PART 4. STORAGE MANAGEMENT

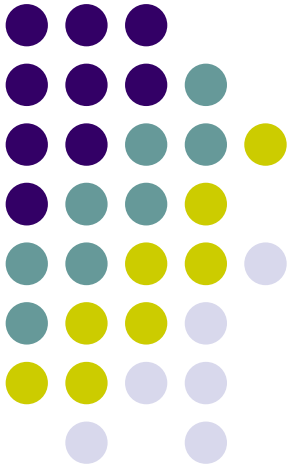
Chapter 11: Implementing File Systems

WHAT'S AHEAD:

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

WE AIM:

- 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

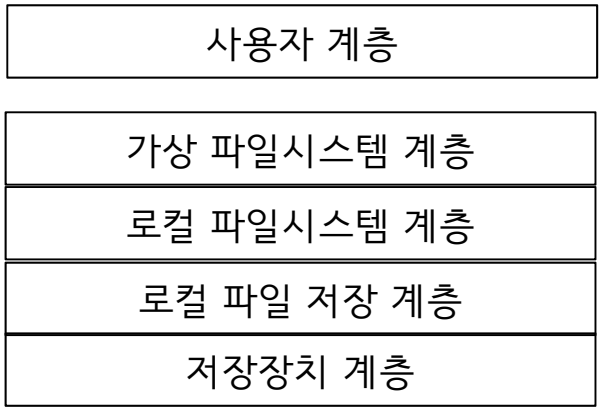


Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심•요점 이슈와 해결방안

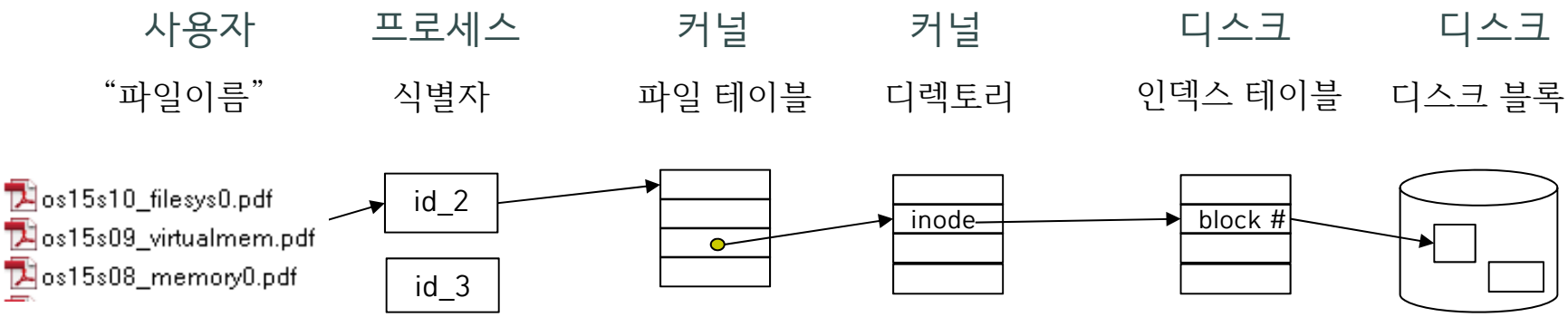
이슈: 어떻게 프로세스가 정확하고 안전하고 효율적으로 디스크 파일을 이용하게 할 것인가

해결방안: 계층화된 구조, “링킹 테이블” 기반의 관리



사용자 레벨

커널 레벨

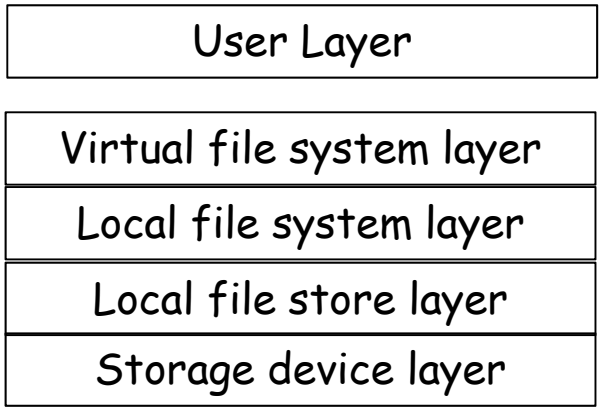


Core Ideas

Issues and Approaches

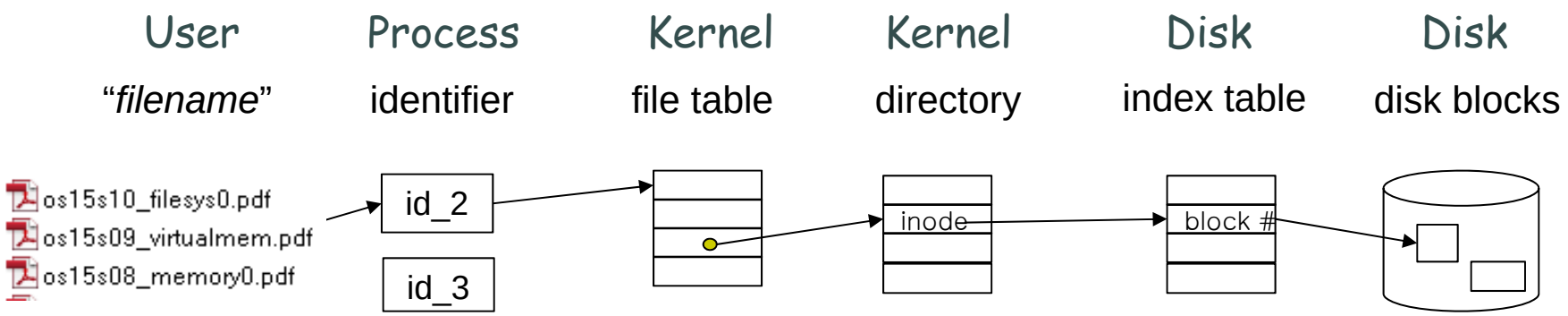
Issues: How to let a process utilize on-disk files *correctly, safely and efficiently*

Approaches: *Layered* structure, "*Linking-tables*"-based management



User level

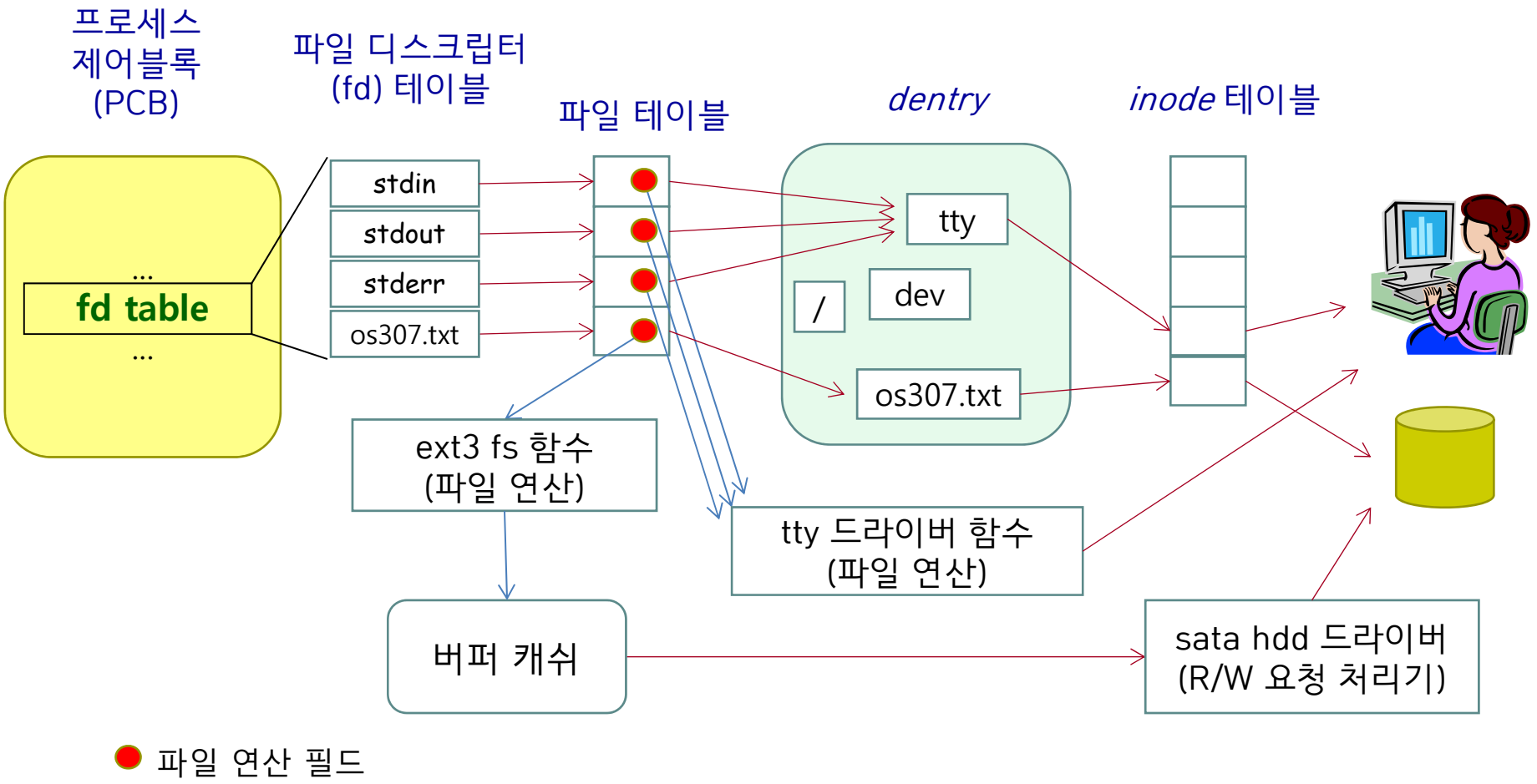
Kernel level



핵심•요점

파일시스템의 동적 구조 및 운용

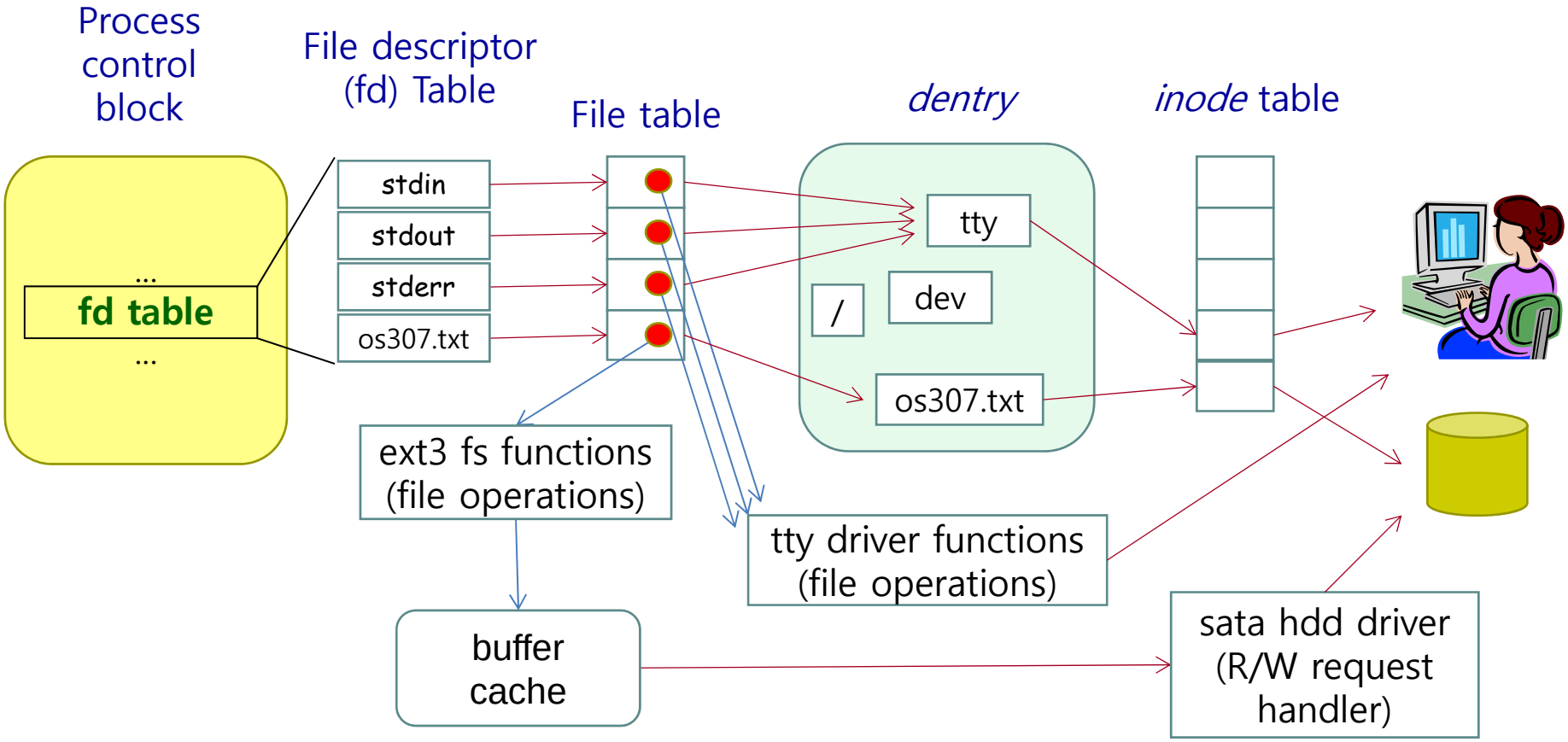
리눅스 사례



Core Ideas

The Way File System Works

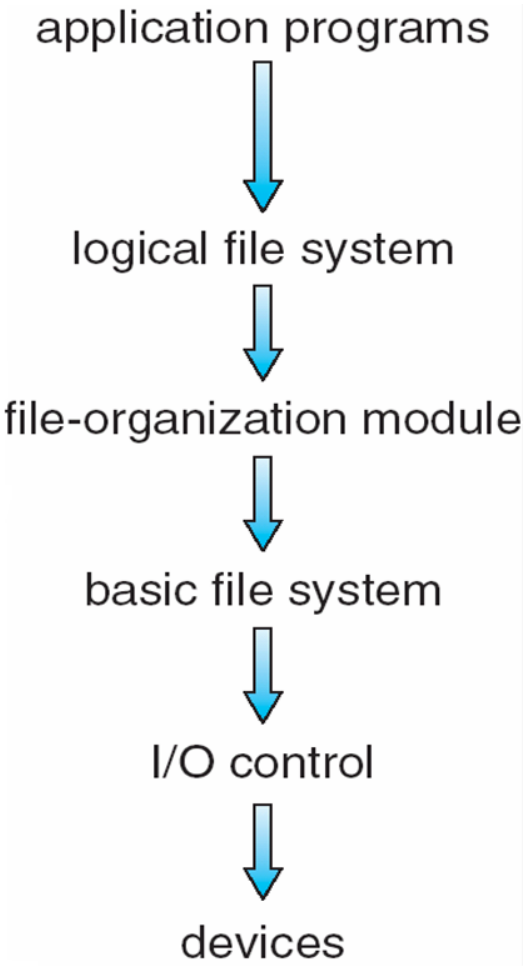
The Linux Example



File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **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, and retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **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



File System Layers

- **Device drivers** 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
- **Basic file system** given command like "retrieve block 123" translates to device driver
  - Also manages memory buffers and caches (allocation, freeing, replacement)
    - Buffers hold data in transit
    - Caches hold frequently used data
- **File organization module** understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation

File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in Unix)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
  - Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an OS
  - 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; plus distributed file systems, etc)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

- For implementation of related system calls at the API level
  - On-disk and in-memory structures

On-Disk Structure

- **Boot control block** contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
  - Boot sector [UFS], partition boot sector [NTFS]
- **Volume control block** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
  - Called superblock [UFS], stored in master file table [NTFS]
- Directory structure (per file system) organizes the files
  - File names and associated inode numbers [UFS], stored in master file table [NTFS]
- Per-file **File Control Block (FCB)** contains many details about the file
  - Inode number, permissions, size, dates [UFS]
  - Stored in master file table [NTFS]

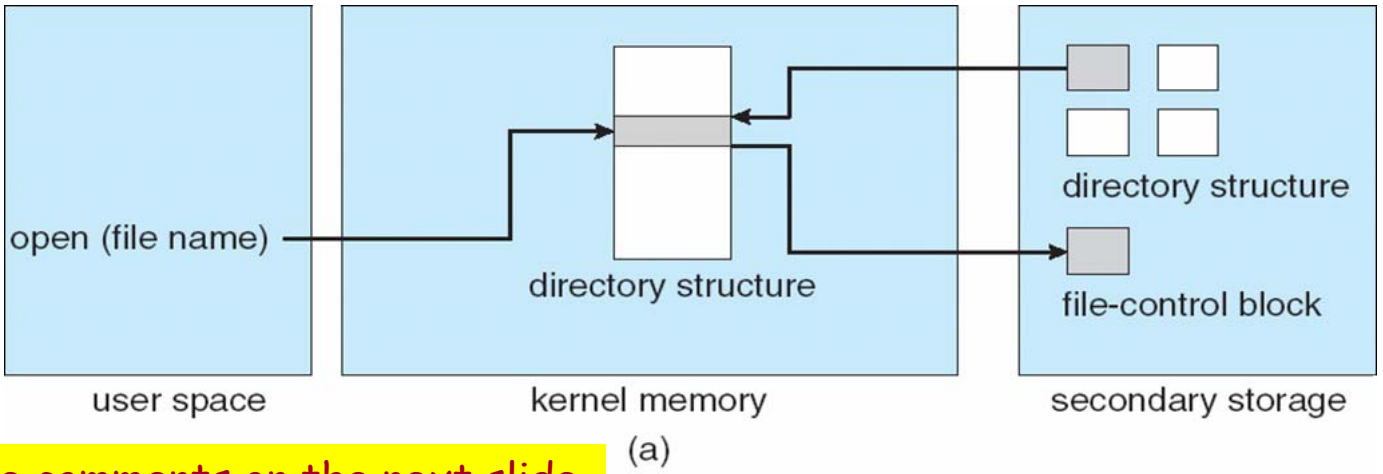
A Typical File Control Block

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

- Mount table storing file system mounts, mount points, file system types
- The following figure illustrates the necessary file system structures provided by the operating systems
  - Figure 11-3(a) refers to opening a file
  - Figure 11-3(b) refers to reading a file
- ***Open*** returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address
- Related data structures
  - In-memory mount table: info about each mounted volume
  - In-memory directory-structure cache
  - System-wide open-file table: contains FCBs
  - Per-process open-file table
  - Buffers holding data blocks from secondary storage

In-Memory File System Structures



Refer to the comments on the next slide

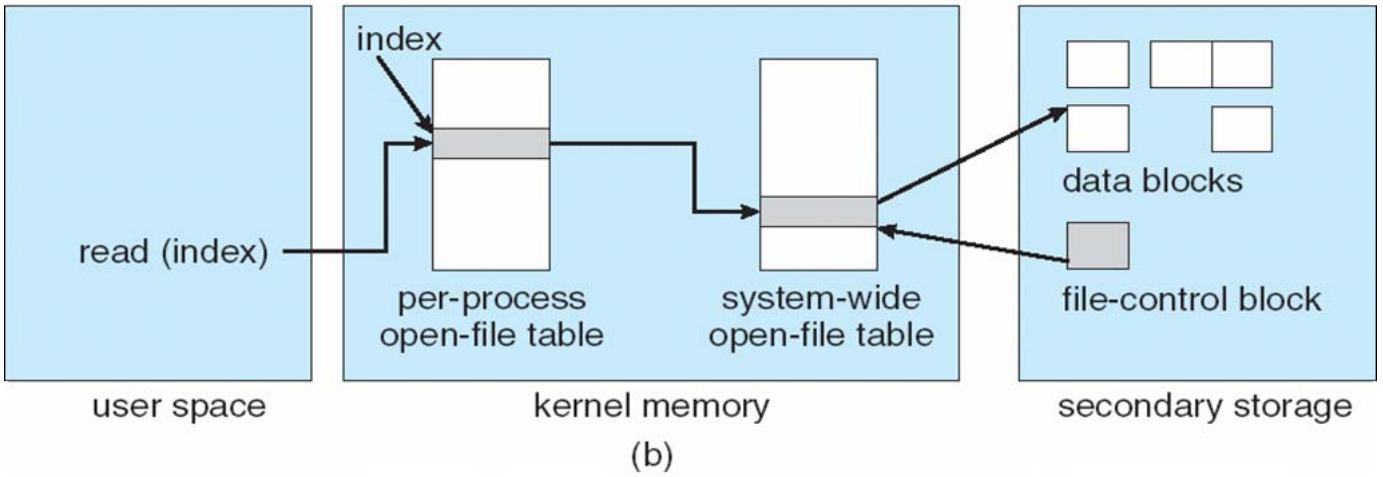


Figure 11.3 In-memory file-system structures. (a) File open. (b) File read.

In-Memory Structures – *File Open*

- We now illustrate how **open( )** works

1) Search the system-wide open-file table to see if the file is already in use by another process.
2) (a) If so, {@} a per-process open-file table entry is created, pointing to the existing system-wide open-file table. (See Fig. 11.3(b))
2) (b) If not open (Fig. 11.3(a)), FCB is brought into system-wide open-file table. Then, @.
   - Per-process open-file table entries include the pointer and file position.
3) Return a pointer (file descriptor) to the per-process open-file table entry.

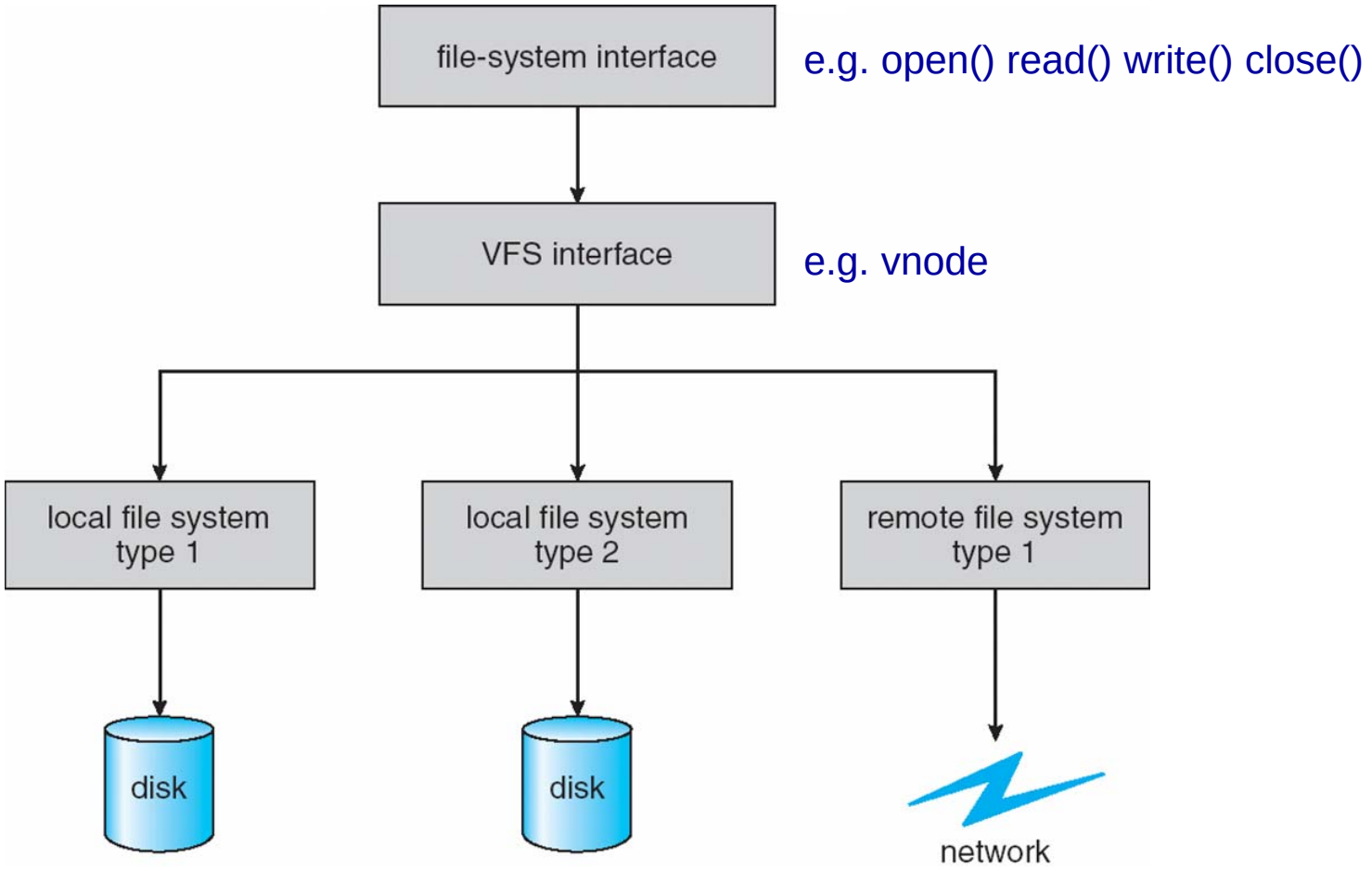
Partitions and Mounting

- Partition can be a volume containing a file system ("cooked") or **raw** – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- **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
- At mount time, file system consistency checked
  - Is all metadata (directory format) 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
- The API is to the VFS interface, rather than any specific type of file system

Schematic View of Virtual File System



Virtual File System Implementation

- For example, Linux has four object types:
  - inode: represents an individual file
  - file: represents an open file
  - superblock: represents an entire file system
  - dentry: represents an individual directory entry
- 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
  - E.g., operations for the file object: `int open(), int close(), ssize_t read(), ssize_t write(), int mmap()`
- VFS need not know in advance exactly what kind of object it is dealing with

Directory Implementation

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - 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

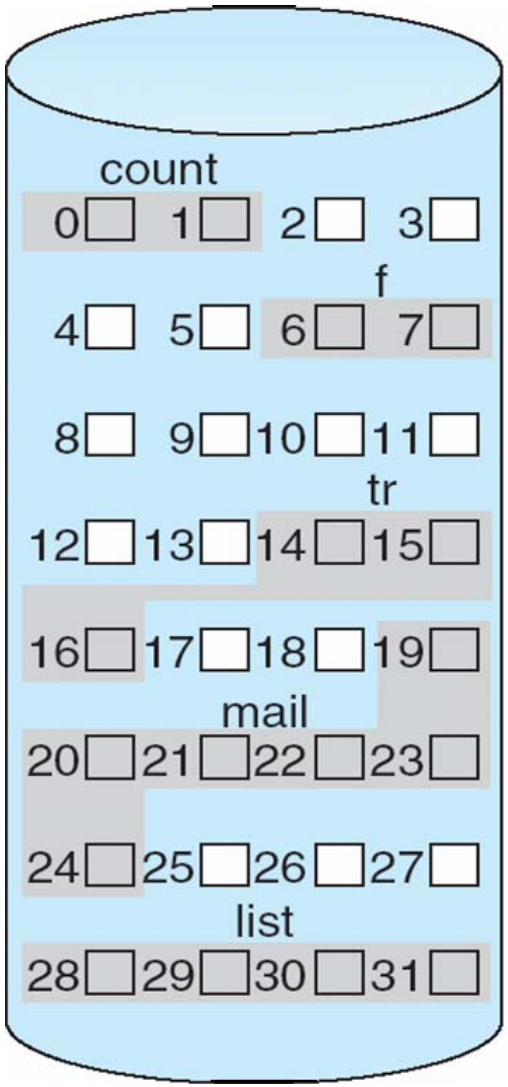
Allocation Methods – Contiguous Allocation

- 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**
- Mapping from logical to physical

  `LA ÷ block_size ➔ Q + R`

  - where LA = logical address, Q = quotient, R = remainder
  - number of blocks needed: Q+1
- Physical address
  - Block to be accessed = Q + starting address
  - Displacement into block = R

Contiguous Allocation of Disk Space



directory

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

Modified: 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
  - Location of a file's block: {location, block count, link to the first block of the next extent}

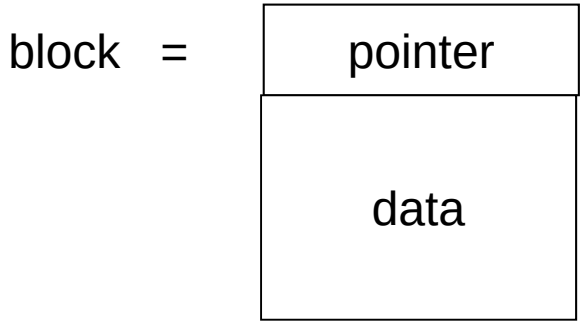
Allocation Methods – Linked Allocation

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

- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

Linked Allocation (Cont.)

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
  - A block consists of a pointer to the next block, and data

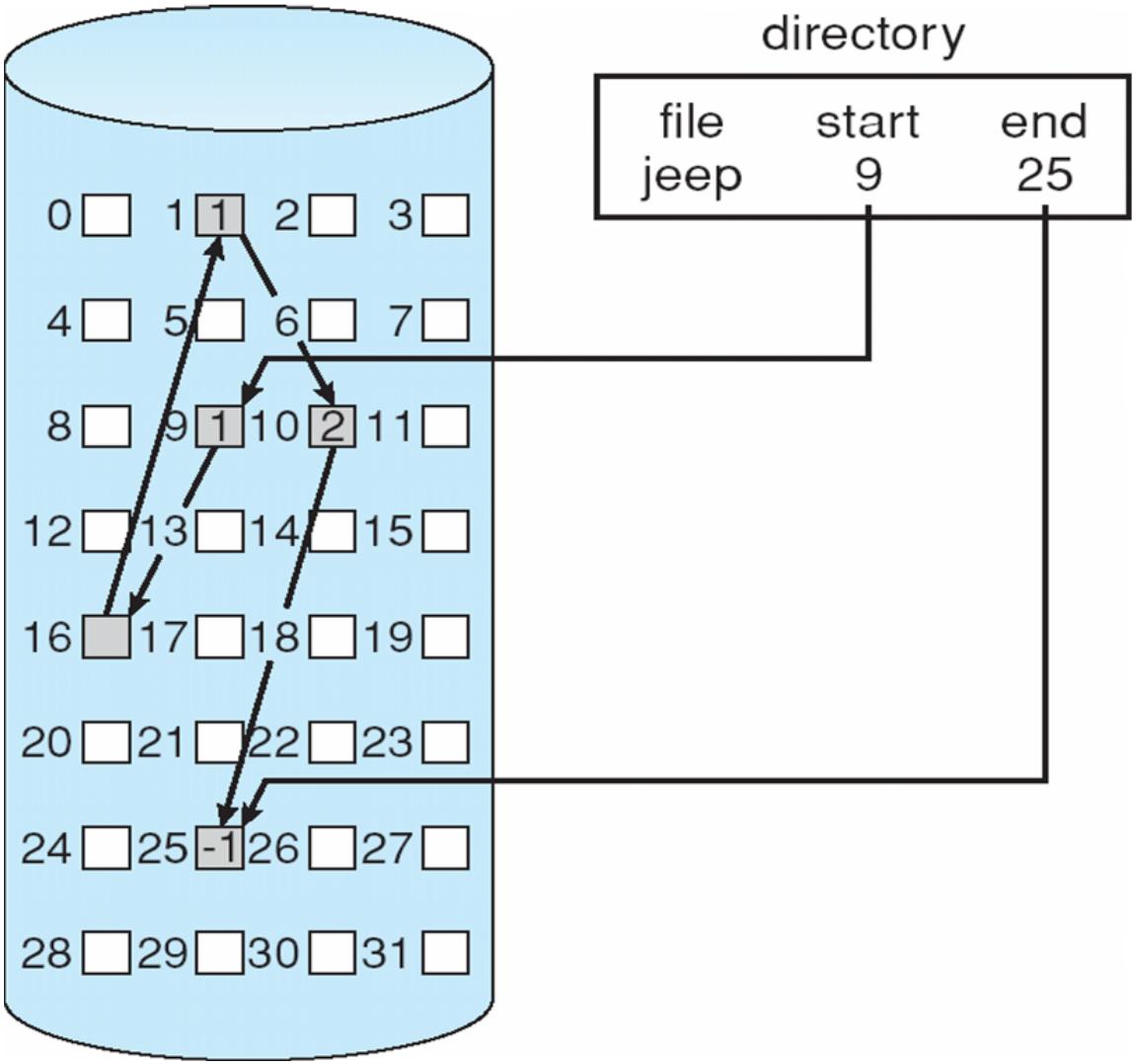


- Mapping

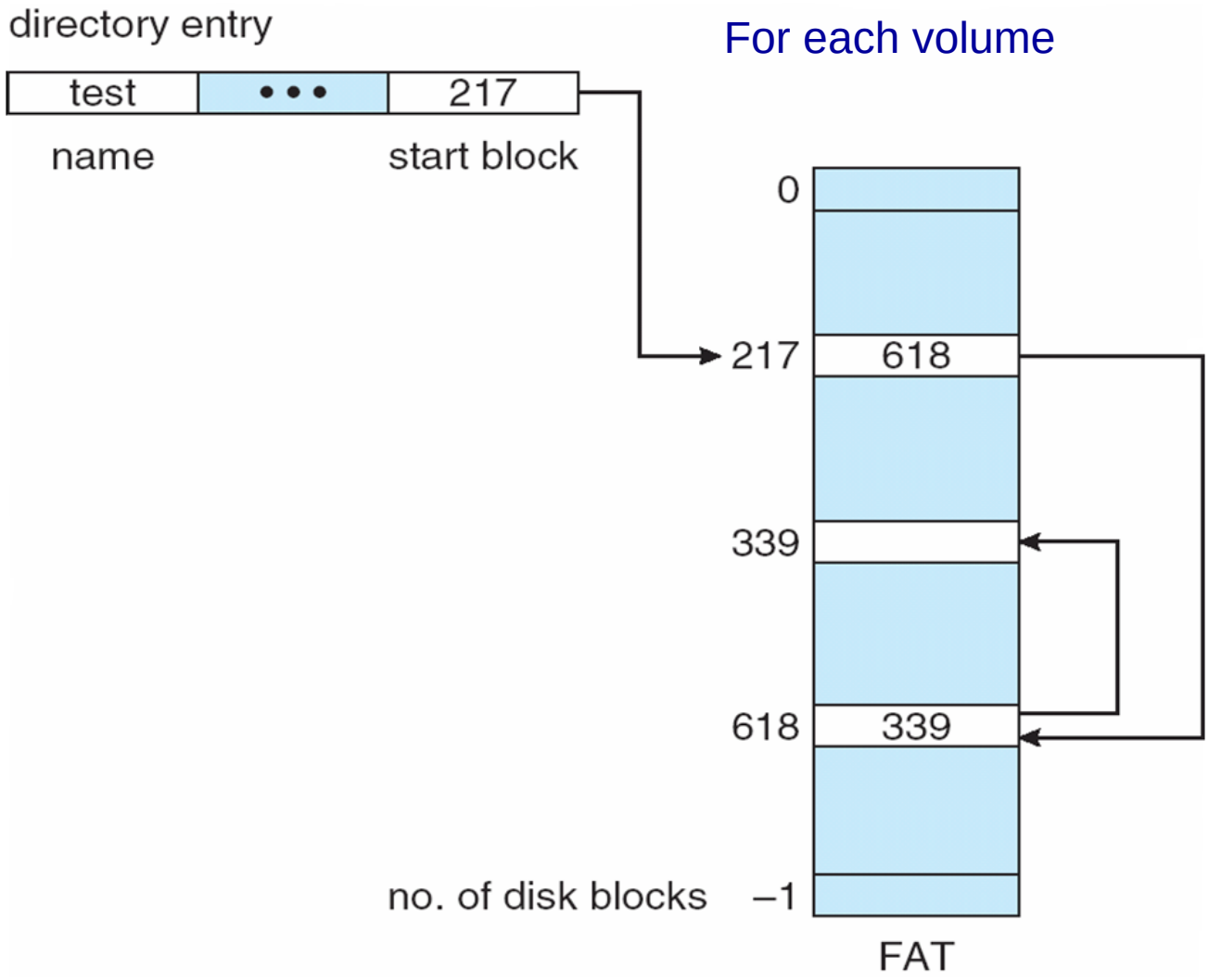
  `LA ÷ (block_size – pointer_size)` ➔ `Q + R`

- Physical address
  - Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file.
  - Displacement into block = R + pointer_size(in byte)

Linked Allocation (Cont.)



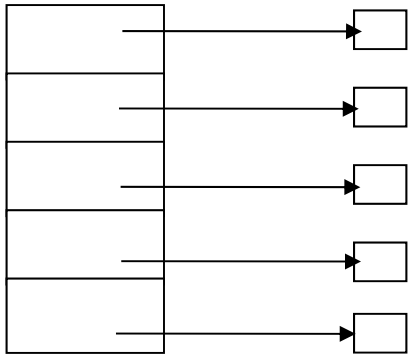
File-Allocation Table



Allocation Methods – Indexed Allocation

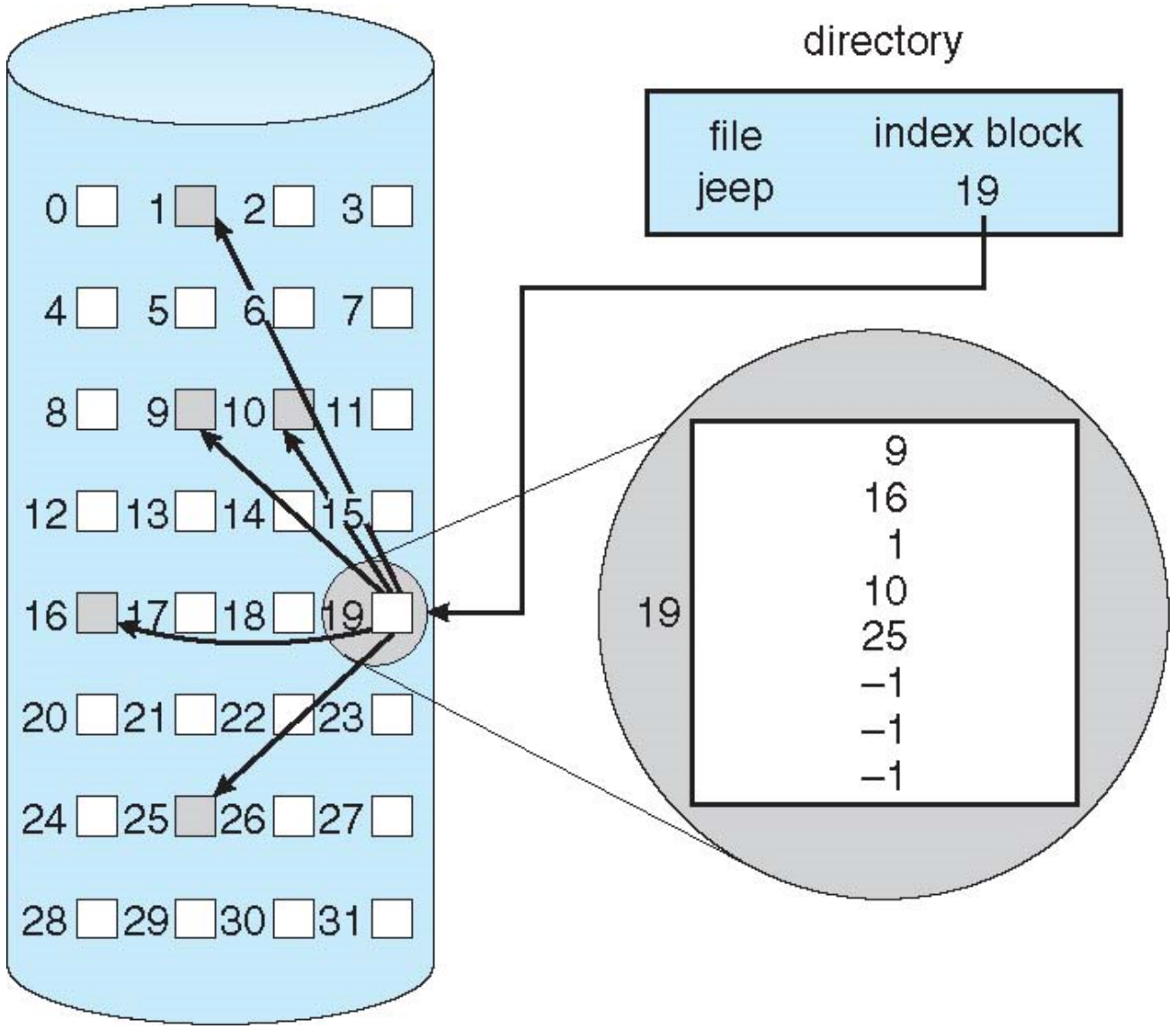
- **Indexed allocation**
  - Each file has its own **index block**(s) of pointers to its data blocks

- Logical view



index table

Example of Indexed Allocation



Indexed Allocation (Cont.)

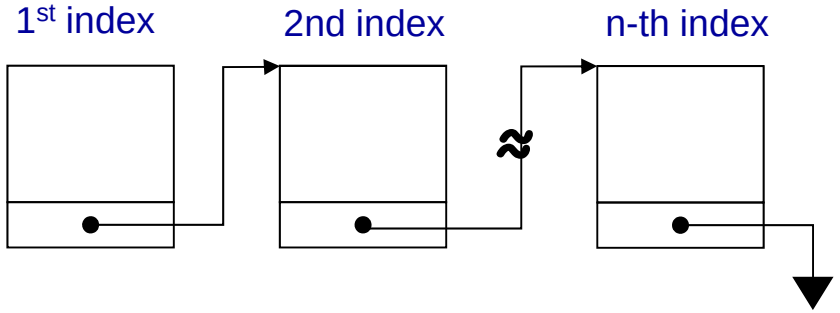
- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 64K (2^{16}) bytes and block size of 256 (2^{8})bytes. We need only 1 block for index table

  `LA ÷ block_size ➔ Q + R`

  - where `Q` = displacement into index table, and `R` = displacement into block (assume one-byte long pointer)

Indexed Allocation – Scalable Index Tables

- The size of index table
  - tradeoff of spatial overhead vs. scalability
  - i.e., if too large, high memory overhead for smaller files, and if too small, unable to accommodate larger files
- Approaches to scalable index tables
  - Linked scheme
  - Multilevel index
  - Combined scheme



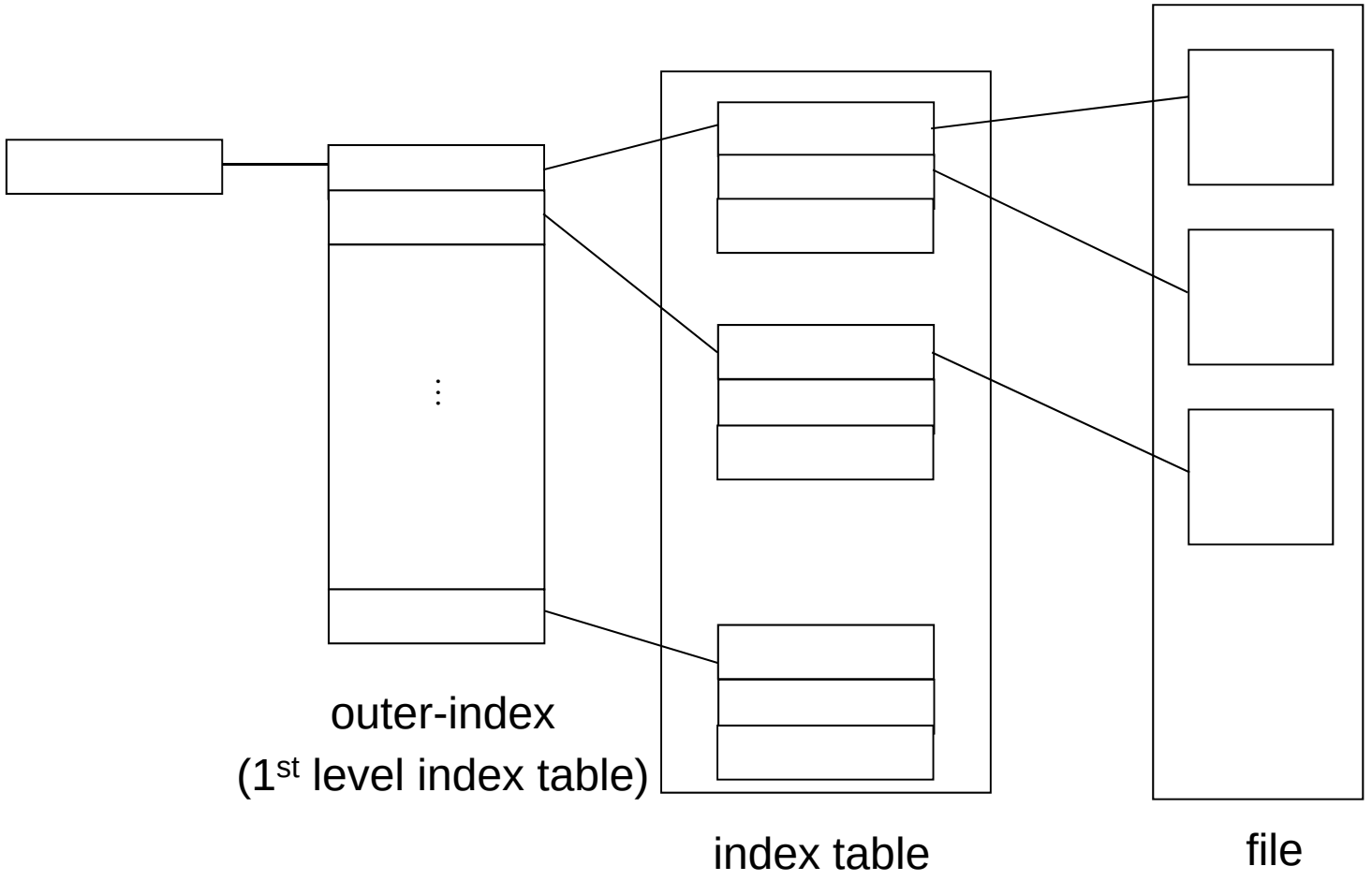
- Linked scheme – Link blocks of index table
  - for example, in an index table, an entry is allocated for a pointer to the next index table, whereas other entries store a pointer to data block

Indexed Allocation – Scalable Index Tables (Cont.)

- Multi-level index
  - A first-level index block to point to a set of second-level index blocks, and so on. The last index block is used to point to data blocks.
  - For instance, in case of two-level index
    - 4K(2^{12})-byte blocks could store 1,024(1K) four-byte pointers in the first index block.
    - The second index has 1K pointer entries → 1K x 1K = 1M pointers to data blocks → 1M x 4K = 4G (max. file size)
    - LA ÷ (block_size/pointer_size)^2 ➔ Q_1 + R_1
    - where Q_1 = displacement into outer index table, and R_1 ÷ (block_size/pointer_size) ➔ Q_2 + R_2 Q_2 = displacement into block of index table, R_2 = displacement into block of file

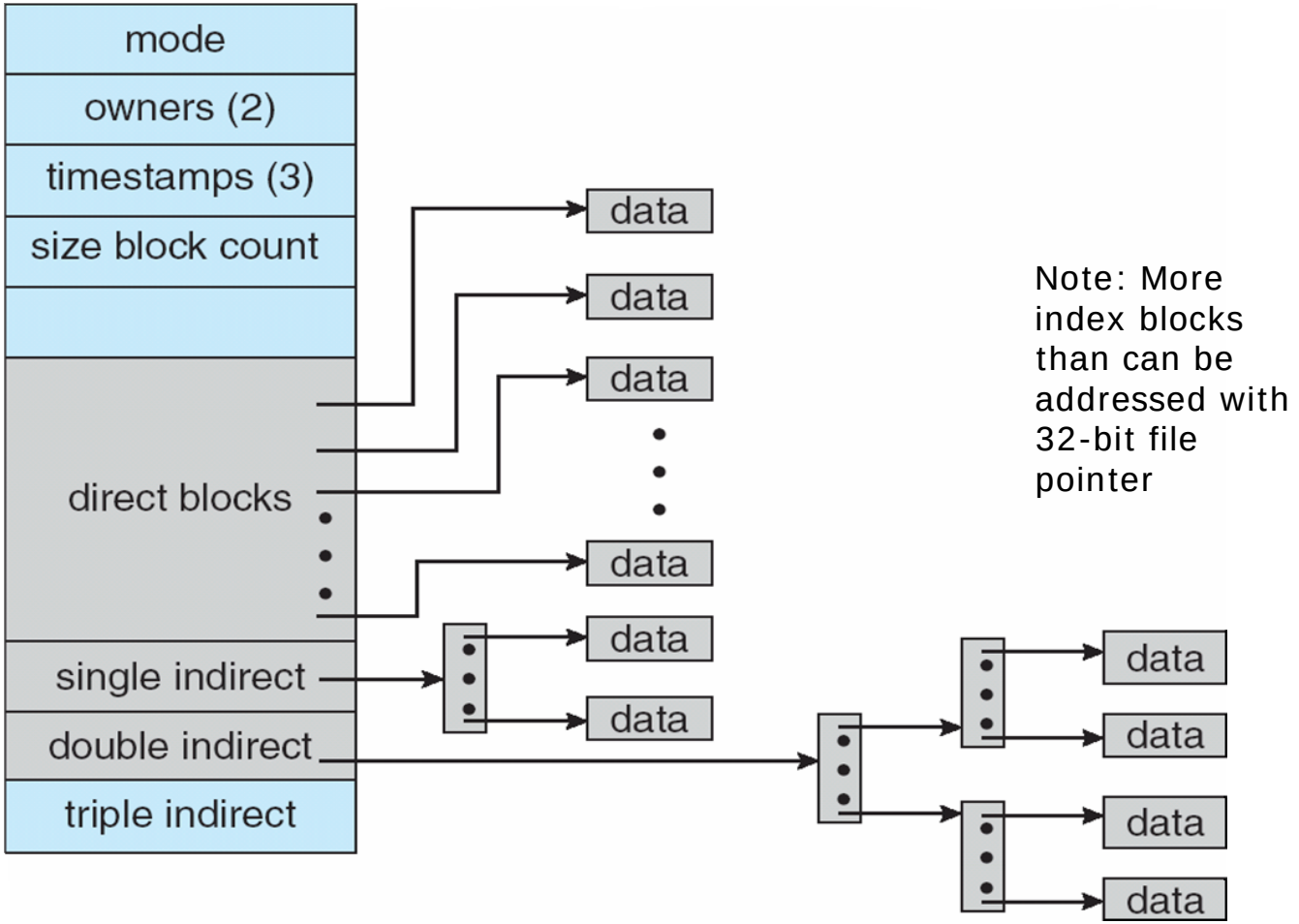
Indexed Allocation – Scalable Index Tables (Cont.)

- Illustration of two-level index



Indexed Allocation – Scalable Index Tables (Cont.)

- Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)



Performance

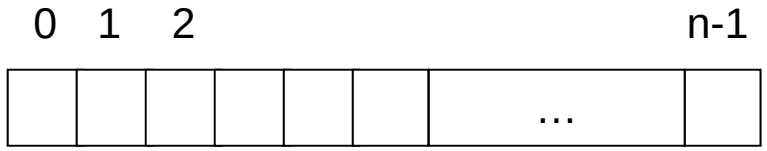
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Performance (Cont.)

- Adding instructions to the execution path to save one disk I/O is reasonable
  - Intel Core i7 Extreme Edition 990x (2011) at 3.46Ghz = 159,000 MIPS
    - http://en.wikipedia.org/wiki/Instructions_per_second
  - Typical disk drive at 250 I/Os per second
    - 159,000 MIPS / 250 = 630 million instructions during one disk I/O
  - Fast SSD drives provide 60,000 IOPS
    - 159,000 MIPS / 60,000 = 2.65 millions instructions during one disk I/O

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (*n* blocks)



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

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

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

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

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

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

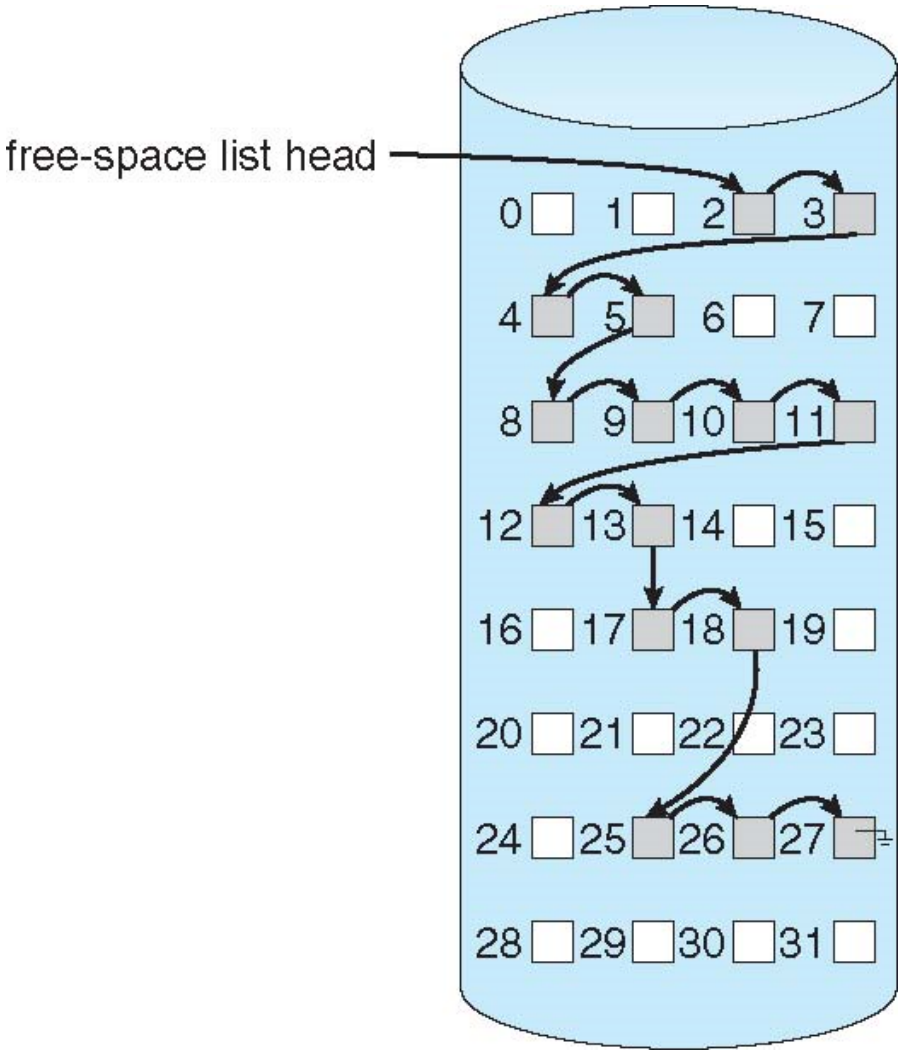
    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 256 MB)

    if clusters of 4 blocks -> 64MB of memory

- Easy to get contiguous files

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded)

Linked Free Space List on Disk



Free-Space Management (Cont.)

- 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)

- Counting
  - Because space is frequently contiguously used and freed, with contiguous allocation, extents, or clustering
    - Keep address of first free block and count of following free blocks
    - Free space list then has entries containing addresses and counts

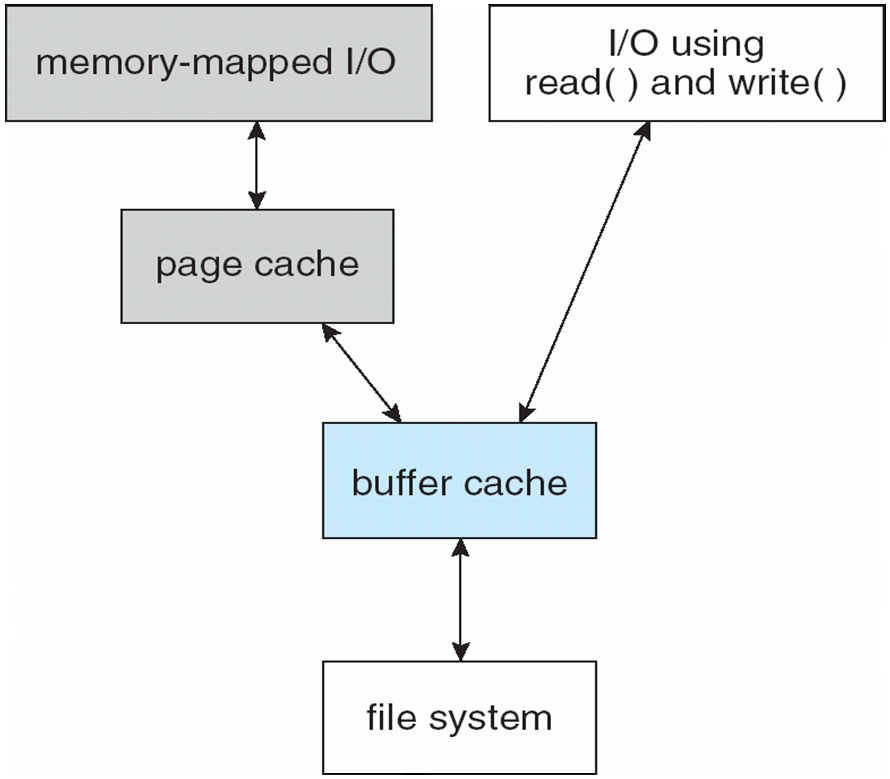
Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
- Performance
  - Keeping data and metadata close together
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes

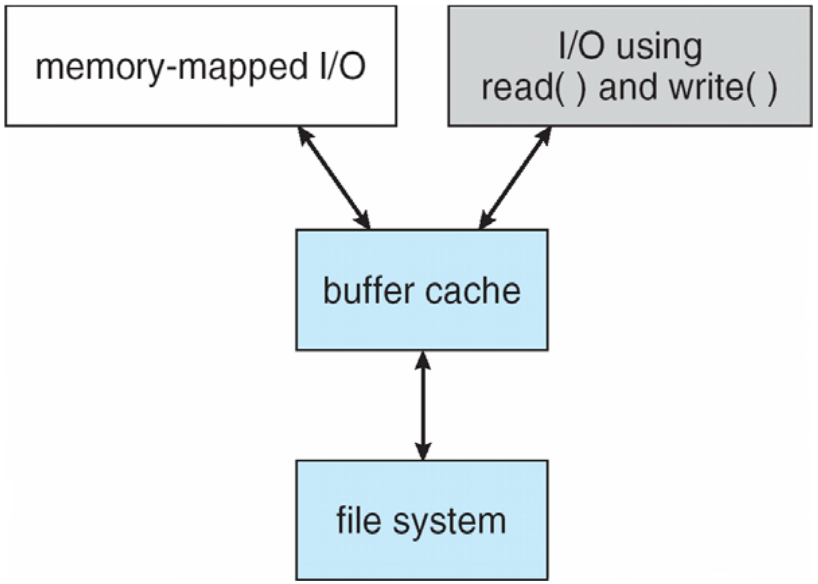
Page Cache vs. Unified Buffer Cache

- A **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 (disk) cache
  - This leads to the following figure
- A **unified buffer** cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid **double caching**
  - But which caches get priority, and what replacement algorithms to use?

I/O and a Unified Buffer Cache



I/O without a unified buffer cache



I/O with a unified buffer cache

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- 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 metadata 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 (sequential recording)
  - 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 structures
  - When the file system structures are 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

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)

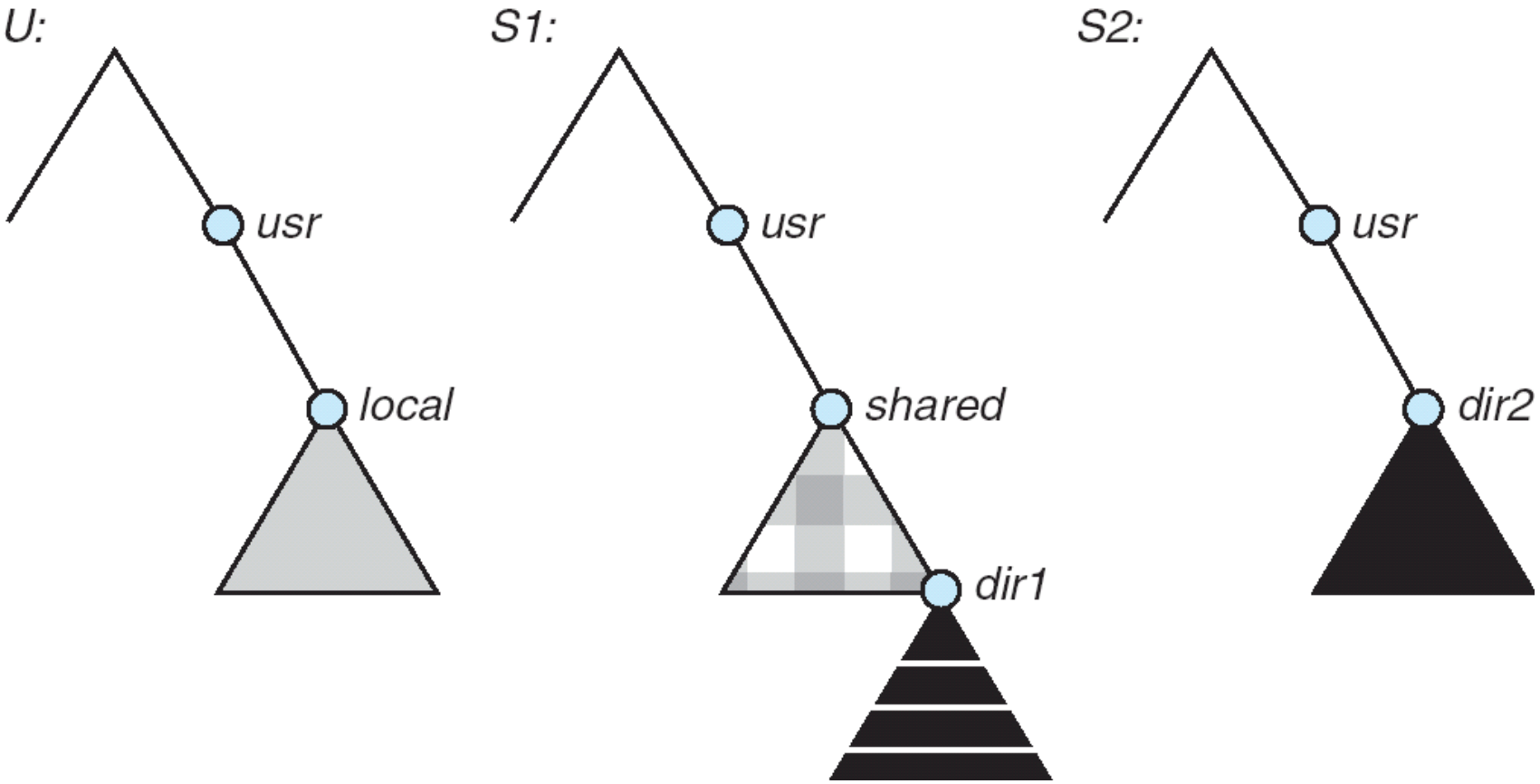
NFS (Cont.)

- Interconnected workstations viewed as a set of independent machines with independent file systems
  - Allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

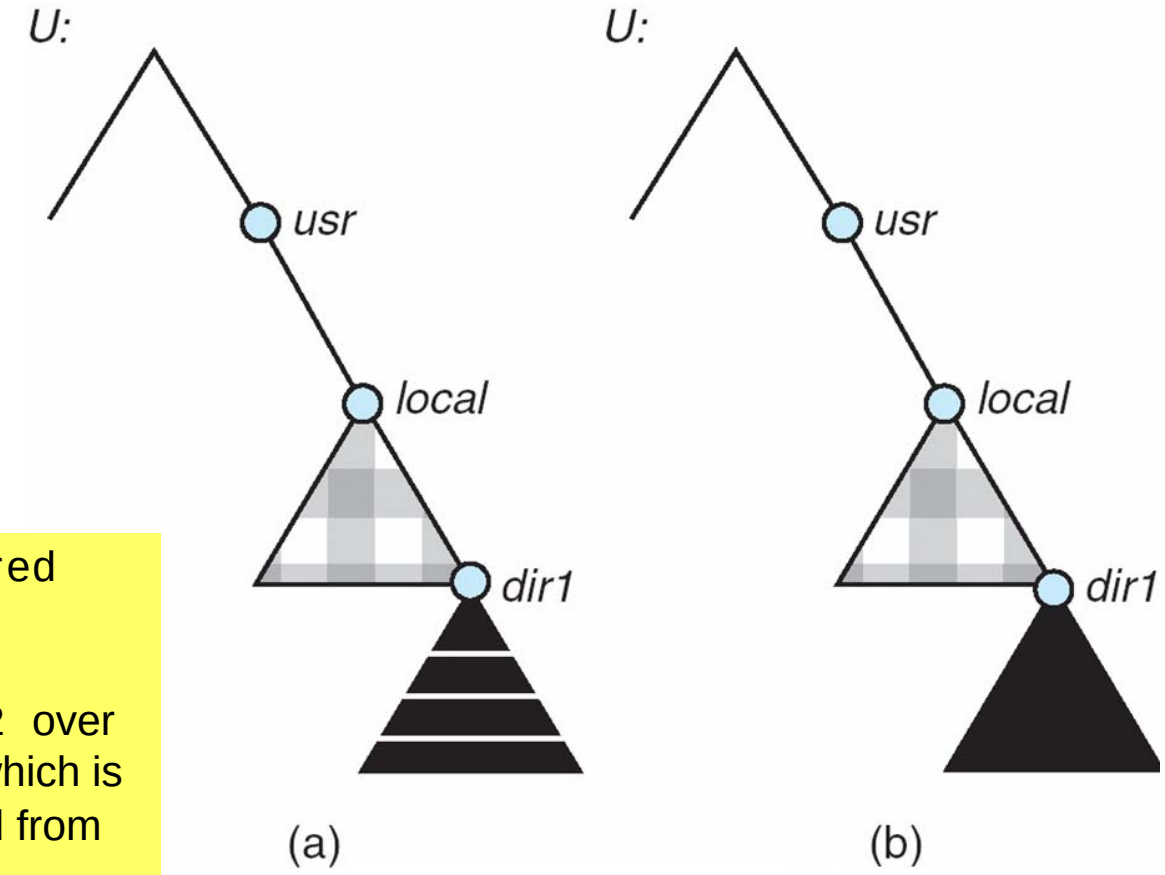
NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures
  - the NFS specifications independent of these media
- This independence is achieved through the use of RPC primitives
  - RPC is built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces
- The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote-file-access services
  - The NFS specification: mount protocol + NFS protocol

Three Independent File Systems



Mounting in NFS



(a) Mounting `S1:/usr/shared` over `U:/usr/local`

(b) Mounting `S2:/usr/dir2` over `U:/usr/local/dir1`, which is already remotely mounted from S1

Mounts

Cascading mounts

NFS Mount Protocol

- Establishes initial logical connection between server and client
- Mount operation includes name of remote directory to be mounted and name of server machine storing it
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine
  - Export list – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them
- Following a mount request that conforms to its export list, the server returns a file handle—a key for further accesses
- File handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system
- The mount operation changes only the user's view and does not affect the server side

NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  - searching for a file within a directory
  - reading a set of directory entries
  - manipulating links and directories
  - accessing file attributes
  - reading and writing files
- NFS servers are **stateless**; each request has to provide a full set of arguments (NFS V4 is just coming available – very different, stateful)
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching)
- The NFS protocol does not provide concurrency-control mechanisms

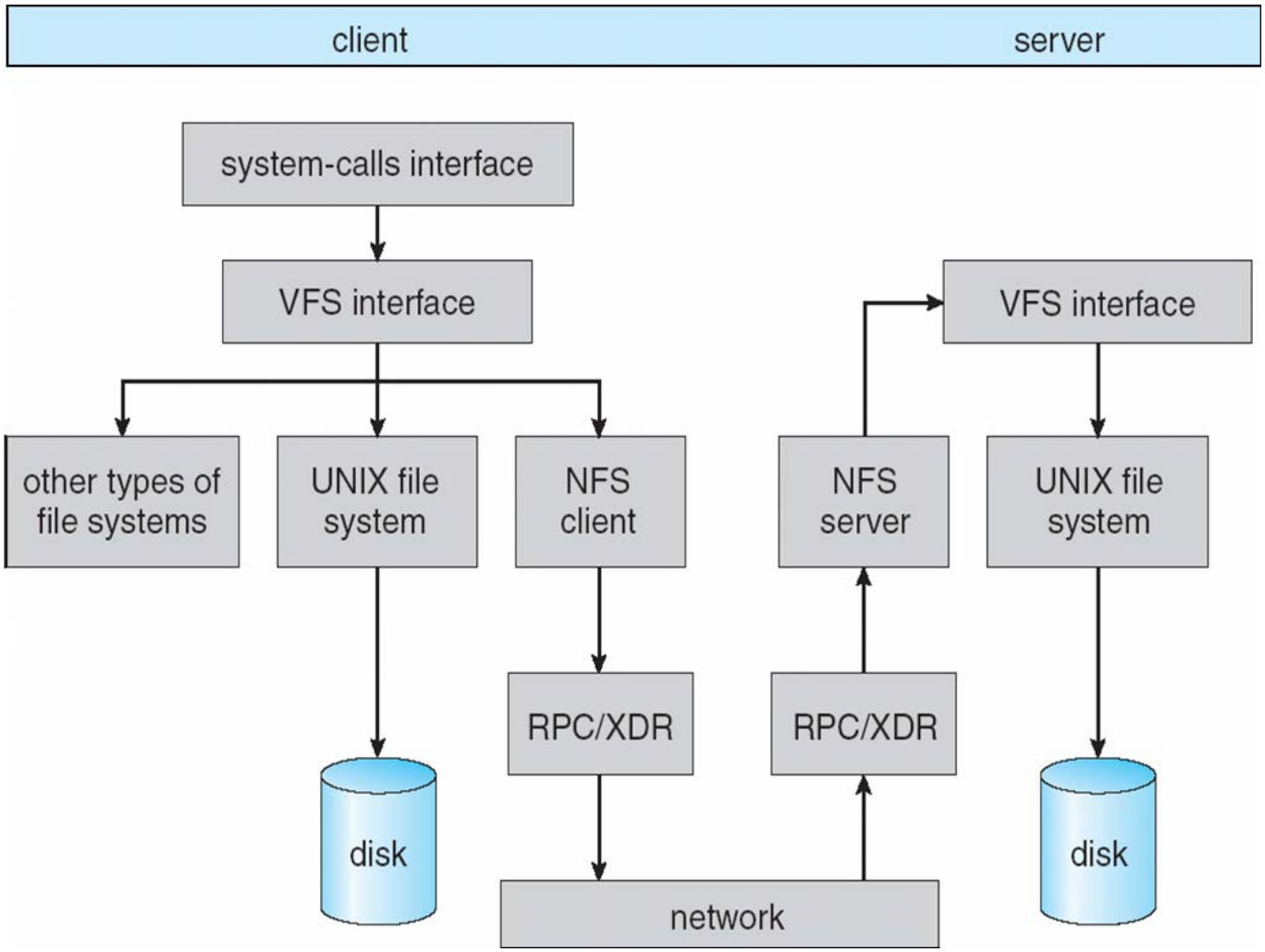
Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the **open**, **read**, **write**, and **close** calls, and **file descriptors**)

- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types
  - The VFS activates file-system-specific operations to handle local requests according to their file-system types
  - Calls the NFS protocol procedures for remote requests

- NFS service layer – bottom layer of the architecture
  - Implements the NFS protocol

Schematic View of NFS Architecture



NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files)
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes
  - Cached file blocks are used only if the corresponding cached attributes are up to date
- File-attribute (inode-information) cache – the attribute cache is updated whenever new attributes arrive from the server
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk

Summary

- 파일시스템의 구조
  - 계층적 구조
  - 논리적 파일시스템 → 파일 구성 모듈 → 기본 파일시스템 → 입출력 제어 → 장치(디스크)
- 파일시스템의 구현
  - on-disk: 부트 제어 블록, 볼륨 제어 블록
  - in-memory: 디렉토리 구조체, 시스템 오픈파일 테이블, 프로세스 소속 오픈파일 테이블
  - 파일 제어 블록(FCB)
  - 파티션과 마운팅
  - 가상 파일시스템과 구현(Unix)
    - objects: inode, file, superblock, dentry
- 디렉토리의 구현
  - 파일 이름의 선형 리스트
  - 해쉬 테이블 (해쉬 데이터 구조를 갖는 선형 리스트)
- 디스크 블록의 할당 방법
  - 인접 블록의 할당
    - extent-based
  - 링킹에 의한 할당
    - 파일 할당 테이블(FAT)
  - 인덱싱에 의한 할당
    - scalable index tables

- File-system structure
  - hierarchical
  - logical fs → file organization module → basic fs → I/O control → device(disk)
- File-system implementation
  - on-disk: boot control block, volume control block
  - in-memory: directory structure, system-wide open-file table, per-process open-file table
  - file control block (FCB)
  - partition and mounting
  - virtual file sys. and its implementation (Unix)
    - objects: inode, file, superblock, dentry
- Directory implementation
  - linear list of file names
  - hash table (linear list with hash data struct.)
- Allocation methods
  - contiguous allocation
    - extent-based
  - linked allocation
    - file allocation table (FAT)
  - indexed allocation
    - scalable index tables

Summary (Cont.)

- 자유공간의 관리
  - 미사용 블록의 할당
  - bit map, linked list
- 효율성과 성능 향상
  - 효율성: 디스크 공간의 효율적 사용
  - 성능: 버퍼 캐시, 페이지 캐시, 동기화/비동기화된 쓰기, 순차접근을 위한 기법 (free-behind, read-ahead)
  - 페이지 캐시와 버퍼캐시의 구성
- 복구
  - 일관성 검사
  - 로그 구조의 파일시스템
- NFS
  - 마운팅 방법
  - 프로토콜
    - 마운트 프로토콜
    - NFS 프로토콜
  - 원격 연산(작업)
    - 파일 블록 캐시
    - 파일 속성 캐시

- Free space management
  - allocation of unused, free blocks
  - bit map, linked list
- Efficiency and performance
  - efficiency: efficient use of disk space
  - performance: buffer cache, page cache, synchronous/asynch writes, techniques for sequential access (free-behind, read-ahead)
  - configuration of page cache and buffer cache
- Recovery
  - consistent checking
  - log-structured file system
- NFS
  - mounting methods
  - protocols
    - mound protocol
    - NFS protocol
  - remote operations
    - file-block cache
    - file-attribute cache