Operating Systems

17. Storage: File system implementation

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

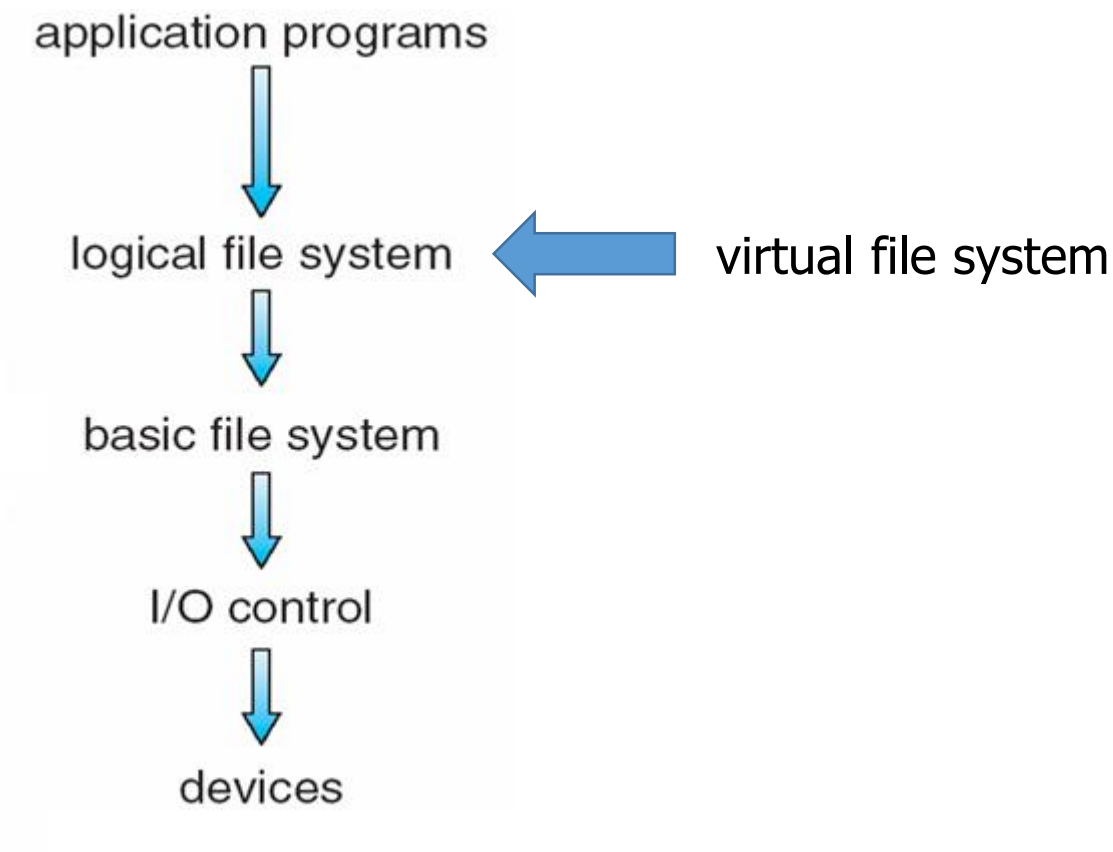
Contents

- Layered File-System Structure
- File-System Implementation
  - File and Directory Implementation
  - Allocation Methods
  - Free-Space Management
- Efficiency and Performance: Buffer cache
- Others
  - Unified buffer cache
  - NFS
  - Example: WAFL File System

File-System Structure

- 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
- And main memory after OS boots
  - In-memory FS structure: additional metadata for active files and directories
- File control block – storage structure consisting of information about a file
- File system organized into layers

Layered File System



File System Layers

- Device: provides in-place rewrite and random access
  - I/O transfers performed in blocks of sectors (usually 512 bytes)
  - Provides LBA (an abstraction of a block storage)
- I/O control: Device Drivers, I/O schedulers, and buffer cache
- Device drivers: manage I/O devices at the I/O control layer
  - Without LBA: Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific commands to hardware controller
- I/O scheduler: schedules I/O requests in a scheduler queue
- Buffer cache: Manages memory buffers and caches (allocation, freeing, replacement)
  - Buffers hold data in transit
  - Caches hold frequently used data

File System Layers (Cont.)

- Basic file system given command like "retrieve block 123 of file A" translates to I/O control
  - According to their own management schemes and algorithms (file, directory, and free space managements, allocation algorithms)
  - Provide other functionalities such as a recovery from system crash, de-fragmentation, version control, and etc.
- Logical file system manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks of basic file system (inodes in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance

Virtual File Systems

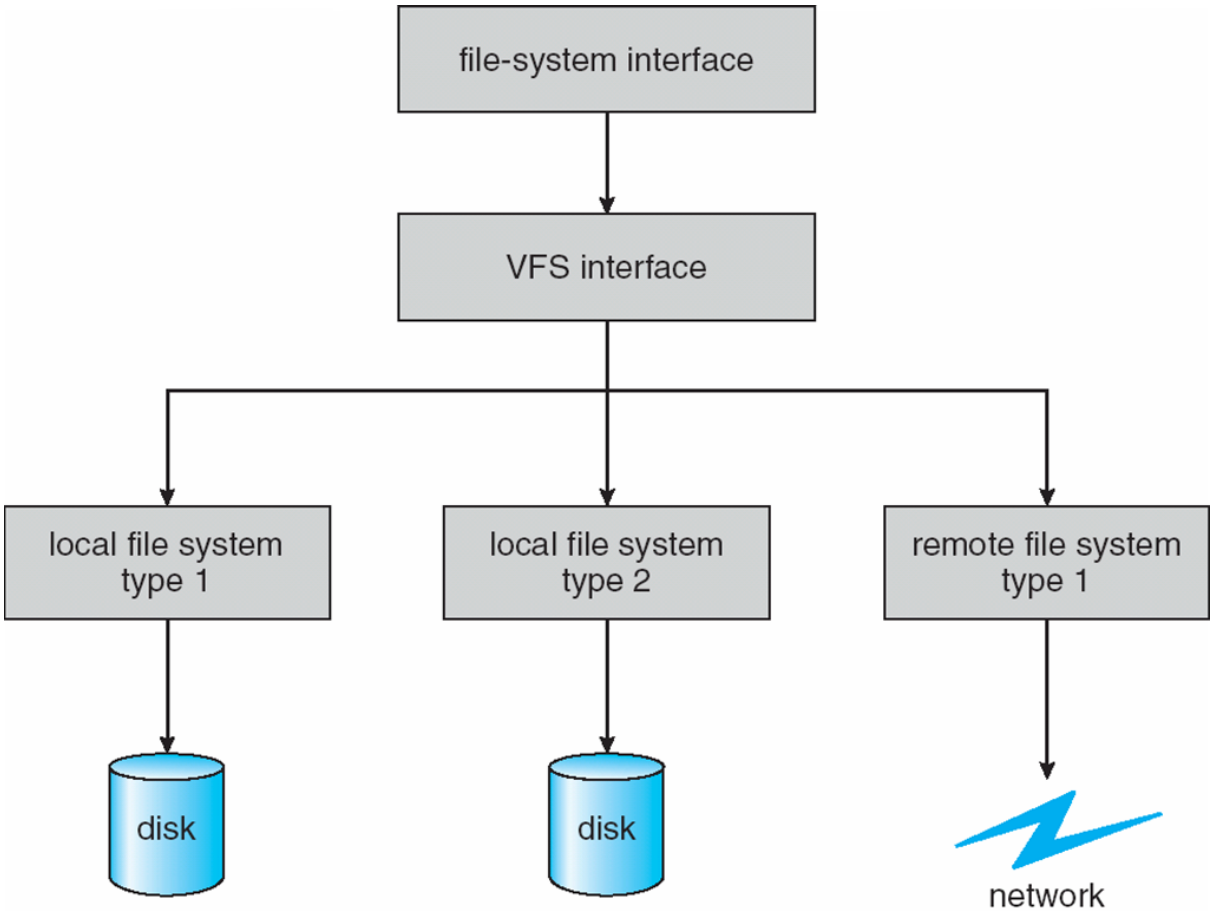
- VFS: supports many file systems, sometimes many within an operating system
  - Each with its own format (CD-ROM is ISO 9660; Unix has UFS, FFS; Windows has FAT, FAT32, NTFS as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types, with extended file system ext2 and ext3 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

Virtual File Systems

- Virtual File Systems (VFS) on Unix provide an object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - Separates file-system generic operations from implementation details
  - Implementation can be one of many file systems types, or network file system
    - Implements vnodes which hold inodes or network file details
  - Then dispatches operation to appropriate file system implementation routines

Virtual File Systems (Cont.)

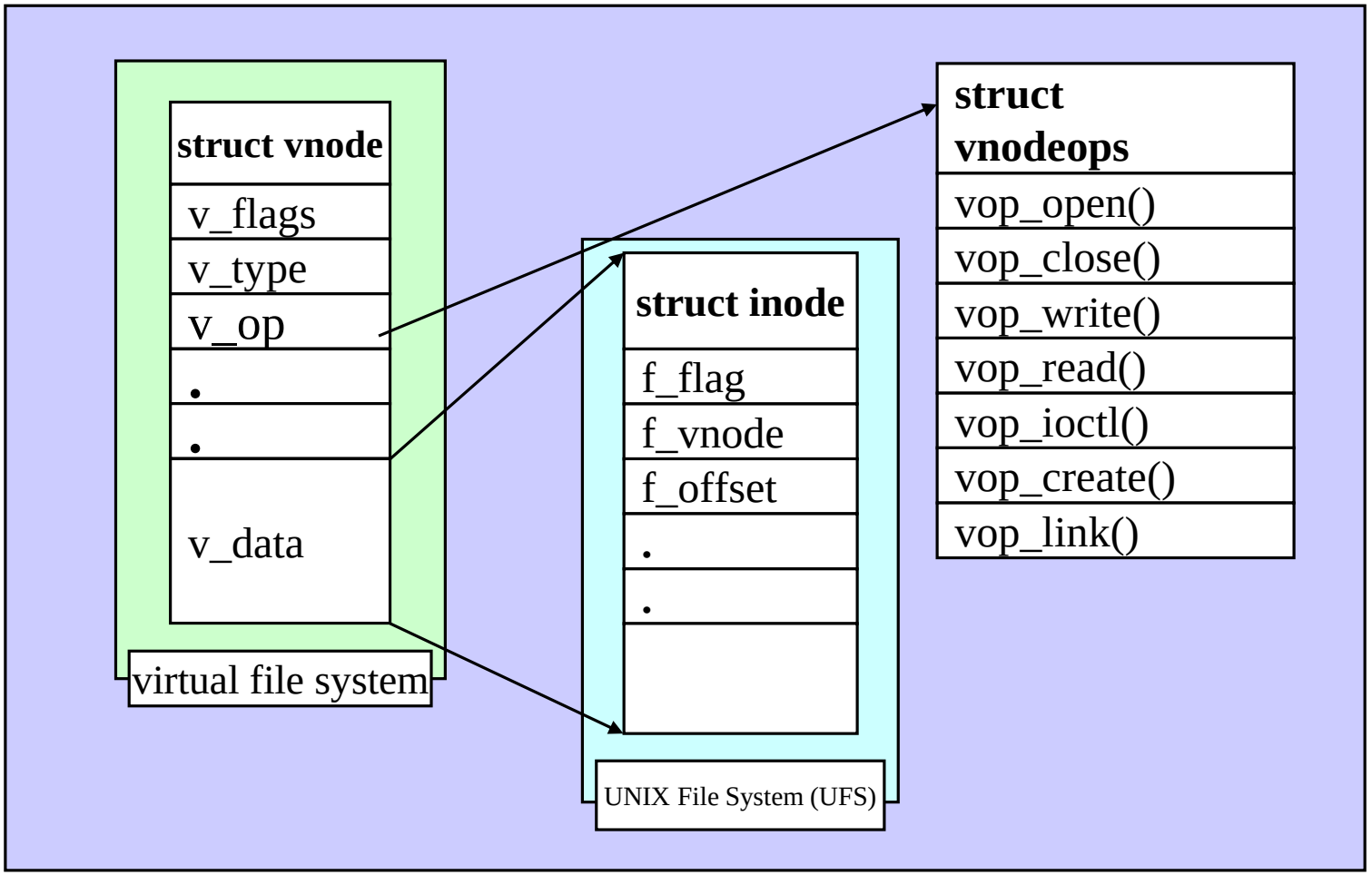
- The API is to the VFS interface, rather than any specific type of file system



Virtual File System Implementation

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

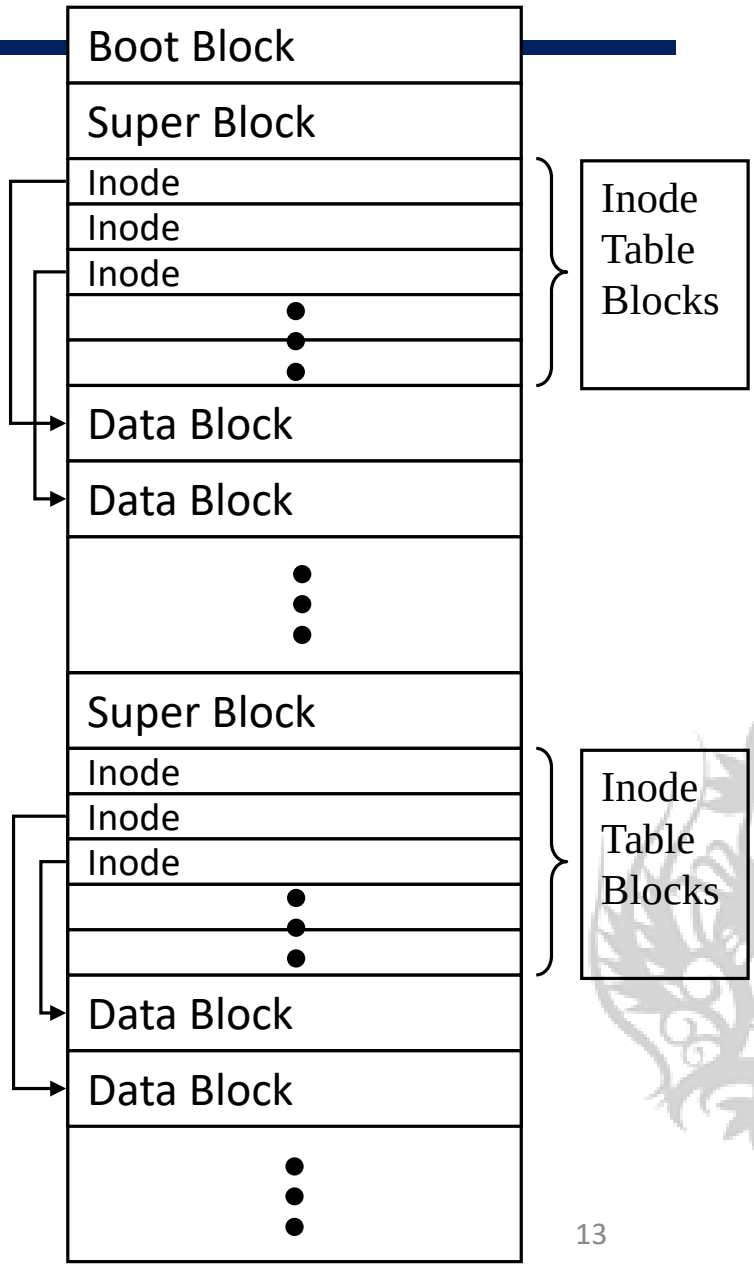
Virtual File System Implementation: Linux



File system Implementation

File-System Implementation <On-disk file system layout>

- Note that there are on-disk and in-memory FS structures
- Boot control block contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- Volume control block (superblock, master file table) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure (inode table blocks) organizes the files
  - Names and inode numbers, master file table



FS implementation: Partitions and Mounting

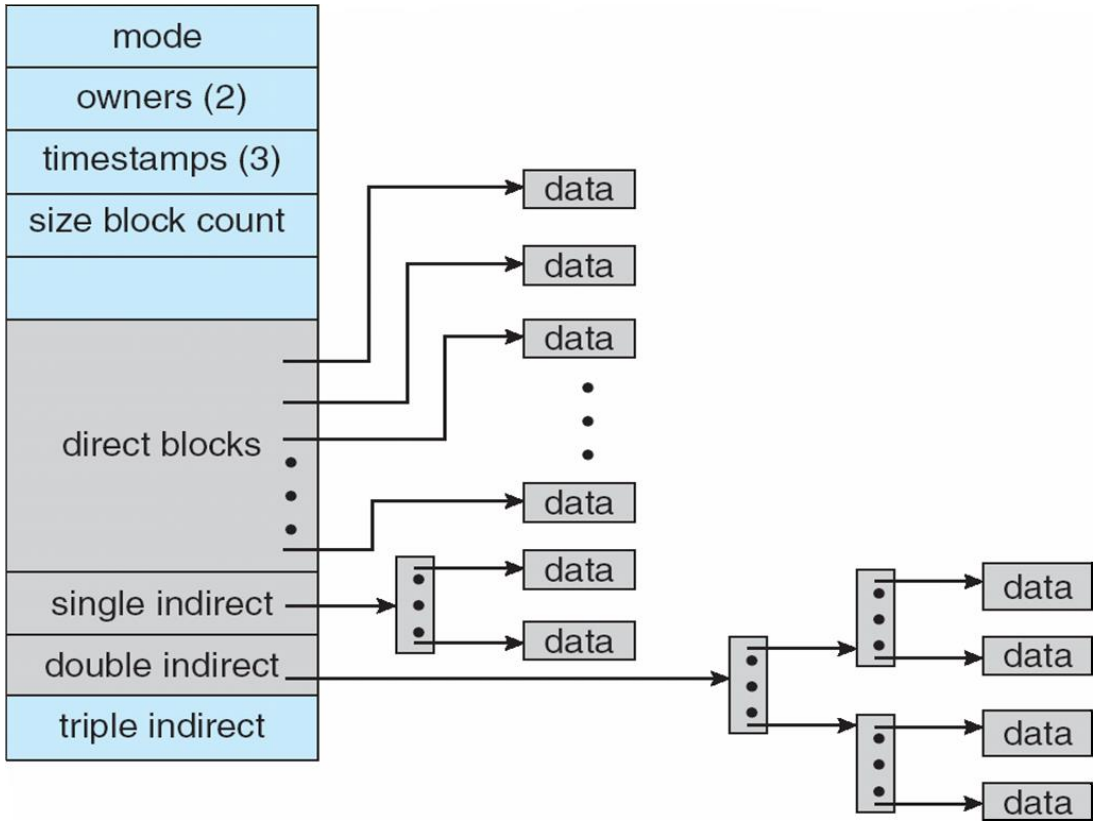
- Partition can be a volume containing a file system (“cooked”) or raw – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os 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 correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

File-System Implementation (Cont.)

- Per-file File Control Block (FCB) contains many details about the file
  - Usually called I-node (index node)
  - inode number, permissions, size, dates
  - NTFS stores into in master file table using relational DB structures

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

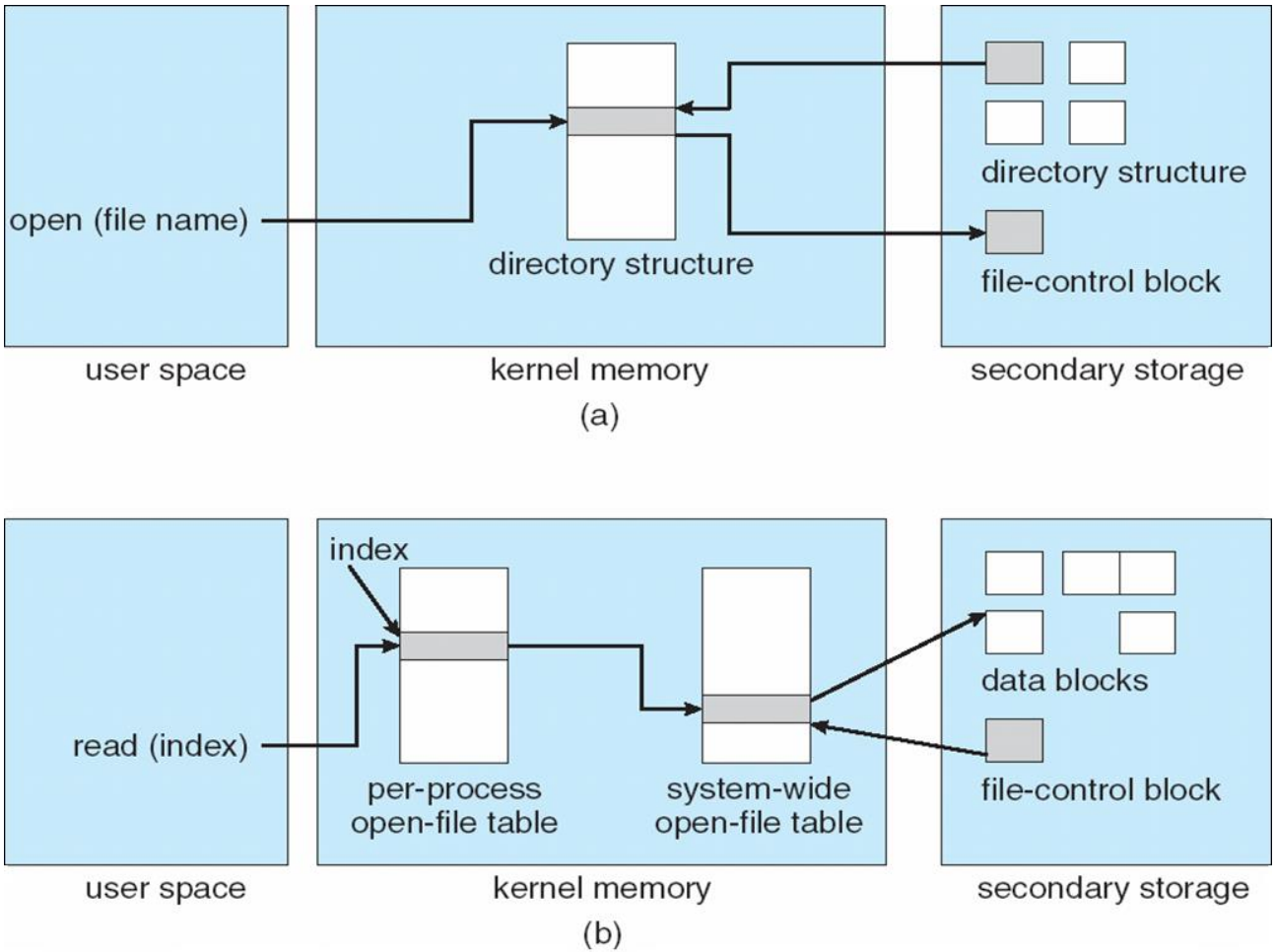
I-node of UNIX UFS



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 (a) refers to opening a file
  - Figure (b) refers to reading a file
- Plus buffers hold data blocks from secondary storage
- Open returns a file handle for subsequent use
- Data from read eventually copied to specified user process memory address

In-Memory File System Structures

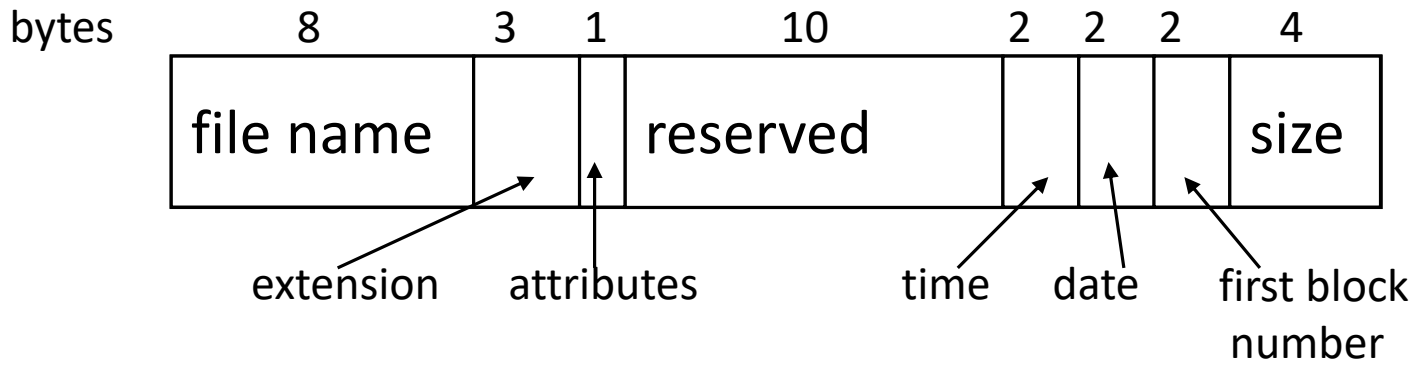


Directory Implementation

- 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
  - Directory is stored in disk as a file
- 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

Directories in MS-DOS

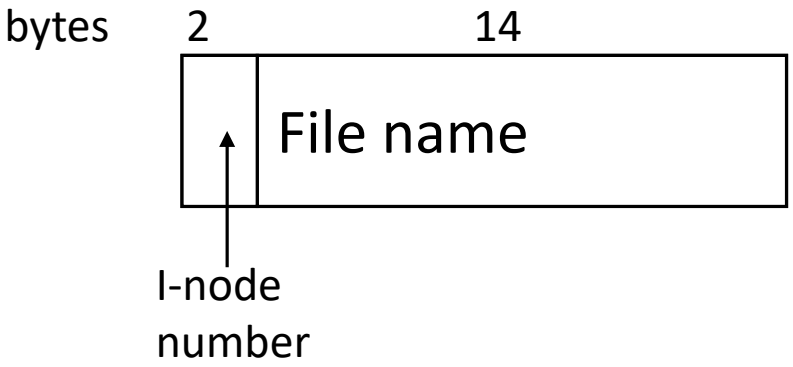
- Uses Linked List Allocation
- Directory entry:



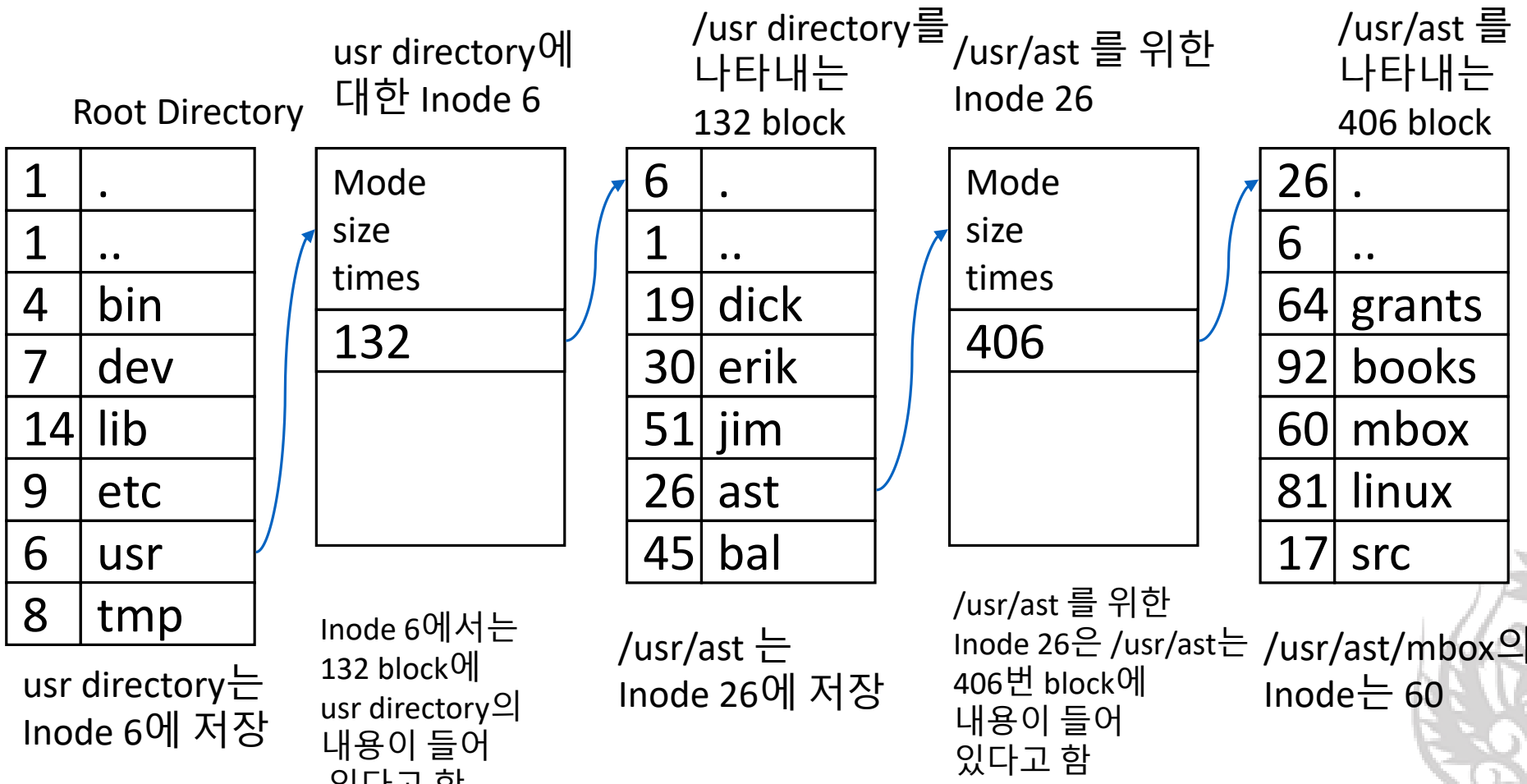
- First block number
  - The start position of data blocks

Directories in UNIX

- UNIX directory entry: file name and i-node number



Directories in UNIX (cont.)

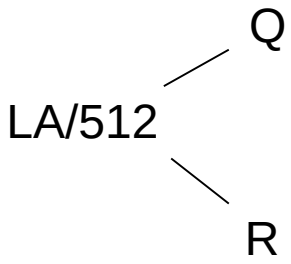


Allocation Methods - Contiguous

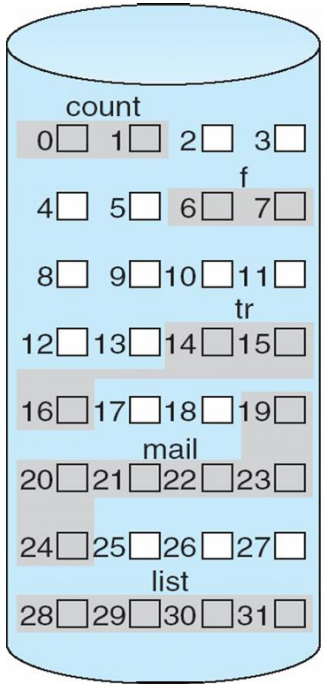
- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - Indexed
- Contiguous allocation – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Contiguous Allocation

- Mapping from logical to physical



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

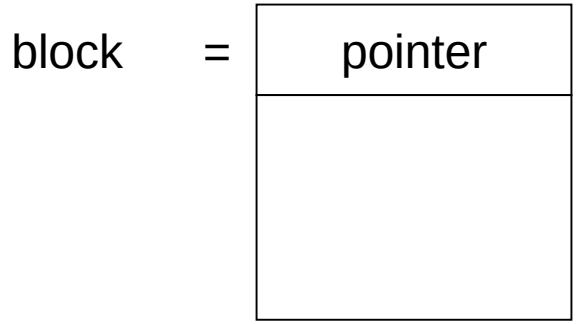


directory

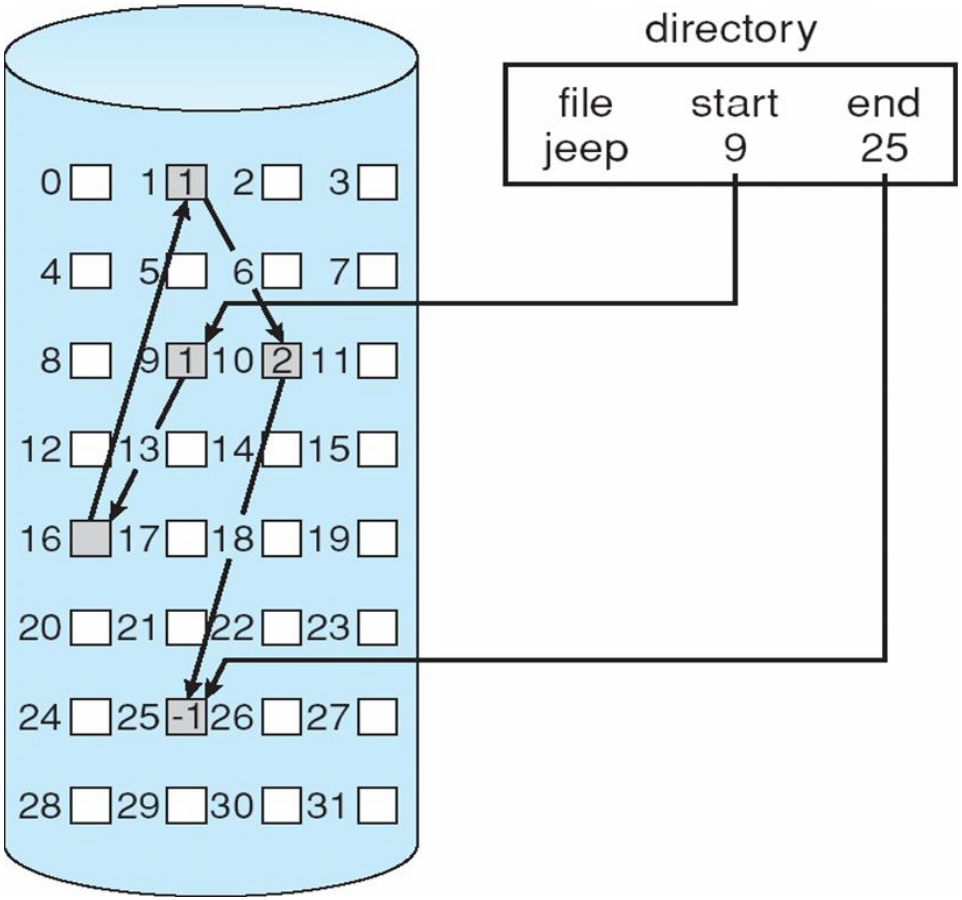
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation Methods - Linked

- Linked allocation – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - No compaction, external fragmentation
  - 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



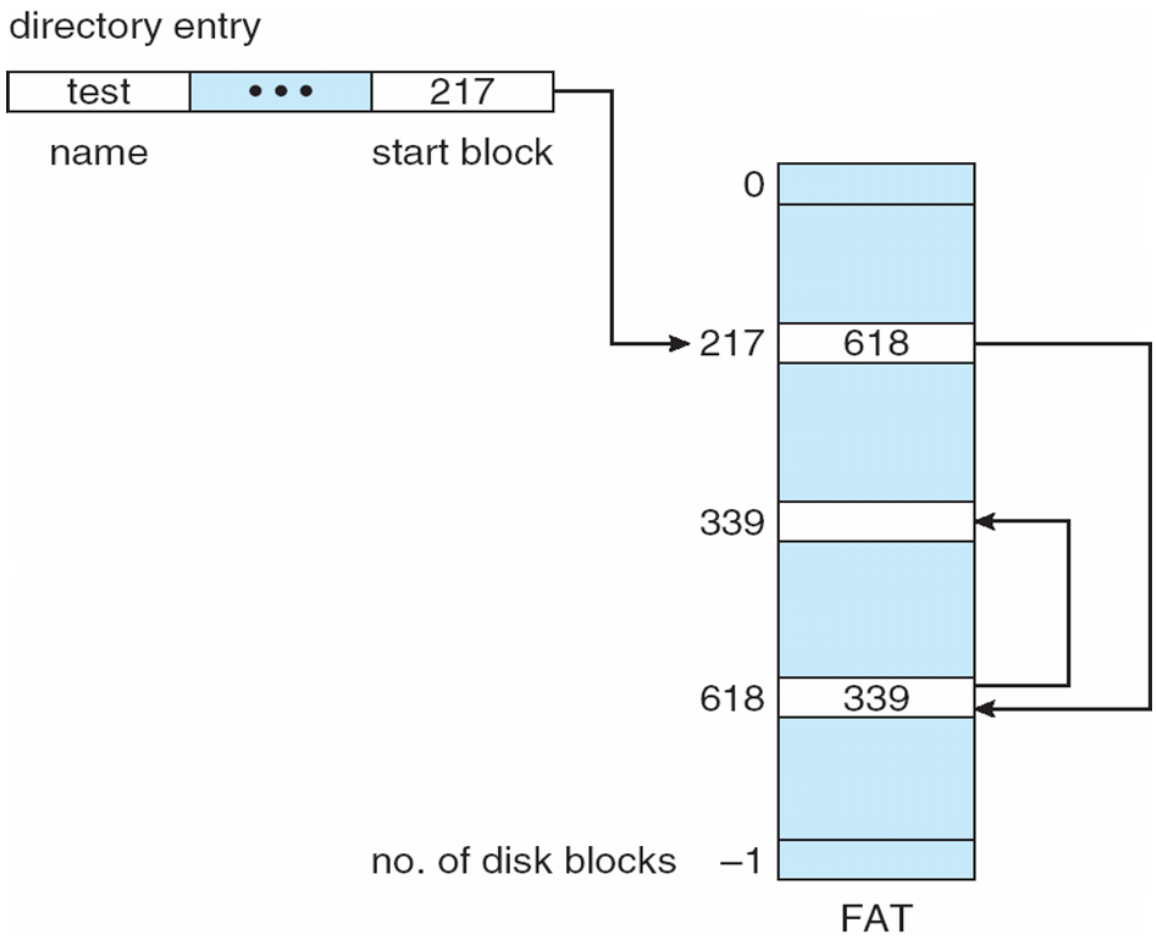
Linked Allocation



Allocation Methods – Linked (Cont.)

- 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 is simple

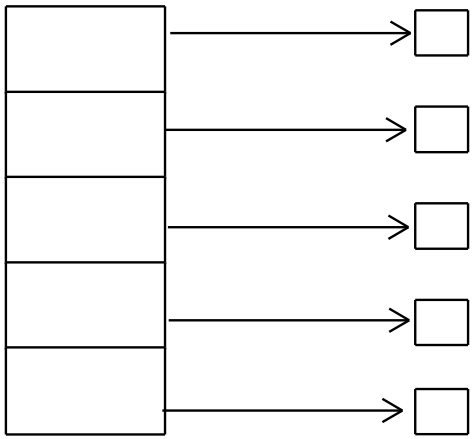
File-Allocation Table



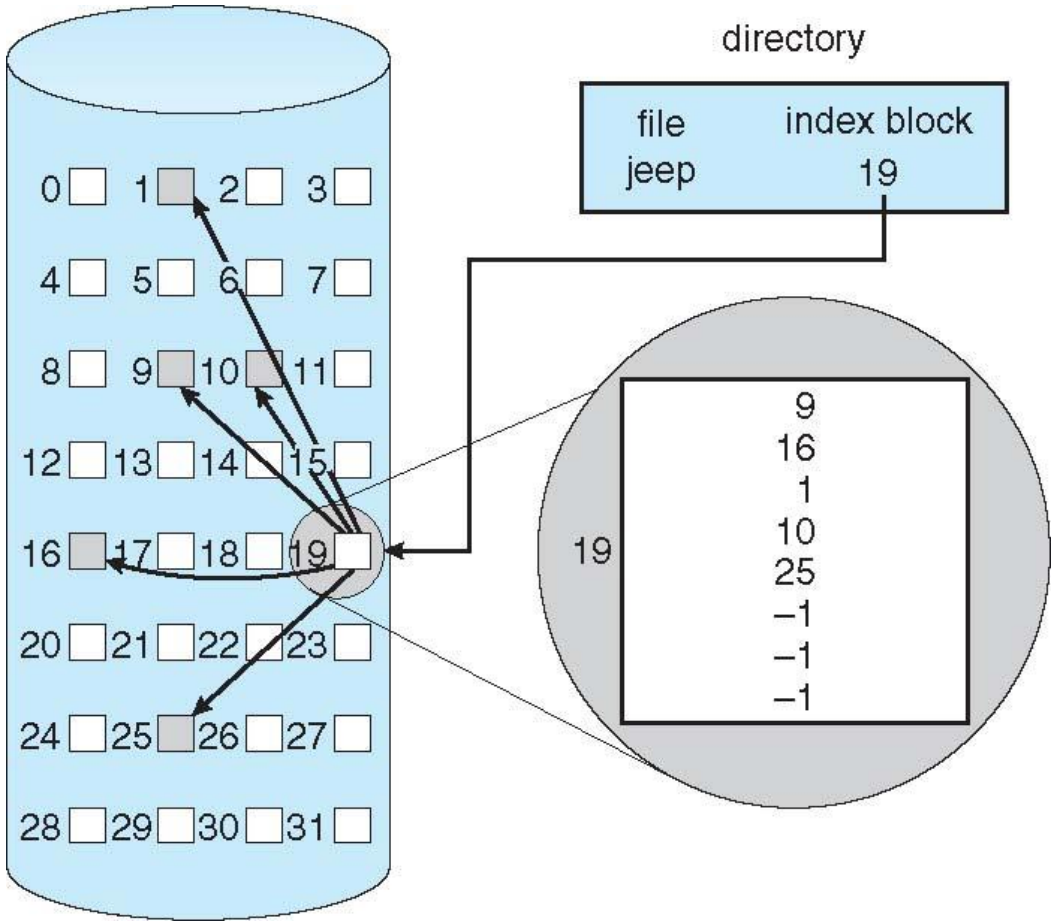
Allocation Methods - Indexed

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

- Logical view



Example of Indexed Allocation

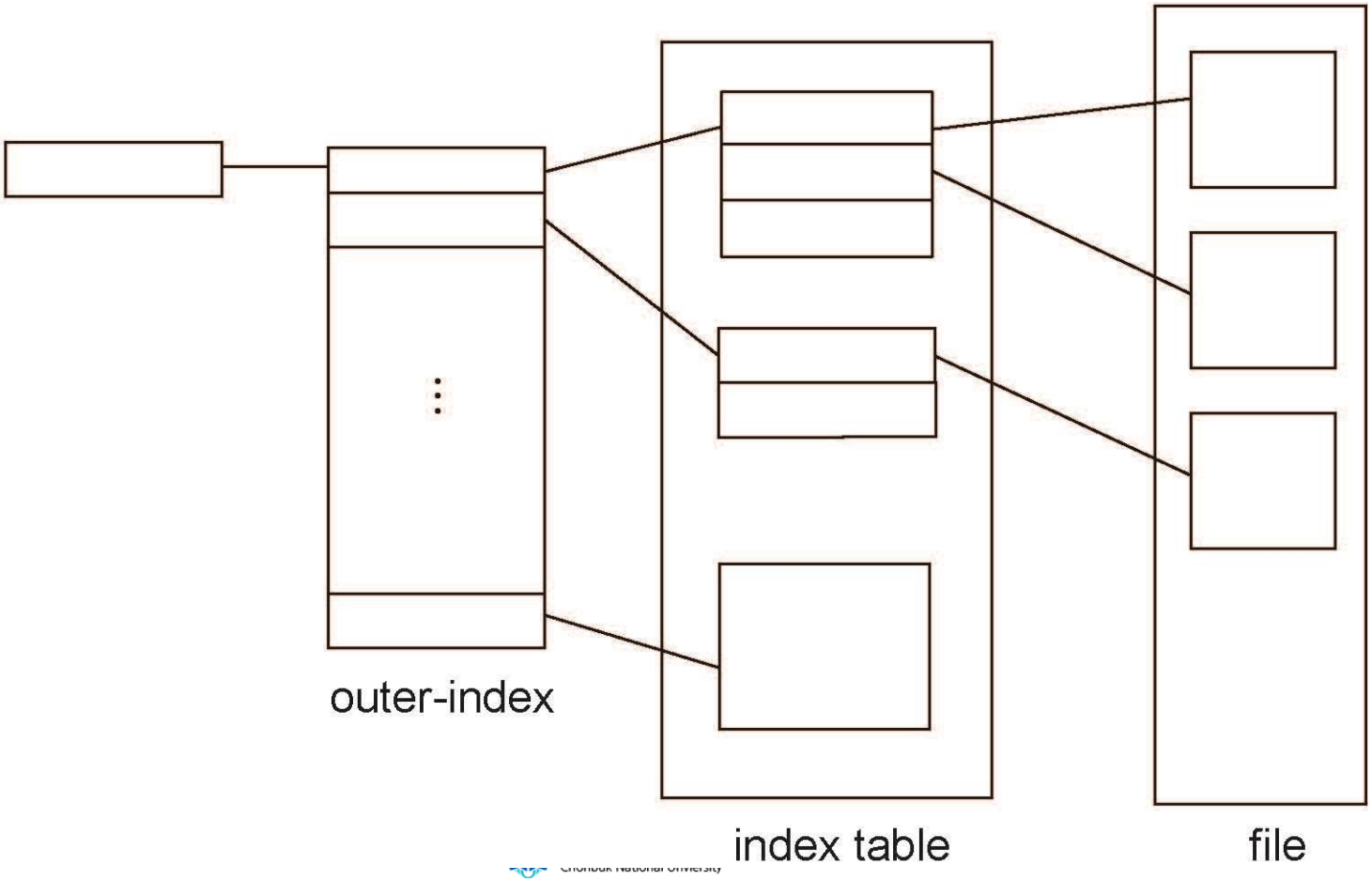


Indexed Allocation (Cont.)

- Need index table: in another disk block(s)
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- File size
  - 4KB block, 4B addressing
  - With 1 index block: addresses 1024 blocks = 4 MB file
    - Allocated blocks: 1024 + 1
      - For an Index block
      - How about FCB?

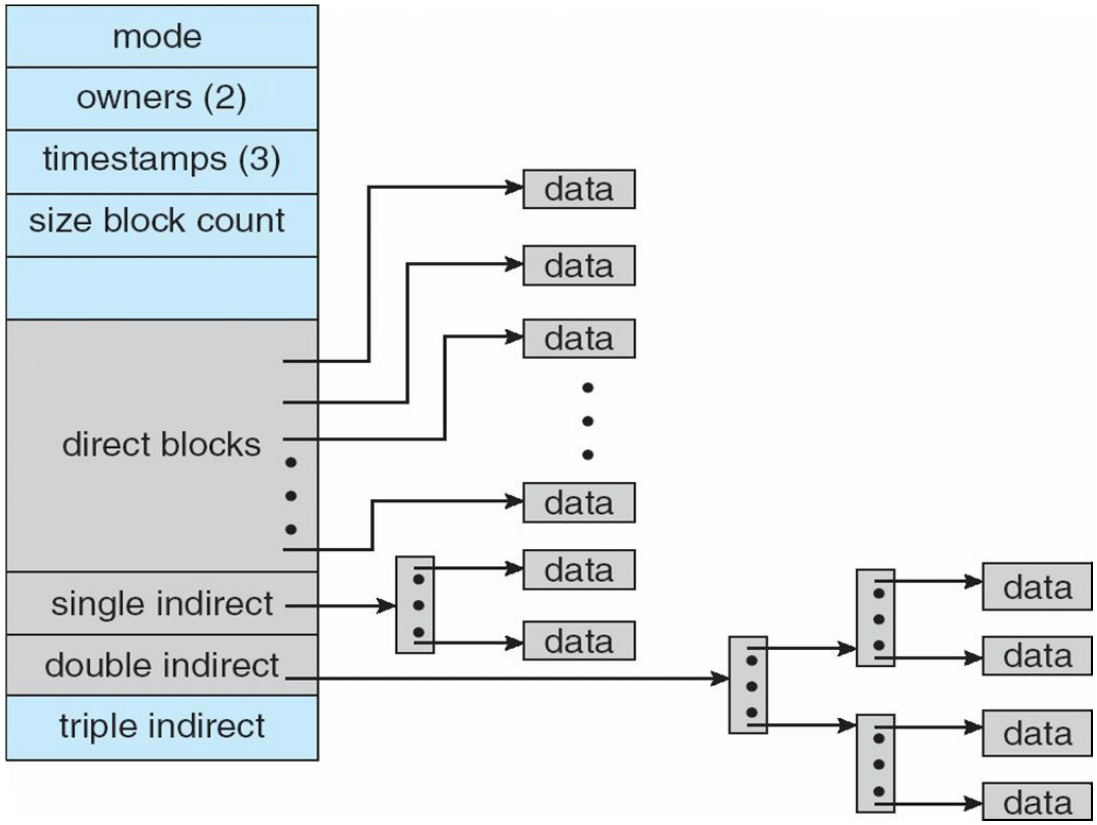
Indexed Allocation – Mapping (Cont.)

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



Combined Scheme: I-node of UNIX UFS

4K bytes per block, 32-bit addresses



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

Quiz!

- How many bytes can be stored in 30 blocks?
  - Block: 512 bytes, Address: 4 bytes
  - 1) Contiguous allocation:512 x 30 = 15360 B
  - 2) Linked: (512-4) x 30 = 15240 B
  - 3) Indexed: 512 x 29 = 14848 B
- Maximum file size for a UFS file?
  - 12 direct blocks:
  - 1 Indirect index block:
  - 1 double indirect :
  - 1 triple indirect:

Performance on allocation methods

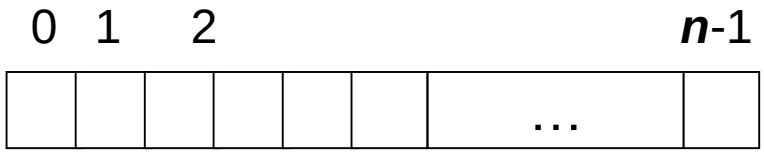
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read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

Performance on allocation methods (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)

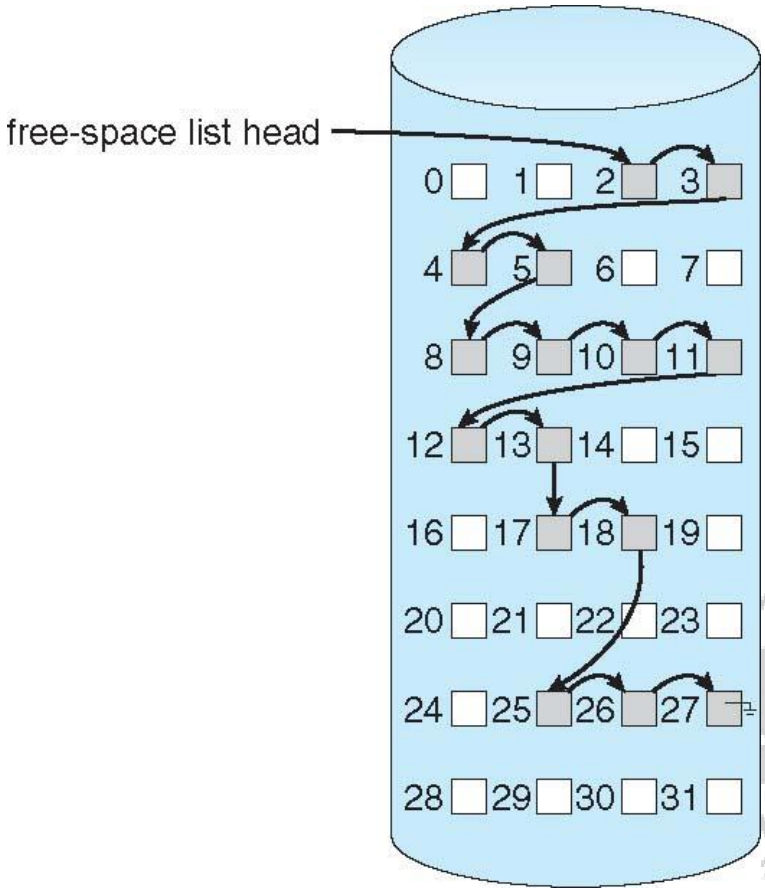
    n = $2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

Linked Free Space List on Disk

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



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 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

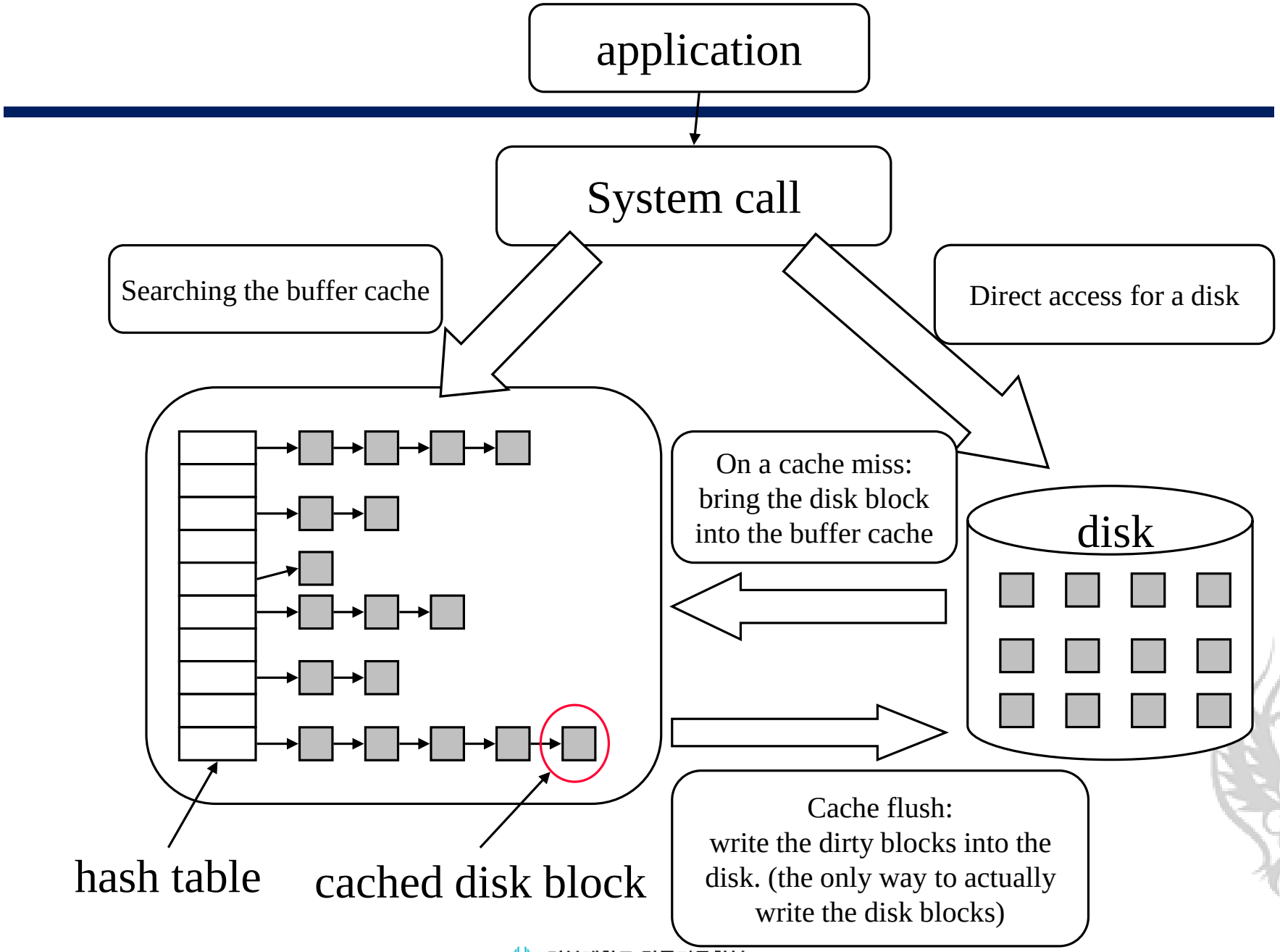
Buffer cache

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

Efficiency and Performance (Cont.)

- Performance
  - With a disk (HDDs): Keeping data and metadata close together
  - Buffer cache – separate section of main memory for frequently used blocks
    - Not a H/W unit (how about CPU caches and TLB?)
  - 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 because of write cache


application
System call
Searching the buffer cache
Direct access for a disk
On a cache miss:
bring the disk block
into the buffer cache
disk
Cache flush:
write the dirty blocks into the
disk. (the only way to actually
write the disk blocks)
hash table
cached disk block

Write System Call – klushd

- kflushd
  - The Linux kernel thread synchronizes the buffer cache and disk blocks
  - Target: dirty blocks that are modified by users (or kernel)
  - Periodically runs (e.g. 1 s) and flush the dirty blocks to the disk
    - And replace the buffers according to its own replacement algorithm
  - fsync(int fd) system call: immediately runs kflushd and flushes the dirty blocks for a file associated with fd

- Note that the storage system with a write cache cannot guarantees that write()–like system calls immediately update the disk blocks

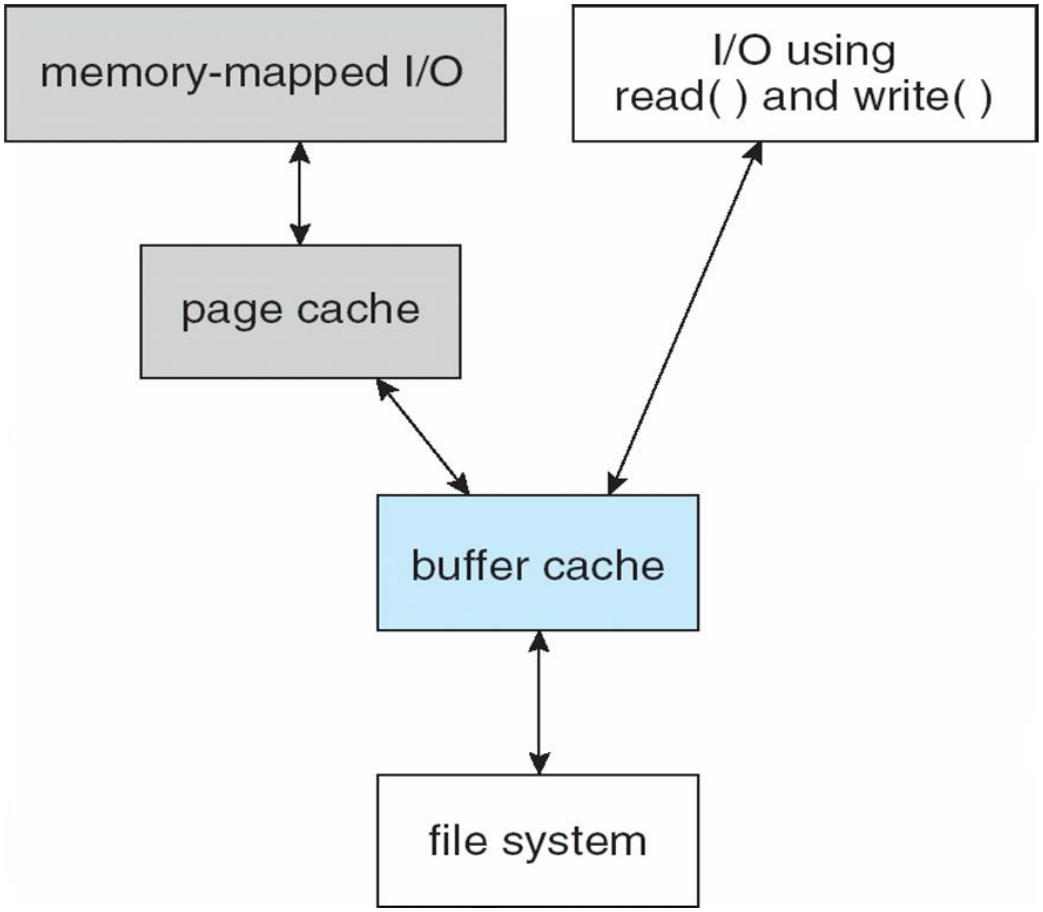
Recovery

- 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

Page 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

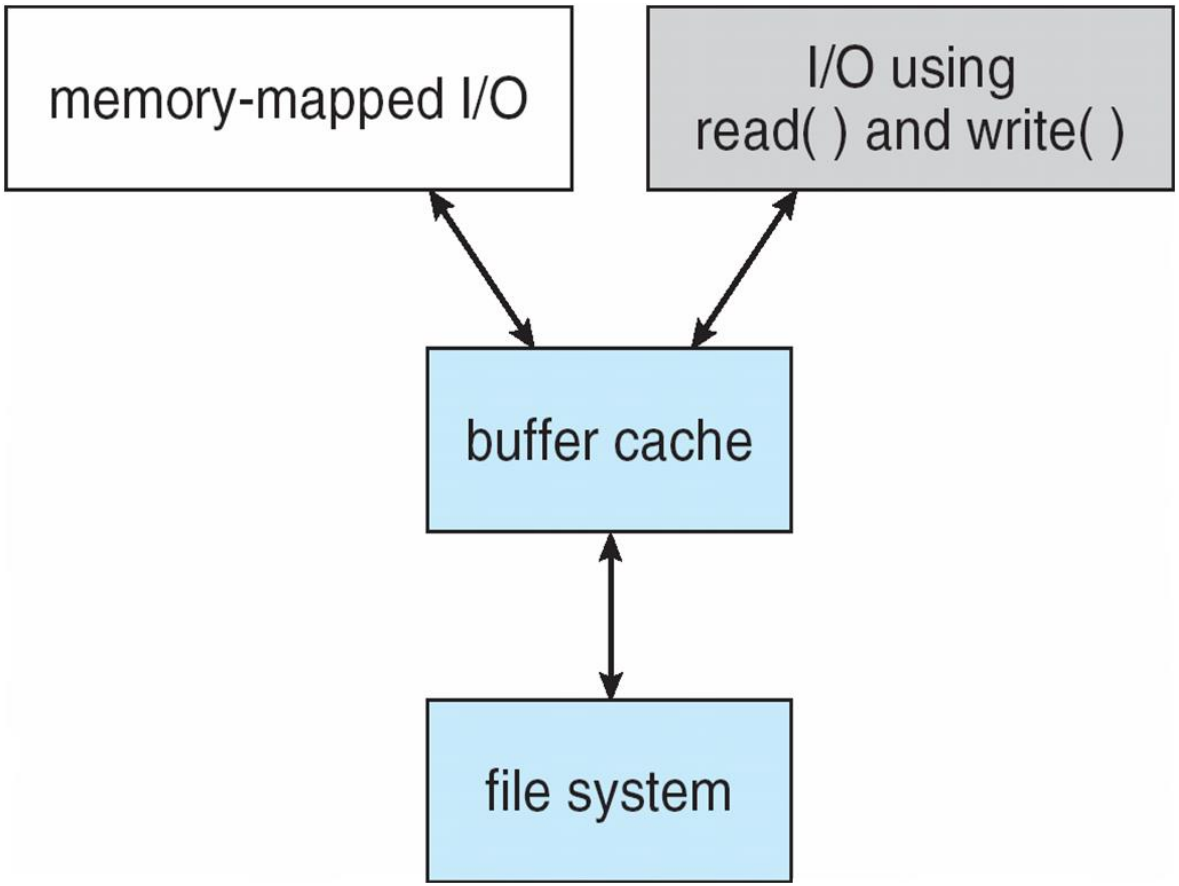
I/O Without a Unified Buffer Cache



Unified Buffer Cache

- 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 Using a Unified Buffer Cache



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 (sequentially)
  - 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, which allows sharing among these file systems in a transparent manner
  - A remote directory is mounted over a local file system directory
    - The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided
    - Files in the remote directory can then be accessed in a transparent manner
  - 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 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