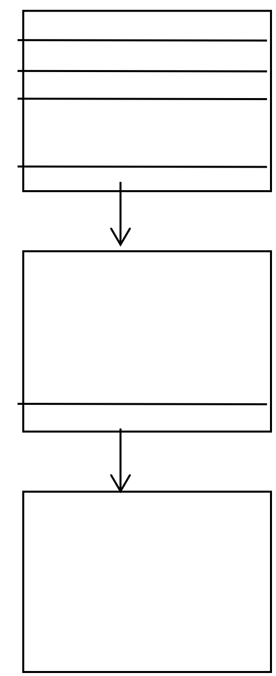
Indexed Allocation – Mapping (Cont.)

- Mapping from logical to physical in a file of unbounded length (block size of 512 words)

扩充的实现方案如下:

1. Linked scheme – Link blocks of index table (no limit on size)

$$LA / (512 \times 511) \rightarrow Q_1, R_1$$



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

$$R_1 / 512 \rightarrow Q_2, R_2$$

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

Indexed Allocation – Mapping (Cont.)

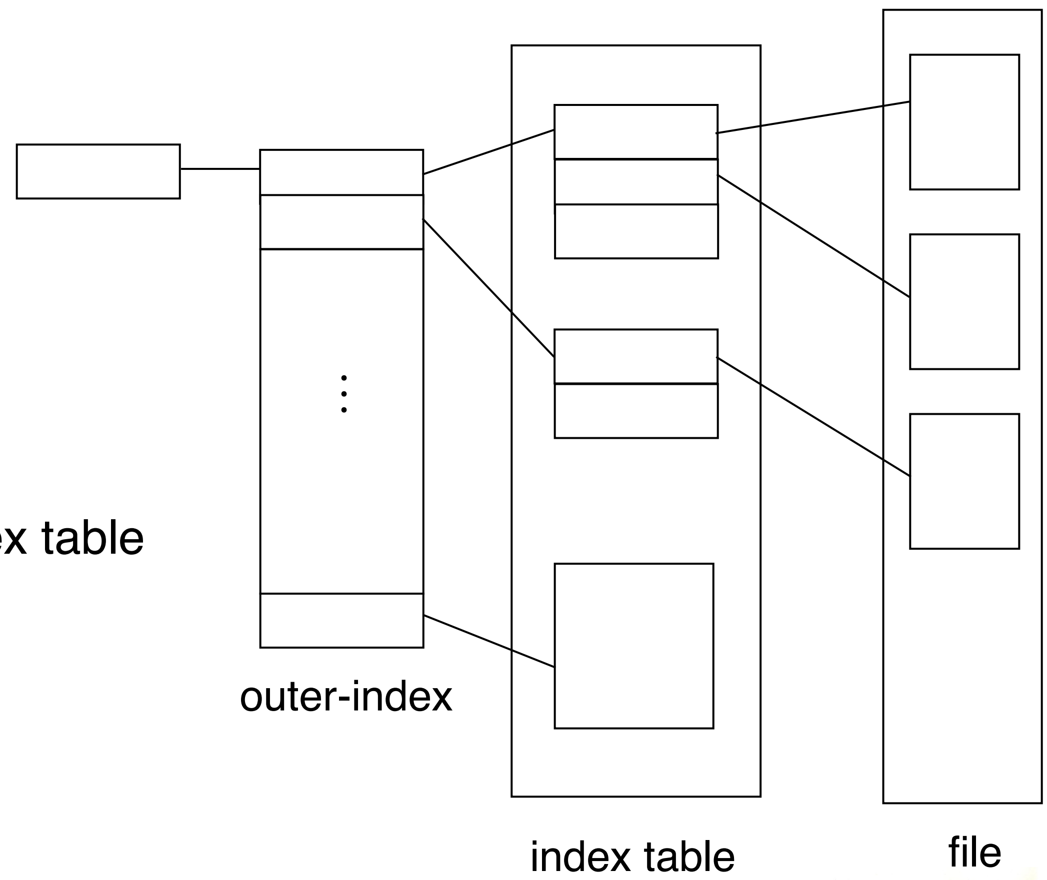
2. Multilevel (for exam. Two-level index)index (maximum file size is 512^3)

$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

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

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

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

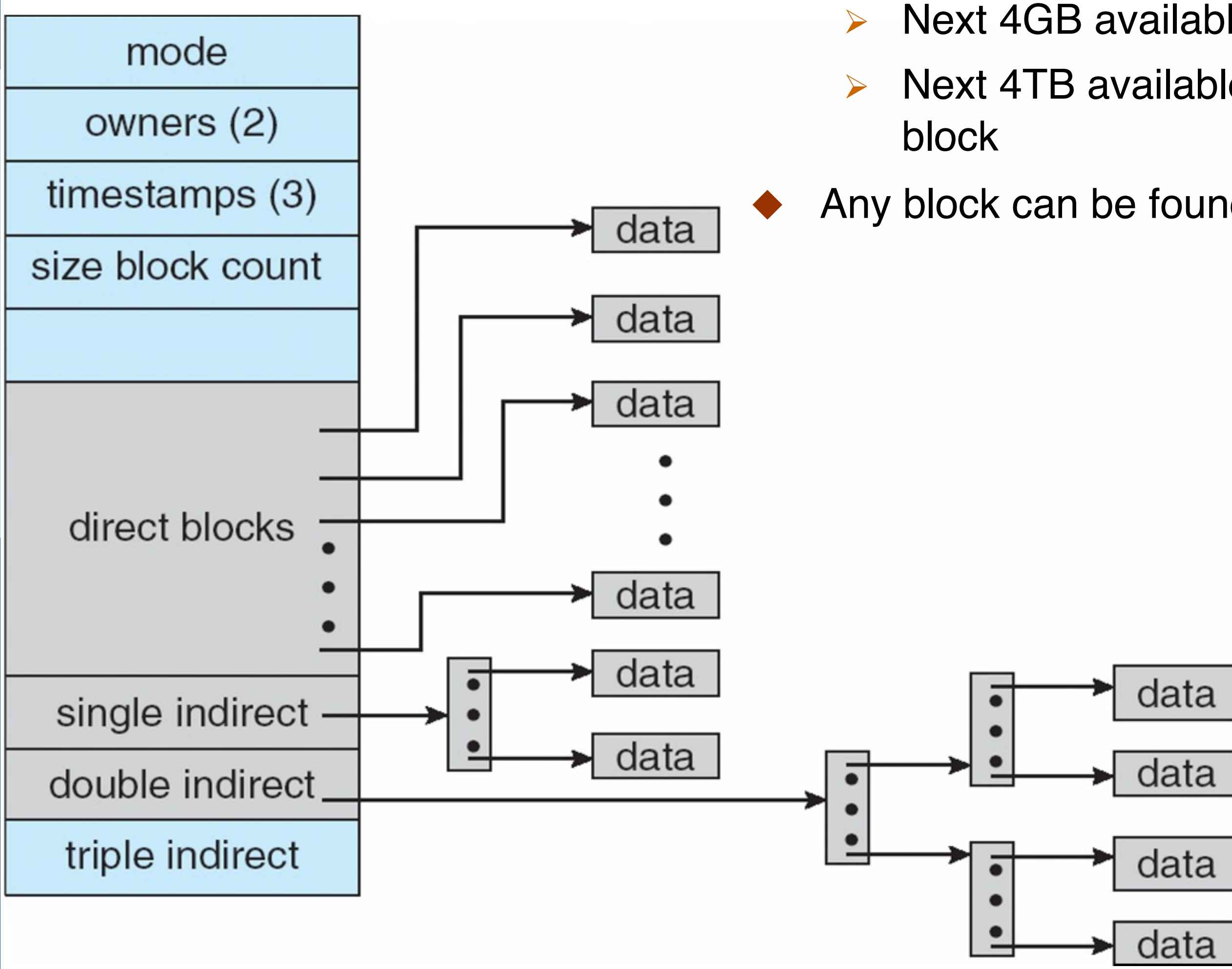


3. Combined Scheme: UNIX UFS (4K bytes per block)

共15个指针项，前

12个是直接指针

- If data blocks are 4K …
  - First 48K reachable from the inode
  - Next 4MB available from single-indirect
  - Next 4GB available from double-indirect
  - Next 4TB available through the triple-indirect block
- Any block can be found with at most 3 disk accesses



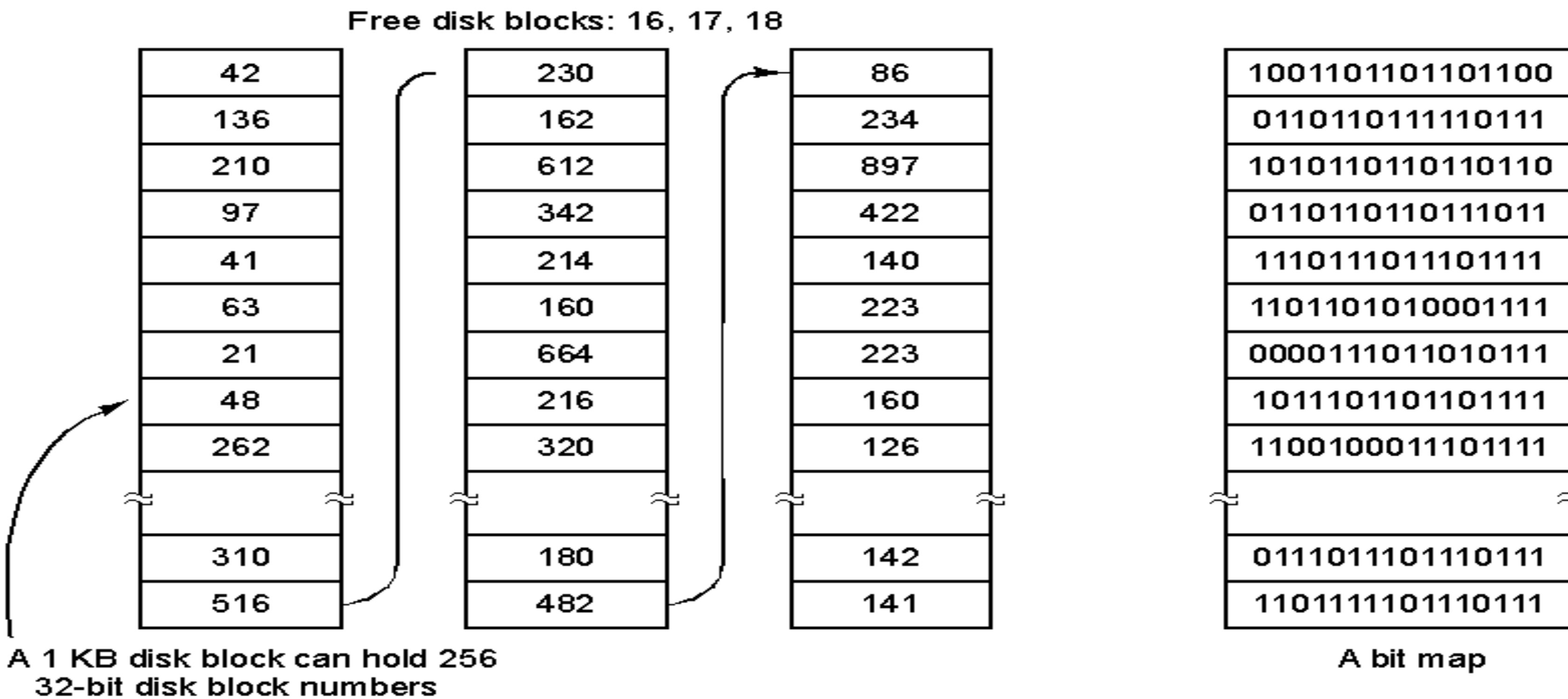
Questions?

- Performance?
- Efficiency?
  - For sequential access? Random access?
  - Large files? Small files? Mixed?
  - HDD VS. SSD?

Managing Free Disk Space

2 approaches to keep track of free disk blocks

- Linked list and bitmap approach

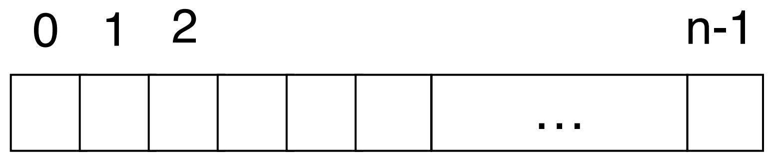


(a) (b)

28

Free-Space Management

- Bit vector(bitmap) (*n* blocks)



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

Block number calculation

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

- Bit map requires extra space
  - Example:
    - block size = 2^{12} bytes
    - disk size = 2^{30} bytes (1 gigabyte)
    - $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files

Linked Free Space List on Disk

◆Linked list (free list)

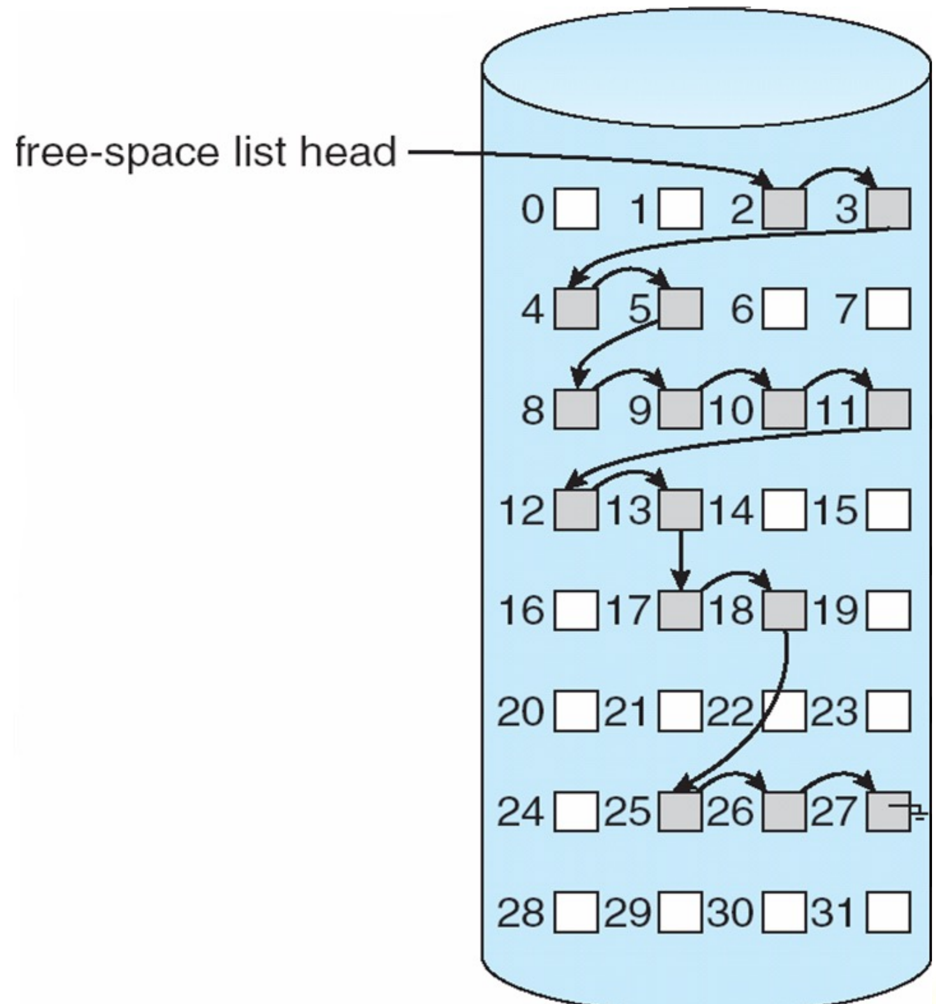
➢Cannot get contiguous space easily

➢No waste of space

空闲链的改进方法：

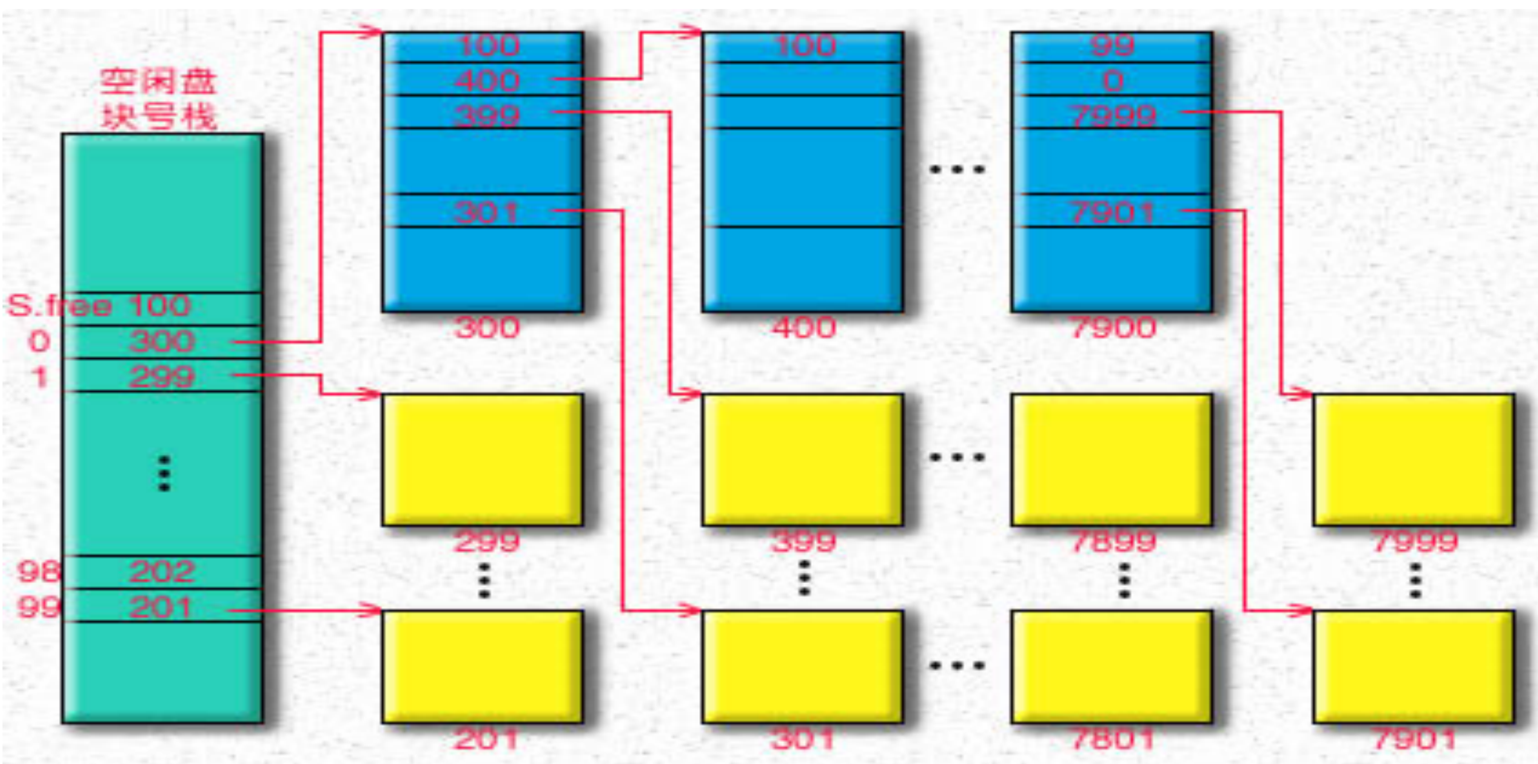
◆Grouping(空闲块按组存放)

◆Counting(利用连续性：只记录第一个块地址和后续连续的空闲快个数)



UNIX的成组链接法

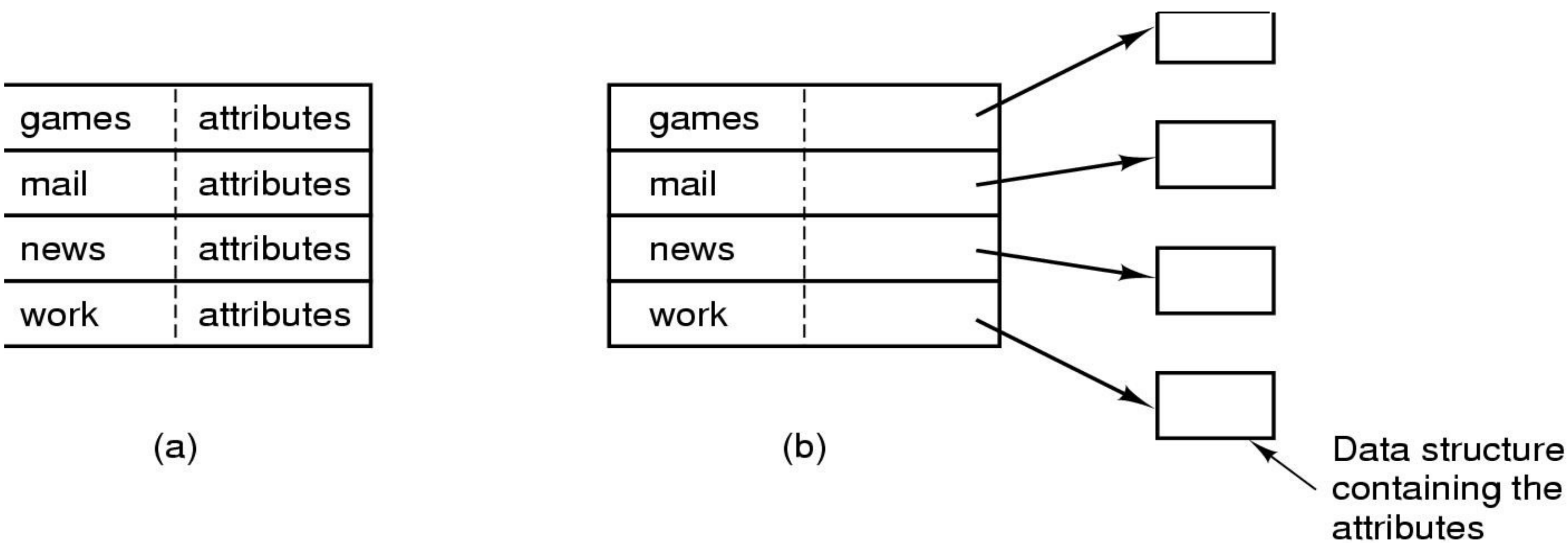
- 空闲盘块号栈：存放当前可用的空闲盘块号及空闲盘块号数N（最多100个）;文件区的所有空闲盘块被分成若干组;
- 第一组盘块总数和盘块号记入空闲盘块号栈中;每一组的第一个盘块中记录有下一组的盘块总数和盘块号



空闲盘块的成组链接法

Directory Implementation

- When a file is opened, OS uses path name to find dir
  - Directory has information about the file's disk blocks
    - Whole file (contiguous), first block (linked-list) or I-node
  - Directory also has attributes of each file
- Directory: map ASCII file name to file attributes & location
- 2 options: entries have all attributes, or point to file I-node



UNIX采用(b),给定文件路径名为 / usr / ast / mbox，检索过程如下：

根目录

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

结点6是／usr的目录

132

132#块是/usr的目录

6	.
1	..
19	disk
30	erik
51	jim
26	ast
45	bat

结点26是/usr/ast目录

406

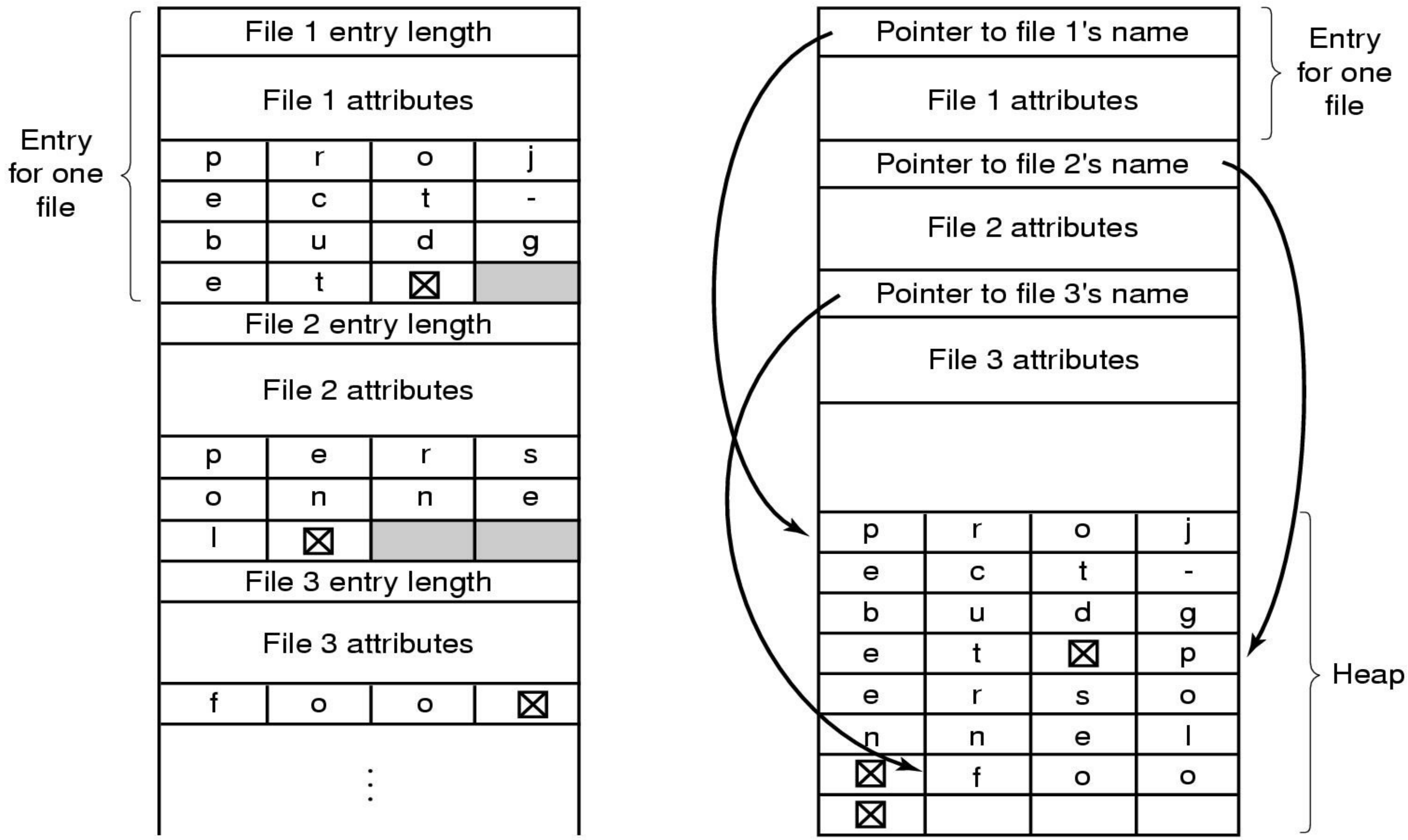
406#块是/usr/ast的目录

26	.
6	..
64	grants
92	books
60	mbox
81	minix
17	src

Directory Implementation

- Linear list of file names with pointer to the data blocks
  - simple to program
  - time-consuming to execute
- Hash Table – linear list with hash data structure
  - decreases directory search time
  - collisions – situations where two file names hash to the same location
  - fixed size

Managing file names: Example



(a) (b)

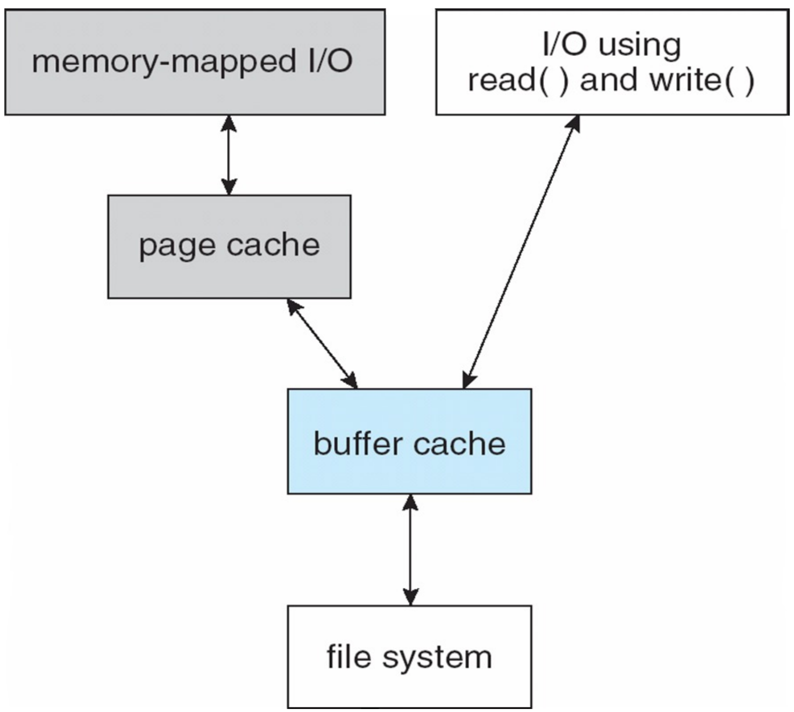
Efficiency and Performance

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

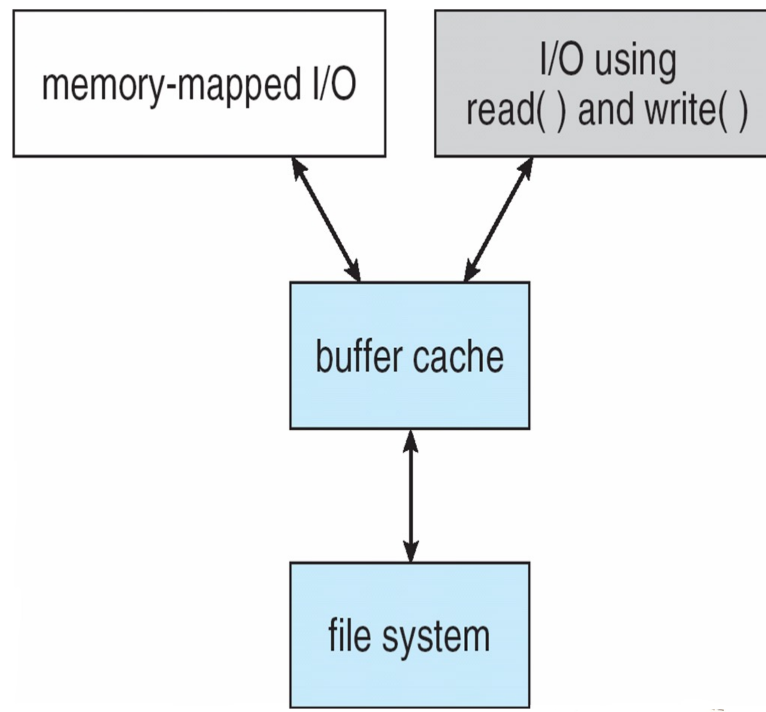
Page Cache

- A page cache caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure:

I/O Without a Unified Buffer Cache



I/O Using a Unified Buffer Cache



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 (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

Log Structured File Systems

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

```
BEGIN_TRANSACTION;
    x = x + 1;
    y = y + 2
    x = y * y;
END_TRANSACTION;
```

Log

[x = 0 / 1]

[y = 0/2]

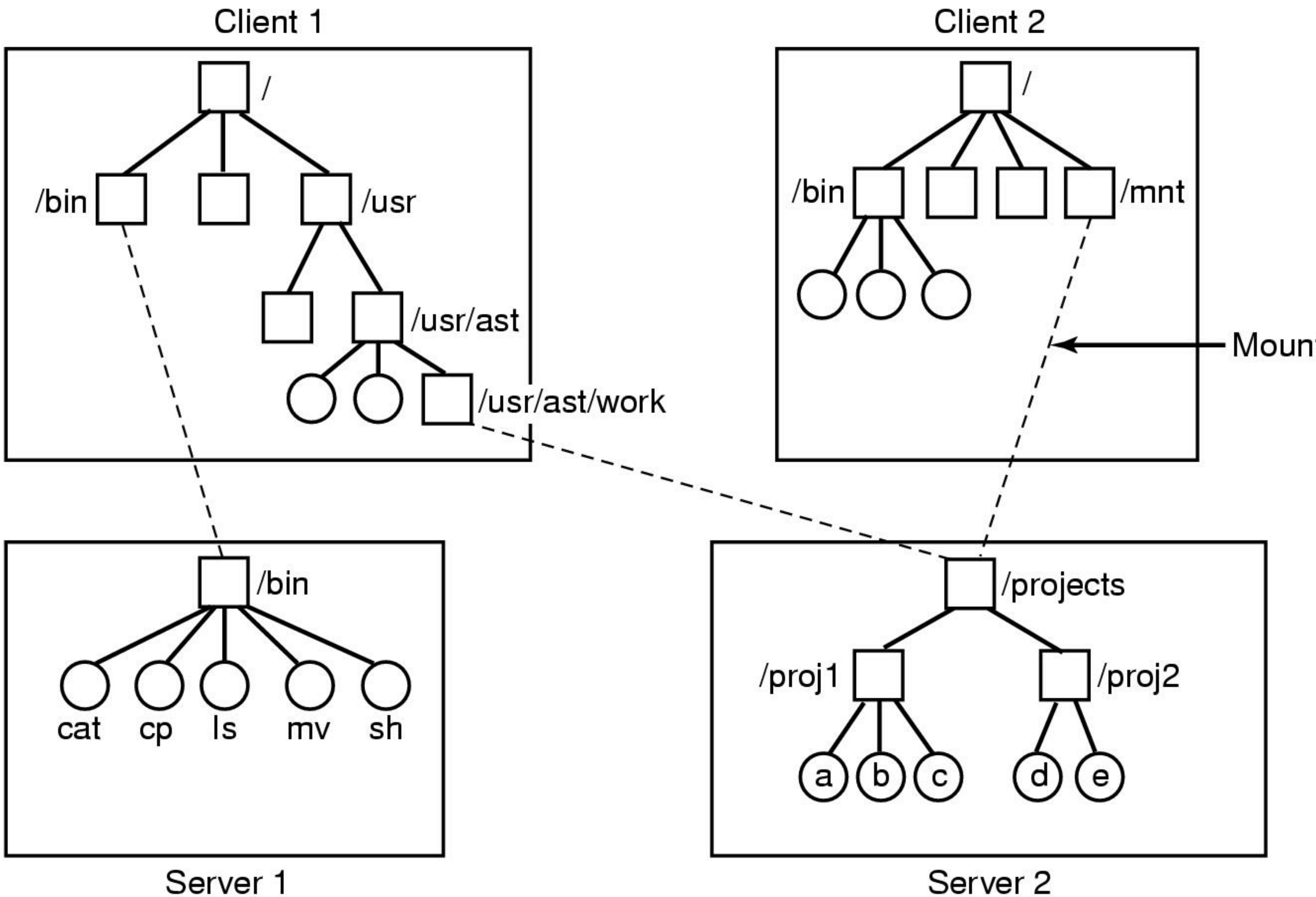
[x = 1/4]

写完后系统崩溃

Network File System (NFS)

- Developed by Sun Microsystems in 1984
  - Used to join FSes on multiple computers as one logical whole
- Used commonly today with UNIX systems
- Assumptions
  - Allows arbitrary collection of users to share a file system
  - Clients and servers might be on different LANs
  - Machines can be clients and servers at the same time
- Architecture:
  - A server exports one or more of its directories to remote clients
  - Clients access exported directories by mounting them
    - The contents are then accessed as if they were local

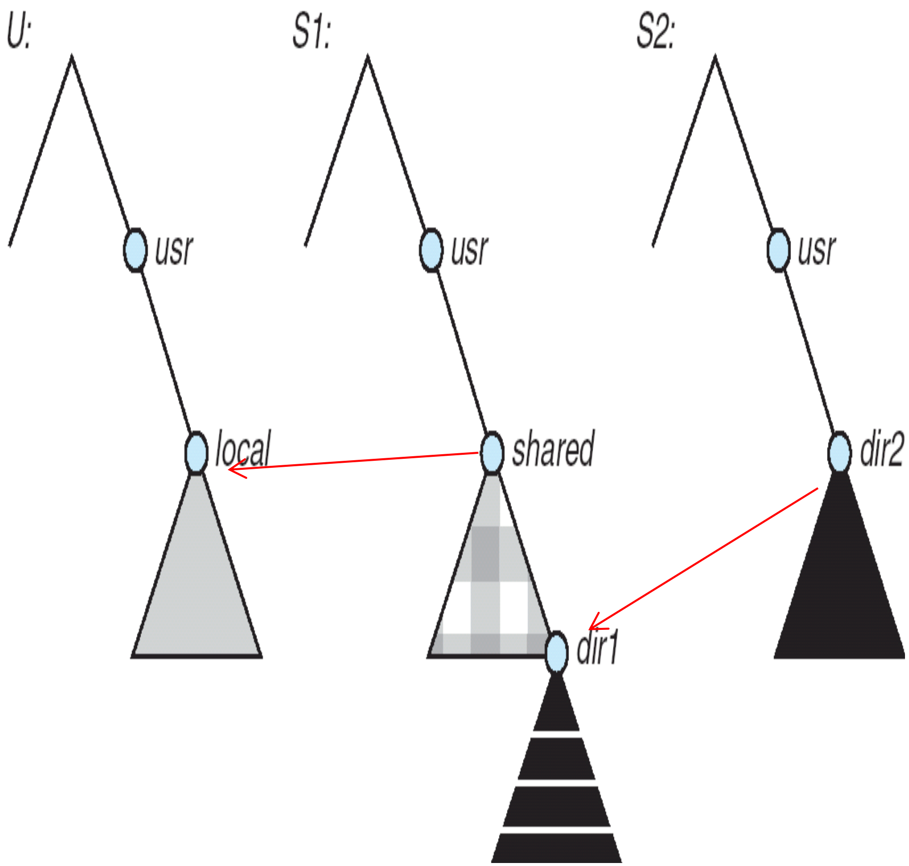
Example



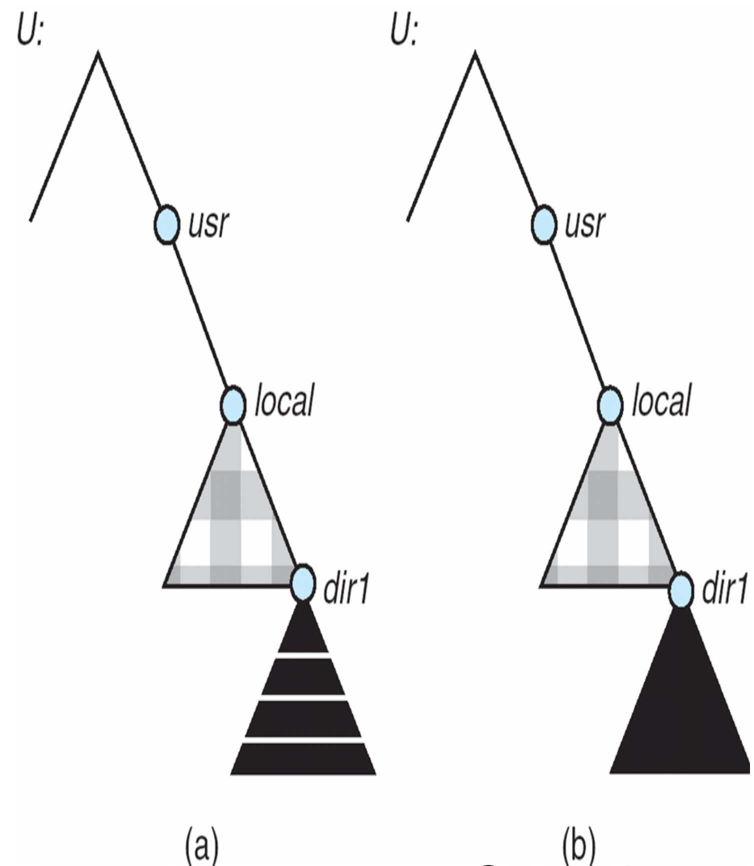
Client 1
/
/bin
/usr
/usr/ast
/usr/ast/work
Client 2
/
/bin
/mnt
Mount
/bin
cat
cp
ls
mv
sh
Server 1
/projects
/proj1
/proj2
a
b
c
d
e
Server 2

41

Three Independent File Systems



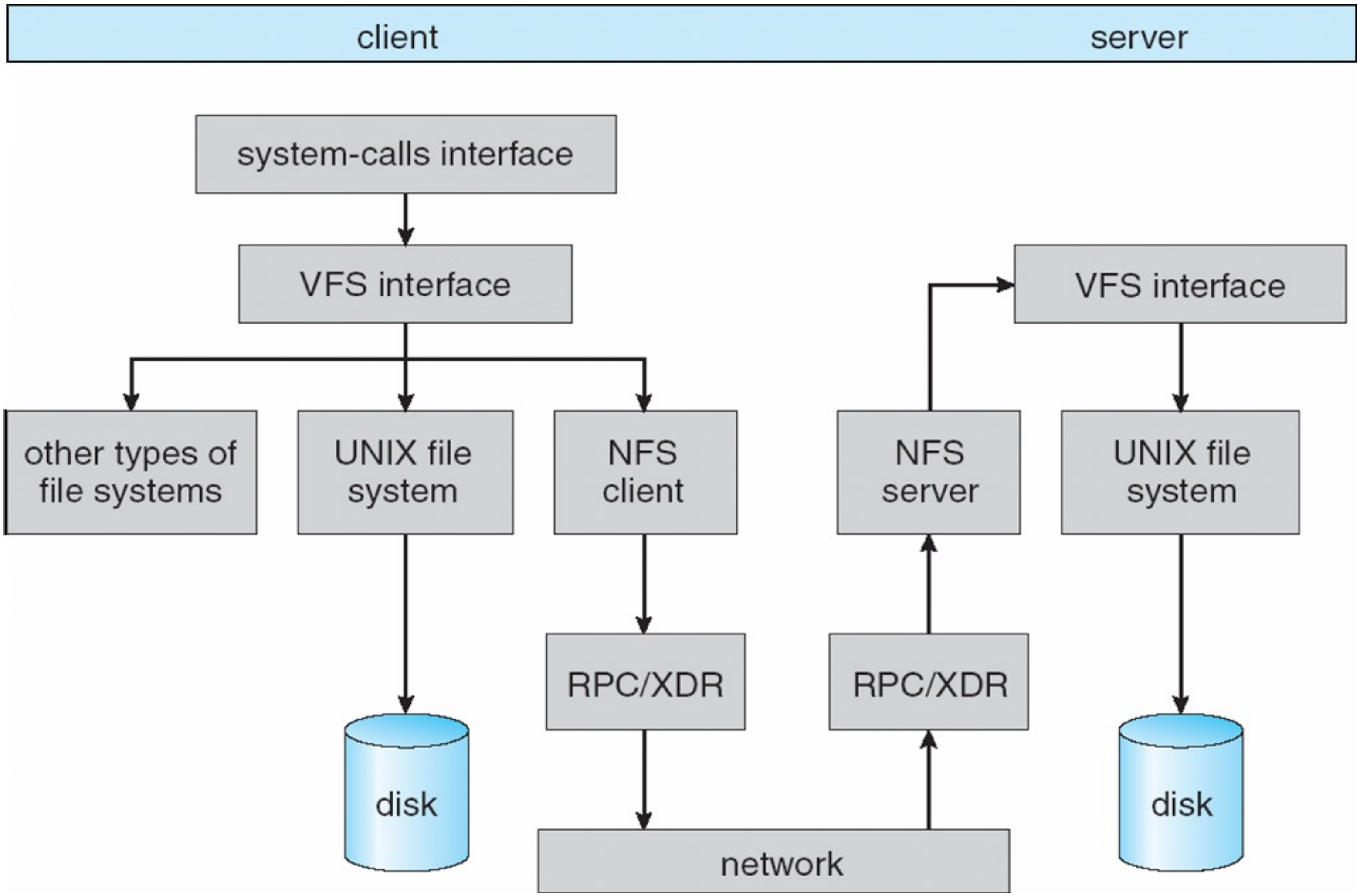
Mounting in NFS



Mounts

Cascading mounts

Schematic View of NFS Architecture



assignment

- 11.3
- 11.7
- 11.8

End of Chapter 11