Chapter 14: File System Implementation

CS 3423 Operating Systems

Fall 2019

National Tsing Hua University

Outline: 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

File-System Structure

- File system (FS)
  - Resides on secondary storage (disks)
  - Provides mapping logical storage to physical storage
  - Provides efficient implementation and convenient API to disk
  - One OS may support multiple types of FS (e.g., FAT32, NTFS)
- Disk
  - provides in-place rewrite and random access to storage
  - I/O transfers in units of blocks (=one or more sectors)
  - one sector usually 512 bytes

Two design problems in FS

- interface to user programs
  - API for user programs `open()`, `read()`, `write()`,
  - mostly independent of file system
- interface to physical storage (disk)
  - API for accessing disk
  - mostly independent of actual disk, isolated by I/O control layer

Layered File System

user	application programs ↓	`read`(fh, buf, size)
logical	logical file system ↓	manages metadata (fp, access) manages memory buffers and caches (allocation, freeing, replacement)
	file-organization module ↓	logical <-> physical mapping Translates logical to physical block # Manages free space, disk allocation
	basic file system ↓	read d1, c735, t5, s10
physical	I/O control ↓	read drive1, cylinder 72, track 2, sector 10, into memory location 1060
	devices	retrieve block 123

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
  - adds overhead and can decrease performance

Multiple file systems within an OS

- CD-ROM is ISO 9660
- Unix has UFS, FFS
- Windows
  - FAT, FAT32, NTFS, floppy, CD, DVD Blu-ray
- Linux ext2, ext3, ext4, distributed FS
- New ones still being invented
  - ZFS, GoogleFS, Oracle ASM, FUSE

File System Implementation

On-disk structures

- Boot control block (per volume or per partition)
  - contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- Volume control block (also called partition control block)
  - (superblock, master file table) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure (per file system)
  - Names and inode numbers, master file table
- File control block (per file)
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational database structures

On-Disk Structure

partition
(aka volume)

Boot Control Block (Optional)
Partition Control Block
List of Directory Control Blocks
List of File Control Blocks
Data Blocks

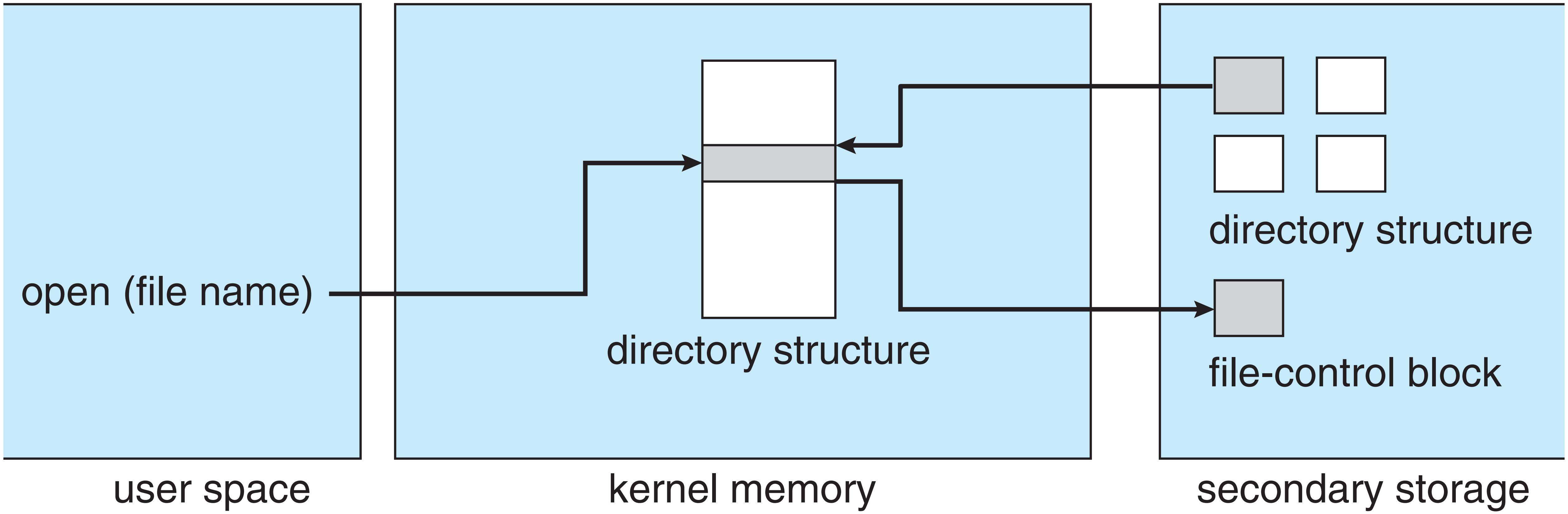
file control block (FCB)

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

In-Memory Structures

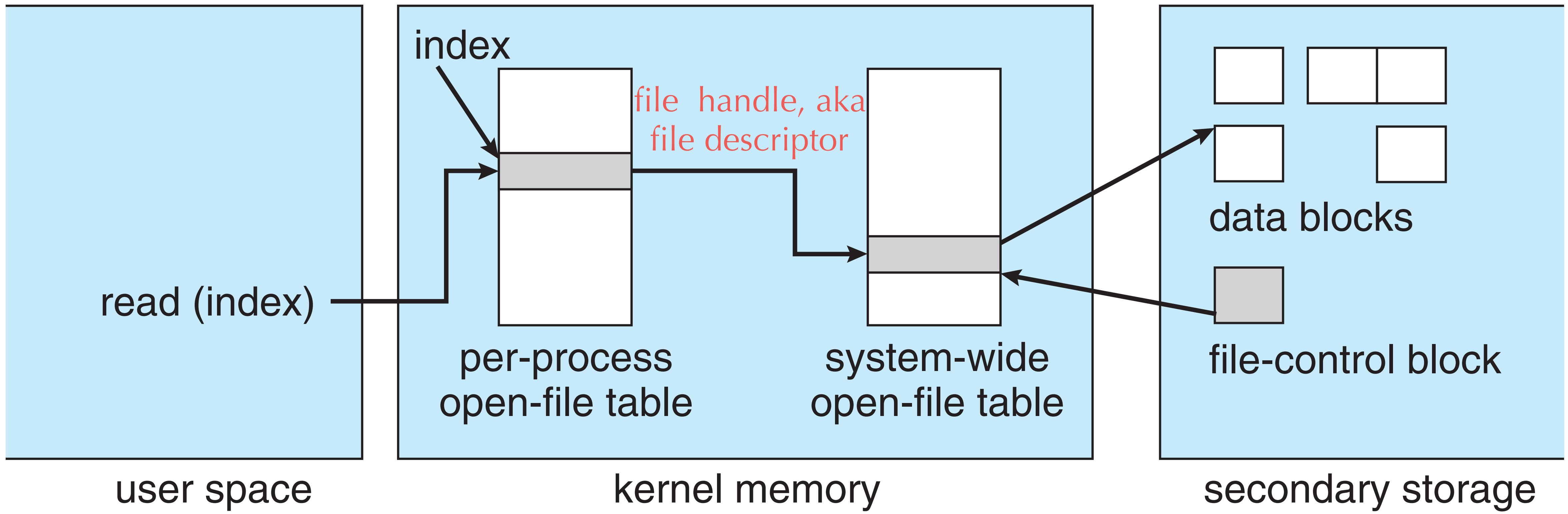
- in-memory Mount table
  - stores file system mounts, mount points, file system types
- in-memory directory structure (as opposed to on-disk)
  - recently accessed directories
- system-wide open-file table
  - contains a copy of each open file's FCB
- per-process open-file table
  - file handle (pointer) to corresponding entry in systemwide table
- buffers that hold data blocks from secondary storage

open(file name)



search directory structure for file by name
if not in memory already,
    add entry to system-wide open file table, fill with FCB and initialize count
add per-process open file table to point to system-wide open file table entry

read(index)



per-process open-file table[index] => entry in system-wide open-file table
advance file pointer by number of bytes read

File Creation Procedure

- OS allocates a new FCB
- OS updates directory structure
  - OS reads the directory structure into memory
  - OS updates dir structure with new name & FCB
  - OS writes dir structure back to disk upon file close
- The file appears in user's directory

Directory Implementation options

- Linear list
  - List of file names with pointer to the data blocks
  - Simple to program but poor performance to execute
    - Linear search time
  - Could keep ordered alphabetically via linked list or use B+ tree
- Hash Table – linear list with hash data structure
  - Search time is constant time in most cases
  - Collisions => requires probing
  - Hash table good for fixed number of entries

Allocation Methods:

- how disk blocks are allocated for files

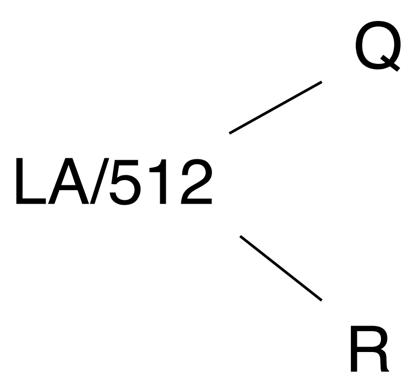
Contiguous allocation

Linked allocation

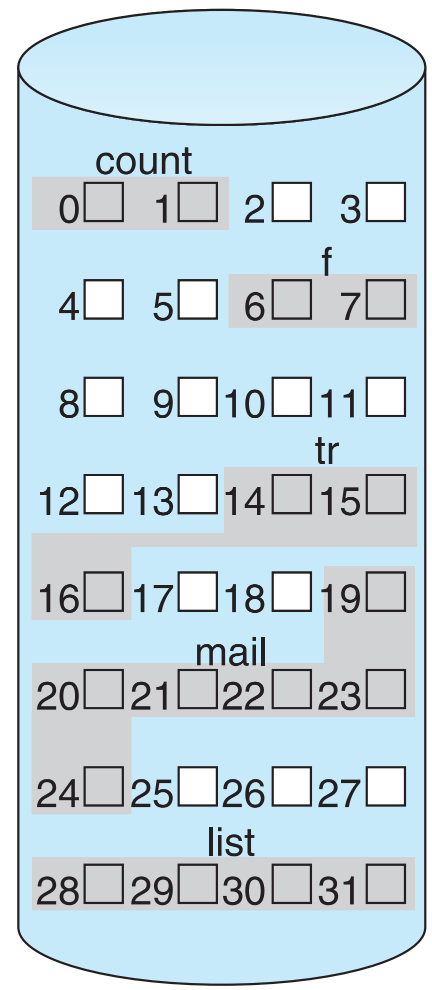
Indexed allocation

Contiguous Allocation

- File occupies contiguous blocks
-



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

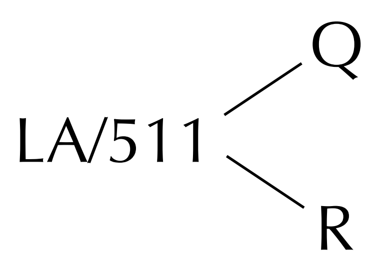
- Pros
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required
- Cons
  - Problems finding space for file
  - have to know size in advance, not easy to change
  - external fragmentation
  - need for compaction off-line (downtime) or on-line

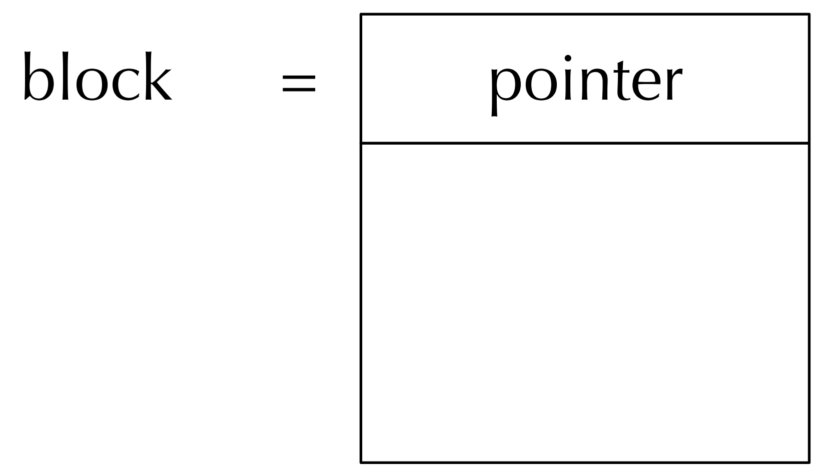
Extent-Based Systems

- An extent = a contiguous sequence of blocks on disk
  - starting block#, length, pointer to next extent
  - an extent file => a linked list of extends
  - Example: Veritas File System (replacement for UFS)
- Issues:
  - random access more costly
  - both internal & external fragmentation

Linked Allocation

- Each file is a linked list of blocks
  - Each block contains pointer to next block till nil
  - Improve efficiency by clustering blocks into groups

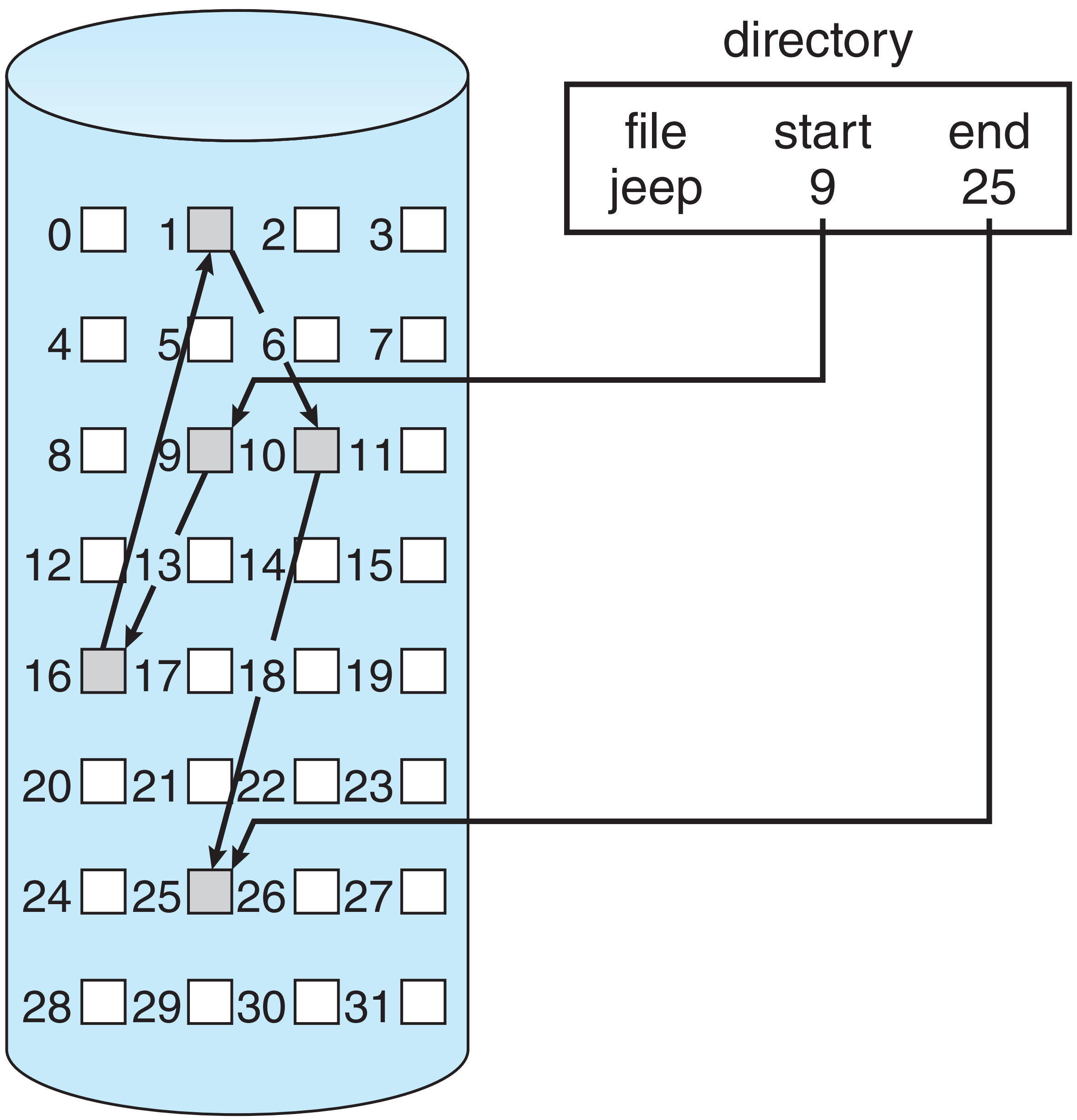




- Block to be accessed is the Q^{th} block in the linked chain of blocks representing the file.
  - Displacement into block = R + 1

Linked Allocation

- Each file is a linked list of disk blocks:
  - blocks may be scattered anywhere on the disk
-

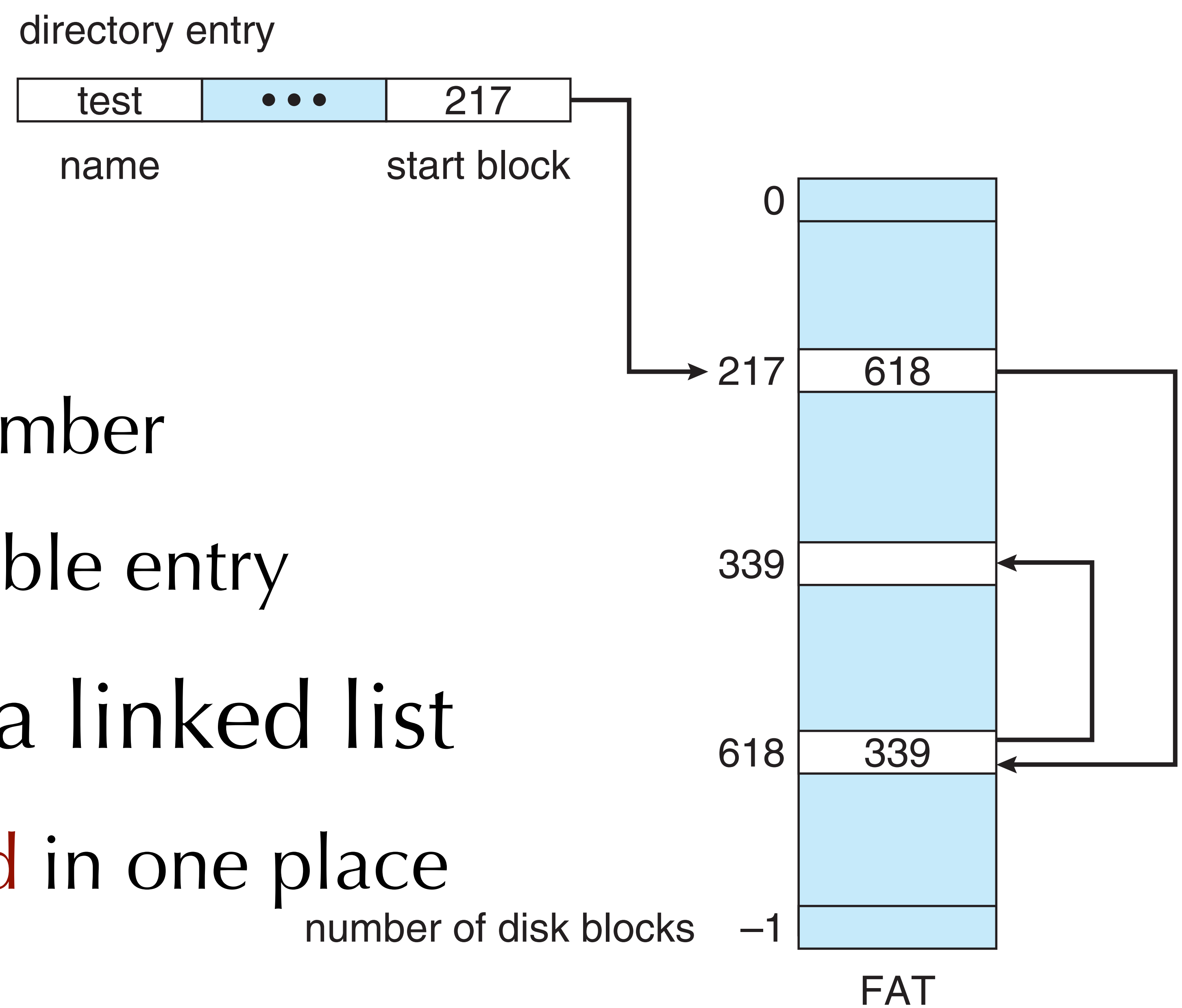


Linked Allocation

- Pros
  - No external fragmentation
  - Good for sequential access
- Cons
  - Reliability can be a problem: one missing link breaks whole file!
  - Random access may take many I/Os and disk seeks

FAT (File Allocation Table)

- Used in Windows, USB drive, MS DOS
- Beginning of volume has table of all links
  - indexed by block number
  - FAT32: 32 bits per table entry
- Conceptually still a linked list
  - all links consolidated in one place

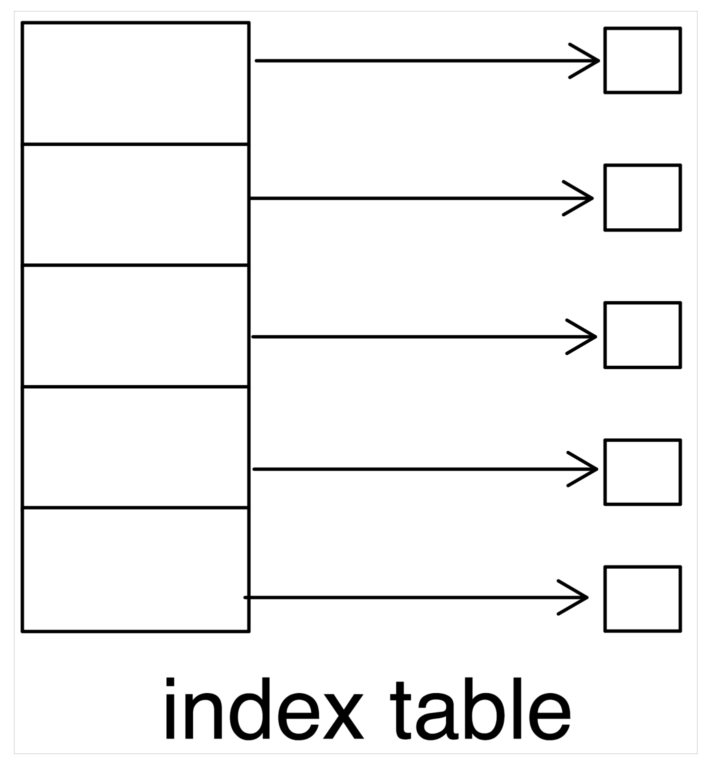


File-Allocation Table

- Pros:
  - Simple for new block allocation
  - FAT can be cached
- Potential cons:
  - flash memory: FAT blocks get more wear-and-tear => need wear-leveling (SD cards do this automatically)
  - if FAT is corrupted => lose links

Indexed Allocation

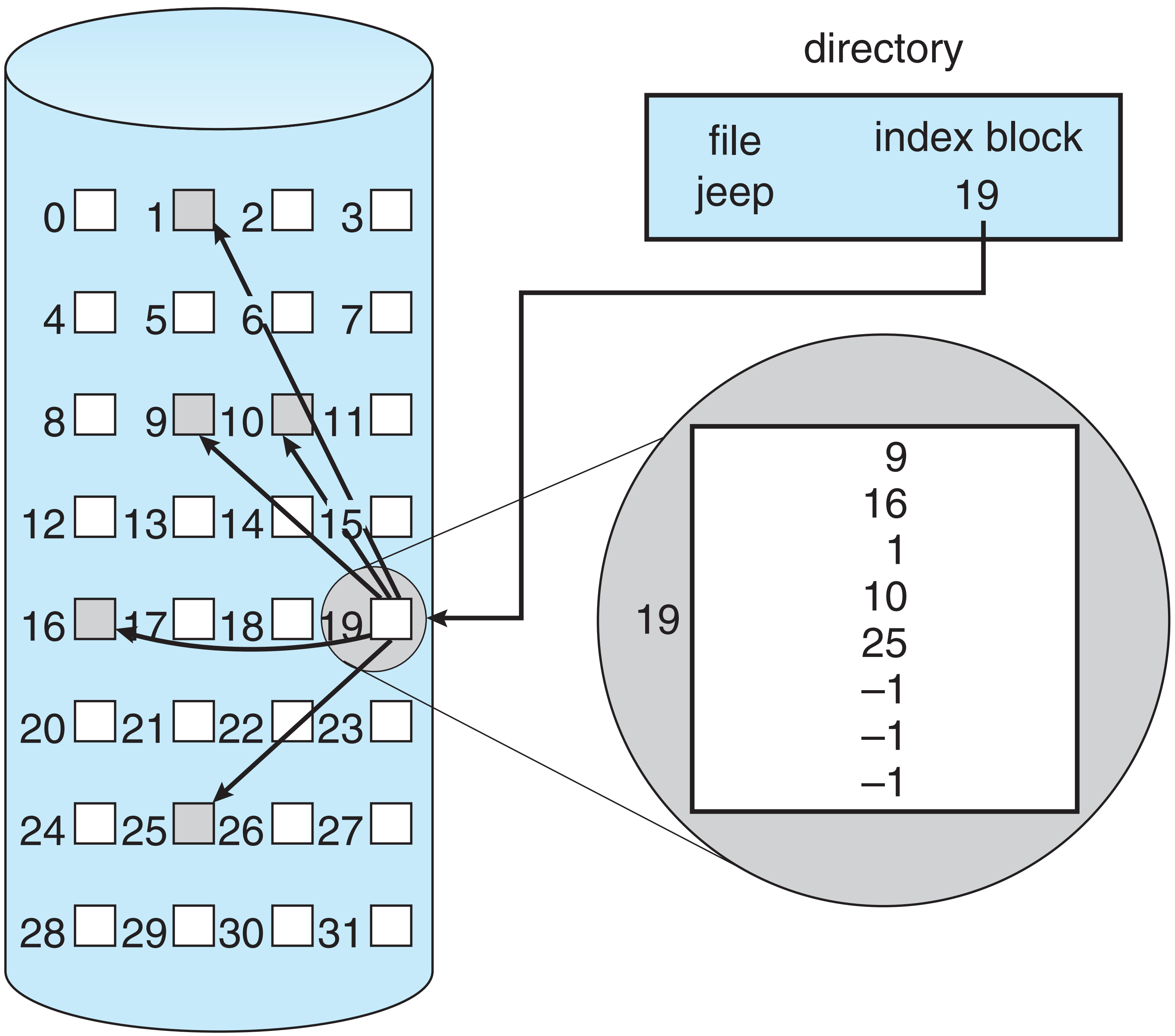
- Each file has its own index
  - index = table of pointers to its data blocks
- Logical view



Indexed Allocation

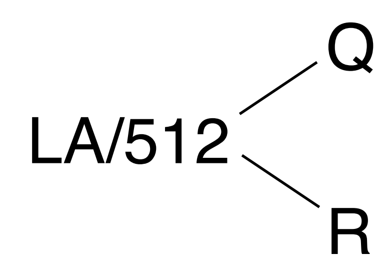
- Pros
  - more efficient random access: look into table
  - no external fragmentation
  - easy to create a file (no allocation problem)
- Cons
  - overhead space taken by index table
  - unclear how large the index table should be
    - linked scheme, multilevel index
    - combined scheme (BSD Unix inode)

Example of Indexed Allocation



Indexed Allocation Example

- Max file size = 256 KB
- Block size = 512 words
  - a word is enough to address block space
- => need only 1 block for index table



Logical Address (LA) divmod 512

quotient Q = displacement into index table
remainder R = displacement into block

Indexed Allocation – Mapping (Cont.)

- block size = 512 words
  - assuming word size is large enough for block space
- Linked scheme:
  - Link blocks of index table
  - no limit on size

$LA / (512 \times 511)$ → Q_1, R_1

Q_1 = block of index table

R_1 is used as follows:

$R_1 / 512$ → Q_2, R_2

Q_2 = displacement into block of index table

R_2 displacement into block of file:

two-level index scheme

- 4K blocks could store 1,024 four-byte pointers in outer index
  - 1,048,567 data blocks and file size of up to 4GB

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

LA / (512 x 512) → Q_1, R_1

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

R_1 / 512 → Q_2, R_2

Combined Scheme: UNIX inodes

- File pointer: 4 bytes (32 bits) => 4GB
- 4KB block size

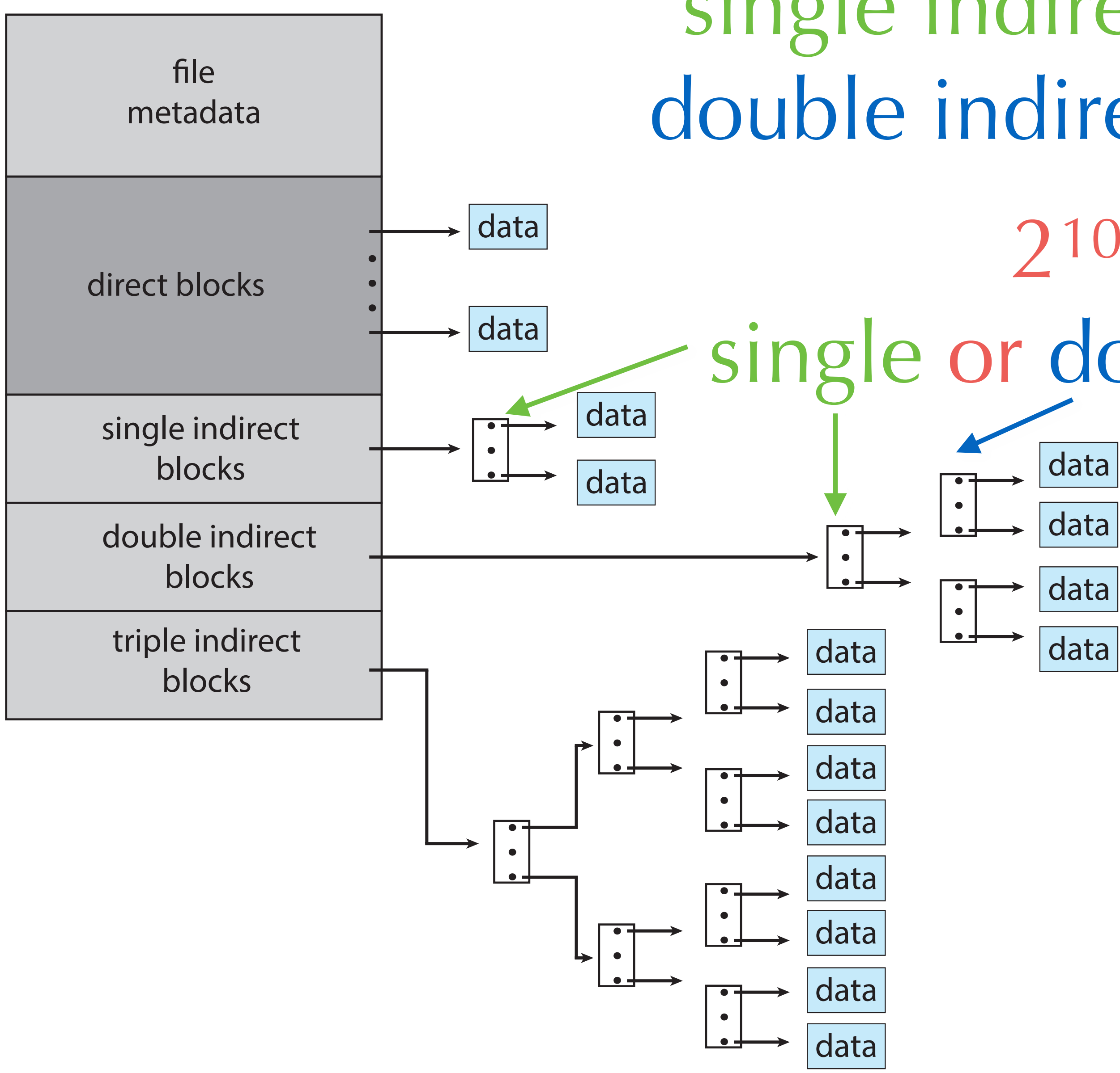
direct size: 12 x 4KB = 48 KB

single indirect size: 2^{10} x 4KB = 4MB

double indirect: 2^{10} x 2^{10} x 4KB = 4GB

12 direct entries

2^{10} entries for single or double indirect table



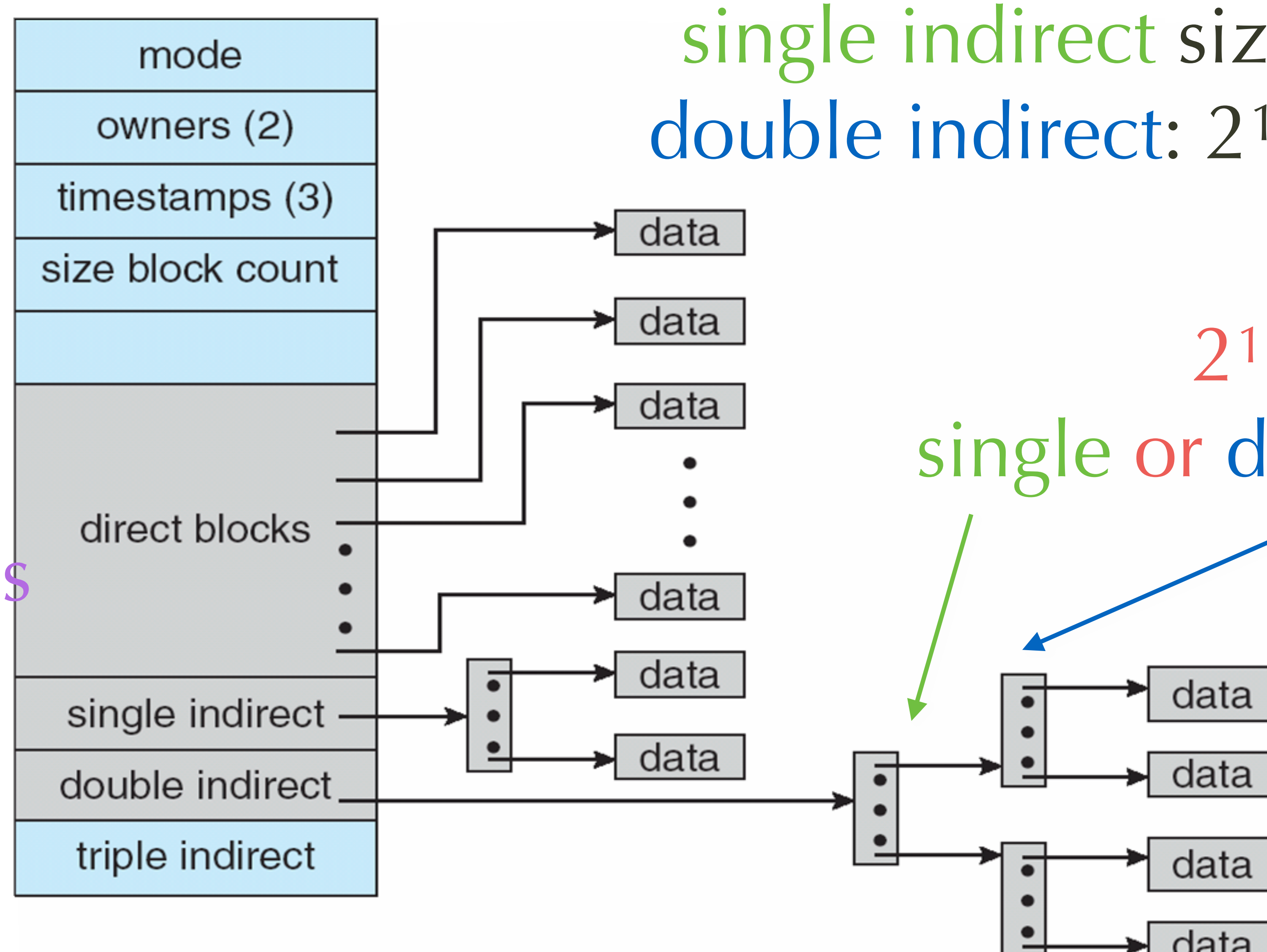
Combined Scheme: UNIX inodes

- File pointer: 4 bytes (32 bits) => 4GB
- 4KB block size

direct size: 12 x 4KB = 48 KB

single indirect size: 2^{10} x 4KB = 4MB

double indirect: 2^{10} x 2^{10} x 4KB = 4GB



12 entries

2^{10} entries for single or double indirect table

Performance

- Contiguous
  - great for sequential and random access (aka "direct" access)
- Linked
  - good for sequential, not random access
- OS approaches
  - may be hybrid: contiguous for direct, linked for sequential
  - Declare access type at creation
    -> select either contiguous or linked, and OS will do conversion to the matching allocation.

Performance (cont'd)

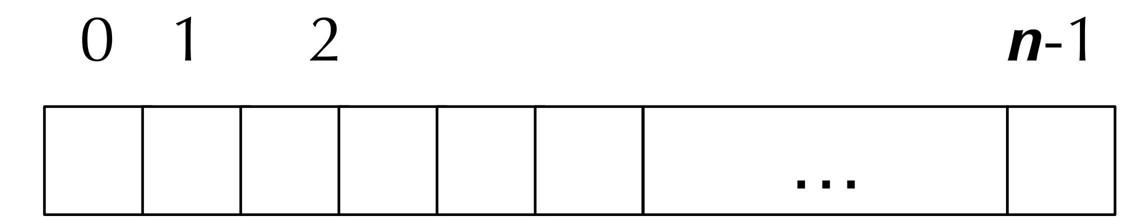
- Indexed more complex
  - Single block access could require 2 index block reads then data block read => caching helps
  - Clustering can help improve throughput, reduce CPU overhead
- Hybrid index and contiguous
  - contiguous for small files
  - switch to indexed allocation as file grows large

Free Space Management

Free-Space list

- Needed to track **available** blocks or clusters
  - (Using term "block" for simplicity)
- Options
  - Bit vector or bit map (n blocks)
  - Linked list (same as linked allocation)
  - Grouping (same as linked indexed allocation)
  - Counting (same as contiguous allocation)
- OS usually manage free space same way as file

Bit Vector



- Example:
  - block size = 4KB
  - disk size = 2^{40} bytes (1 TB)
  - n = $2^{40}/2^{12} = 2^{28}$ bits (or 32MB)
  - clusters of 4 blocks = 8MB of memory
- Pro
  - Easy to get contiguous files
- Con
  - bitmap must be cached for performance
  - 1TB disk requires 32MB bitmap

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

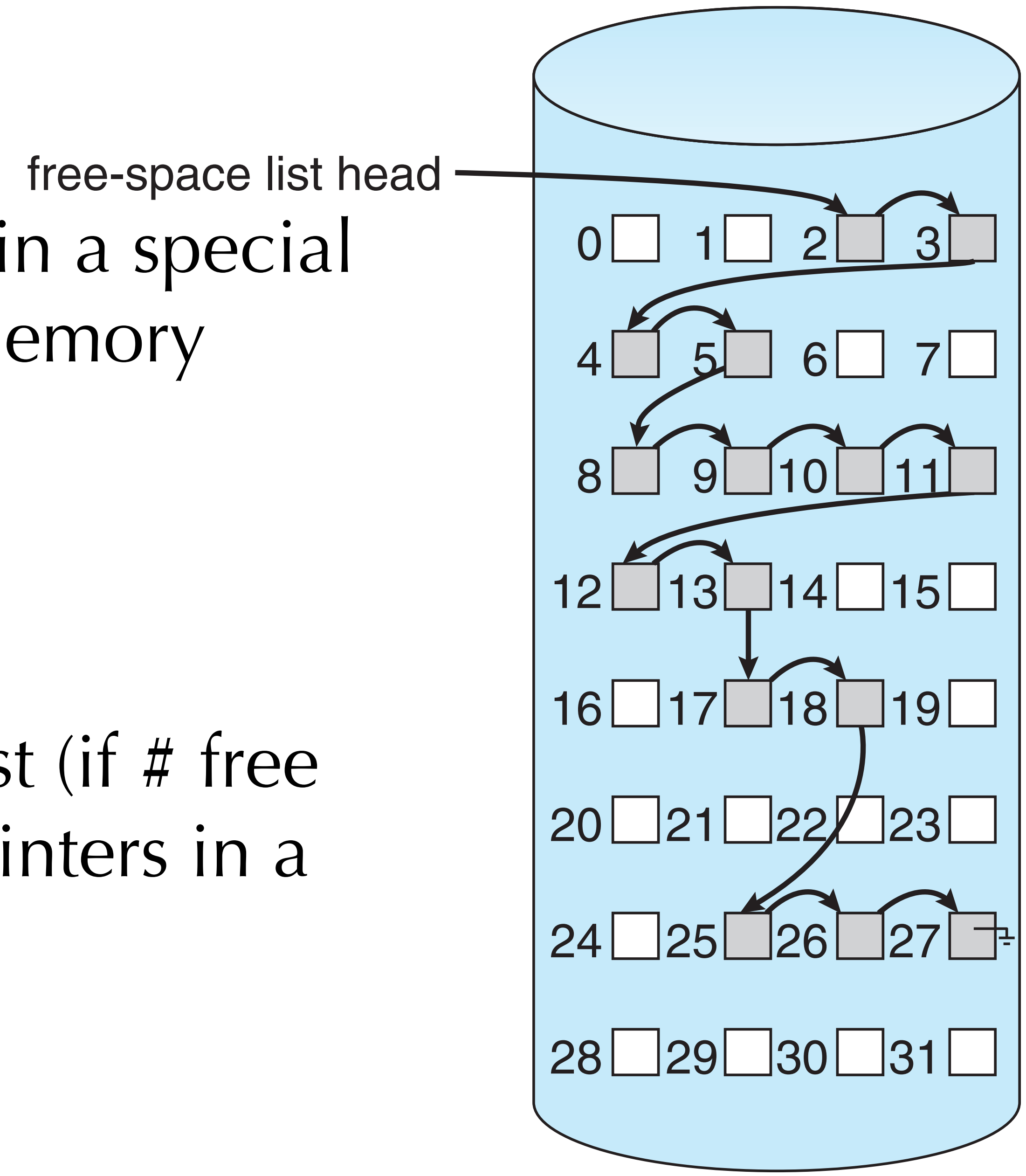
Block number calculation

(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

Linked Free Space List on Disk

- Same as linked allocation
  - keep the first free block pointer in a special location on disk and cache in memory
- Pro
  - No waste of space
  - No need to traverse the entire list (if # free blocks recorded); put all link pointers in a table (FAT)
- Con:
  - Cannot get contiguous space easily



Grouping and Counting in Linked free list

- Grouping (same as linked-index allocation)
  - Modify linked list to store in the first block:
    - address of next (n-1) free blocks in first free block,
    - a pointer to next block that contains free-block-pointers (like this one)
- Counting (same as contiguous allocation)
  - Keep address of first free block and count of following free blocks
  - Free space list then has entries containing addresses and counts

TRIM and Unallocate

- New mechanisms for informing storage devices of pages that can be erased
  - Storage device optimizes erasure by scheduling its own erase operation
  - especially important for NVM
- commands
  - TRIM - for ATA drives
  - Unallocate - for NVMe-based

Efficiency: depends on

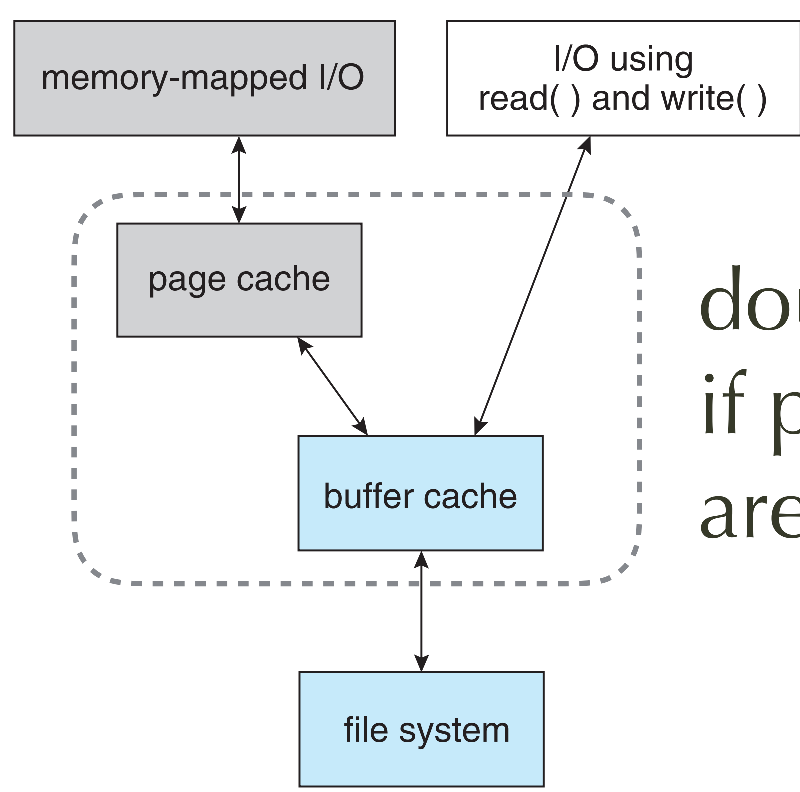
- Disk allocation and directory algorithms
- Types of data kept in file's directory entry
  - e.g., last access date, last modified date
- Metadata structures
  - Pre-allocation or
  - as-needed allocation
- Fixed-size or varying-size data structures
  - FAT-16 => max size of partition = 32 MB. (PC XT HD 10MB)
  - FAT-32: 4 GB per file limit, 16TB max partition size

Performance: depends on

- Whether data and metadata are close
- Buffer cache
  - separate section of main memory for frequently used blocks
  - No buffering / caching => writes must hit disk before ack
- Buffer cache + Page cache: unified or separate?
- Optimizations
  - Synchronous or asynchronous writes
  - for sequential access

Buffer cache vs Page cache

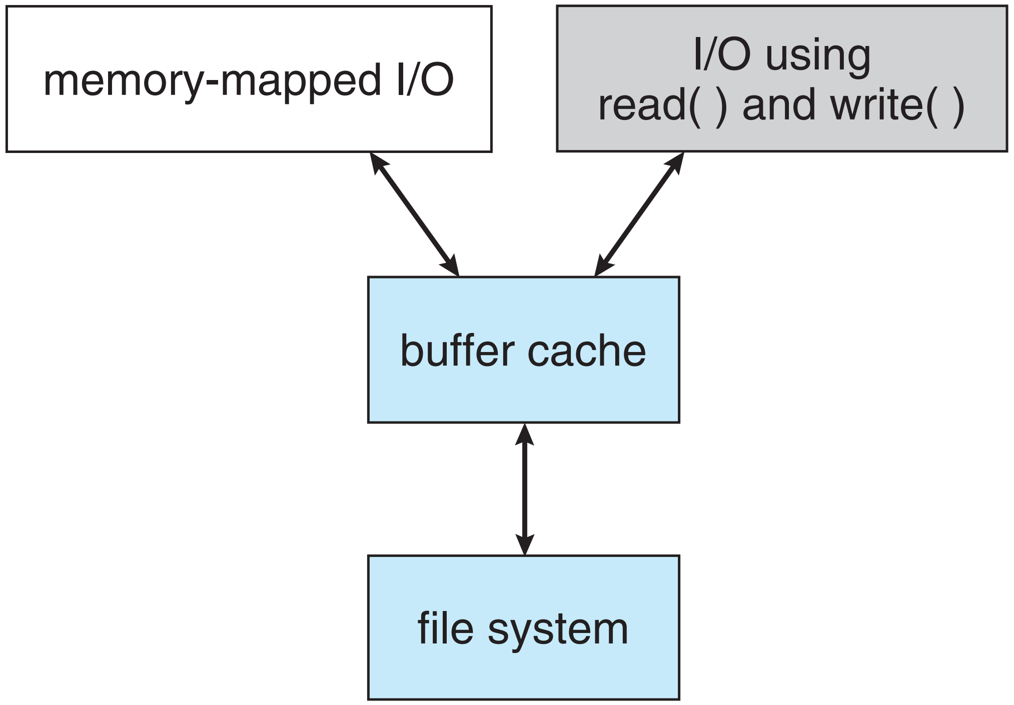
- Buffer Cache
  - caches disk blocks for file system
  - Works with read() and write() calls
- Page Cache
  - caches file data as pages
  - Memory-mapped I/O goes to page cache



double caching
if page& buffer caches
are separate

Unified Buffer Cache

- Uses the same page cache to cache both
  - memory-mapped pages and
  - ordinary file system I/O
- i.e., Virtual Memory system manages file-system data
- Purpose
  - avoid double caching
- Issues
  - which caches get priority?
  - what replacement algorithms to use?



Optimization with Page Cache for sequential access

- LRU is a bad idea for page replacement
  - most recently used unlikely to be used again soon
- Instead: use Free-behind
  - remove a page from buffer upon accessing next page
- Read-ahead
  - idea of prefetch - next page is likely to be needed soon, want to get started early to save latency

I/O Synchrony and impact on performance

- Asynchronous writes
  - more common, buffer-able, faster
  - small writes may appear fast => actual I/O much slower
  - larger writes might not be faster if out of buffer
- Synchronous or asynchronous Reads
  - frequently slower than asynchronous writes
  - only prefetching may help, not synchrony

Consistency among multiple on-disk data structures

- On-disk data structures
  - directory structure, free-block pointers, free FCB pointers
- Cause of inconsistency:
  - system crash before changes get fully flushed to disk
  - pull USB drive without doing "Safely remove"
-

Recovery

- Consistency checking (e.g., unix `fsck`)
  - compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Back up data from disk to A storage device
  - magnetic tape, other magnetic disk, optical
  - Recover lost file by restoring data from backup

Log-Structured File Systems (Journaling)

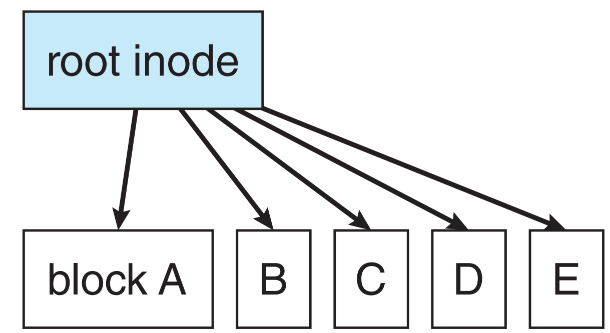
- Journaling to a log (separate disk or section on disk)
  - FS records each metadata update to the file system as a transaction
  - A transaction is committed once it is written to the log
- Transactions in the log are asynchronous to the file system
  - The file system may not have been updated yet
  - When FS structures are modified, the transaction is removed from log
  - If FS crashes => remaining transactions in log must still be performed
- Advantages
  - Faster recovery from crash
  - removes chance of inconsistency of metadata

Example: WAFL File System

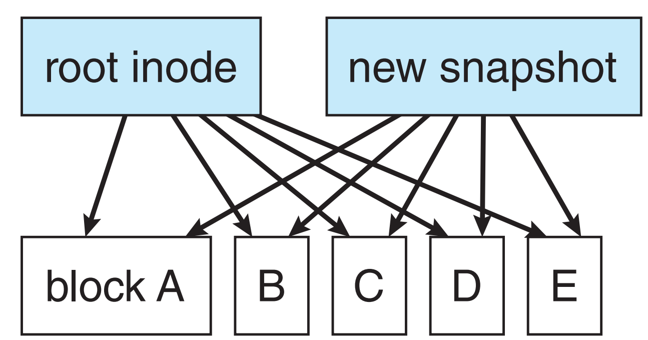
- Used on Network Appliance “Filers”
  - distributed file system appliances
  - WAFL = "Write-Anywhere File Layout"
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Key feature: Snapshots

Snapshots in WAFL

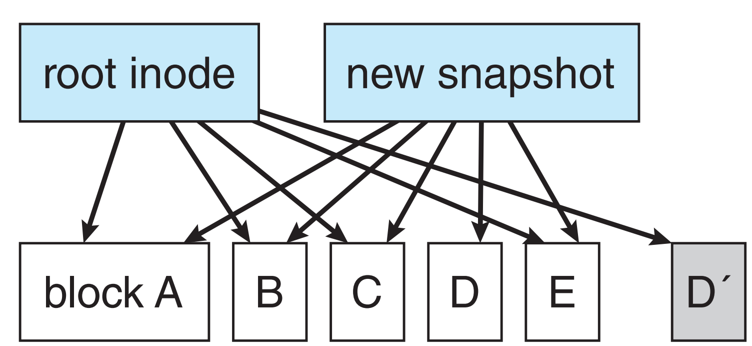
- Snapshot
  - keep old inode
  - make new inode that points to new
- Update to snapshot
  - don't write over existing block - write to new and point
  - useful for versioning
  - Does not require copy on write - automatic!



(a) Before a snapshot.



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



(c) After block D has changed to D´.