

# Operating Systems

## CS3500 – CH-14

**Prof. Sukhendu Das Deptt. of Computer Science and  
Engg., IIT Madras, Chennai – 600036.**

Email: [sdas@cse.iitm.ac.in](mailto:sdas@cse.iitm.ac.in)

URL: <http://www.cse.iitm.ac.in/~vplab/os.html>

OCT. – 2022.

# File System Implementation

## Outline :

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

File system provides the mechanism for online storage and access to file contents, including data and programs. In CH-13: File structure, attributes, types, operations, Access methods, protection etc. were studied;

In Ch-14: - Directory; Allocation, Free-space Mgmt, Efficiency, Recovery.

so - What vs How ?

# File-System Structure

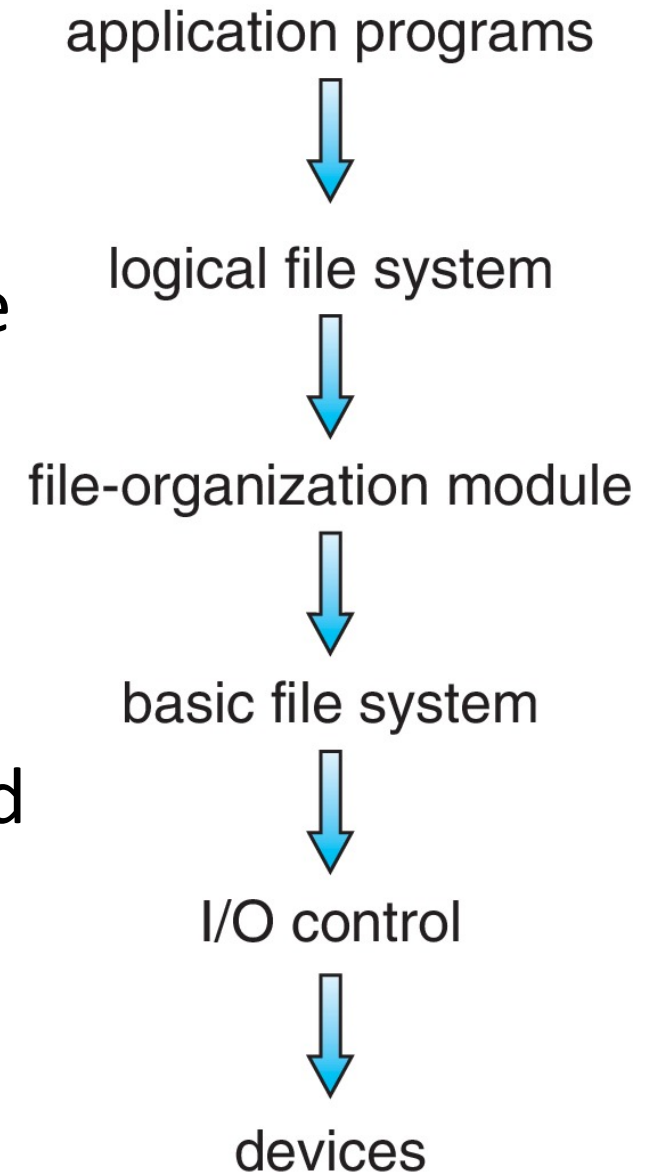
- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located & retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block (FCB)** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers

# 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



**File systems** provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily.

The **logical file system** manages *metadata* information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks. A file control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection.

**File-organization** module knows about files and their logical blocks. Each file's logical blocks are numbered from 0 (or 1) through N. The file organization module also includes the free-space manager.

**Basic file system** (called the "block I/O subsystem" in Linux) needs only to issue generic commands to the appropriate device driver to read and write blocks on the storage device. It issues commands to the drive based on logical block addresses. It is also concerned with I/O request scheduling.

The **I/O control level** consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. Its input consists of high level commands, such as "retrieve block 123." Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the

# File System Layers (Cont.)

- 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 130 types, with **extended file system** ext3 and ext4 leading; plus distributed file systems, etc.)
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

# File-System Operations

- We have system calls at the API level, but how do we implement their functions? → On-disk and in-memory 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 (UFS – Boot block; NTFS – partition boot sector)
- **Volume control block (UFS - superblock, NTFS - master file table)** contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table



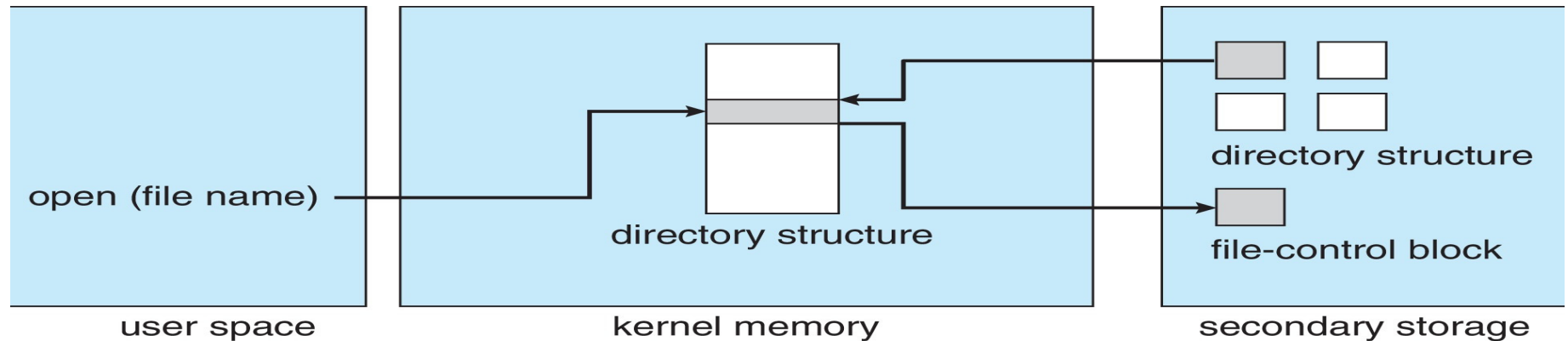
# File Control Block (FCB)

- OS maintains **FCB** per file, which contains many details about the file
  - Typically, inode number, permissions, size, dates
  - Example

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

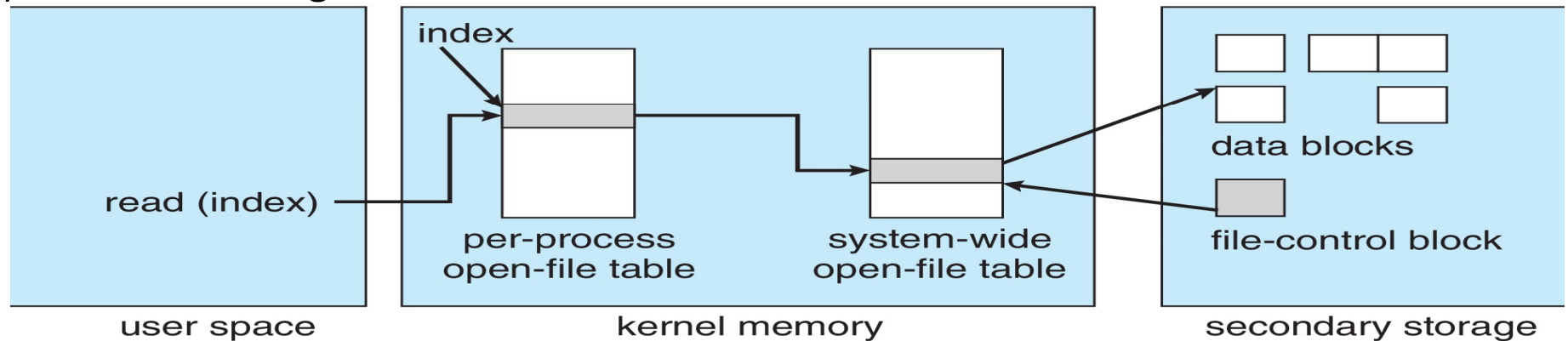
# In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types
- **System-wide open-file table** contains a copy of the FCB of each file and other info
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info (File descriptor or File Handle)



(a)

- Figure (a) refers to opening a file
- Figure (b) refers to reading a file



(b)

# 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 Method

- An allocation method refers to how disk blocks are allocated for files:
  - Contiguous
  - Linked
  - File Allocation Table (FAT)

## Contiguous Allocation Method

An allocation method refers to how disk blocks are allocated for files:

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 on the disk for a file,

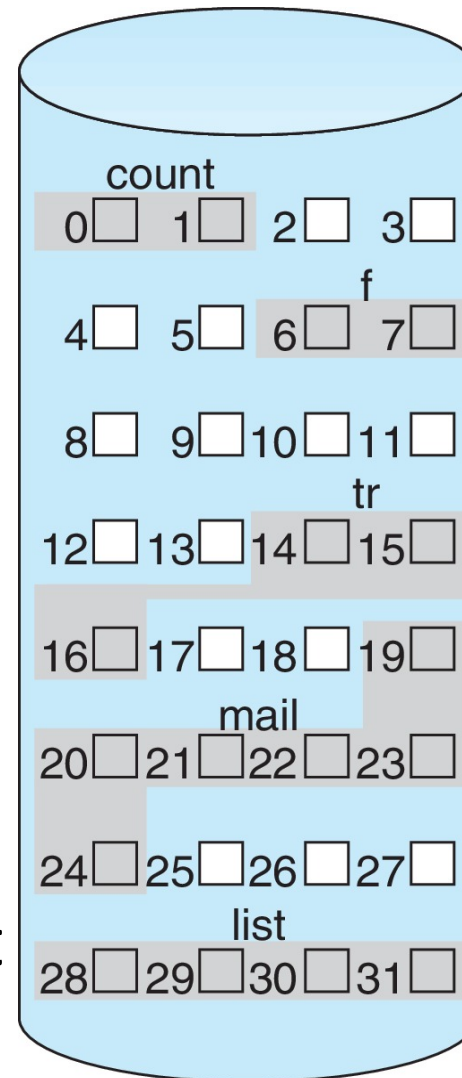
Knowing file size,

External fragmentation, need for **compaction off-line (downtime)** or **on-line**; effectively compacts all free space into one contiguous space, solving the fragmentation problem

# Contiguous Allocation (Cont.)

- Mapping from logical to physical  
(block size = 512 bytes)

LA/512      Q  
                  /\  
                  R



directory

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

- Block to be accessed =  
starting address + Q
- Displacement into block = R
- If the file is n blocks long  
and starts at location b, then it  
occupies blocks  
b, b + 1, b + 2, ..., b + n - 1.

# 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

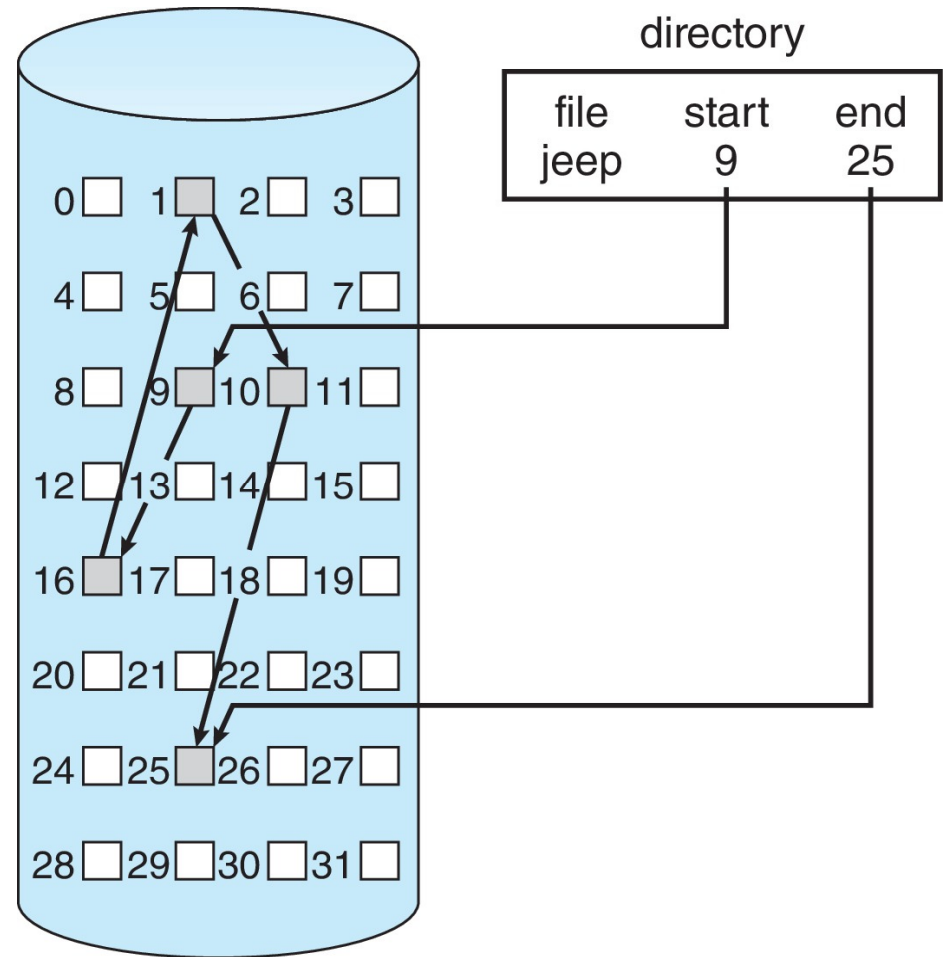
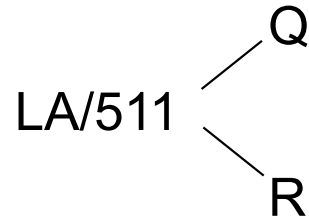
A contiguous chunk of space is allocated initially. Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an ***extent***, is added

# Linked Allocation

- Each file is a linked list of blocks
- File ends at nul 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 Example

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- Scheme
- Mapping
- Block to be accessed is the  $Q^{\text{th}}$  block in the linked chain of blocks representing the file.
- Displacement into block =  $R + 1$



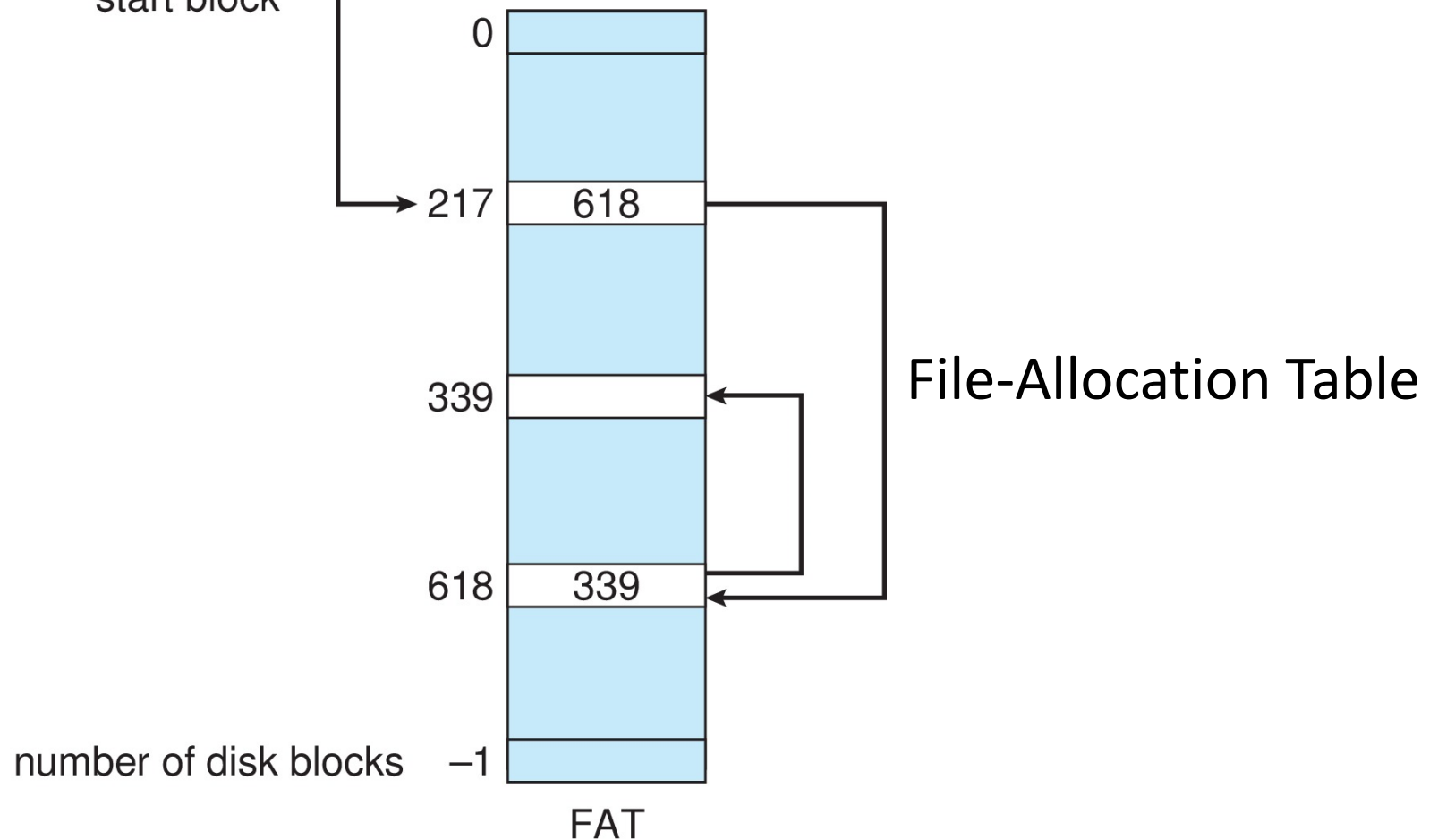
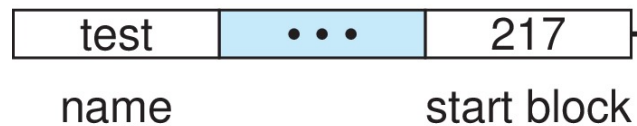
If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space. Soln: collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks



# FAT Allocation Method (Altn to Linked Allocn.)

- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

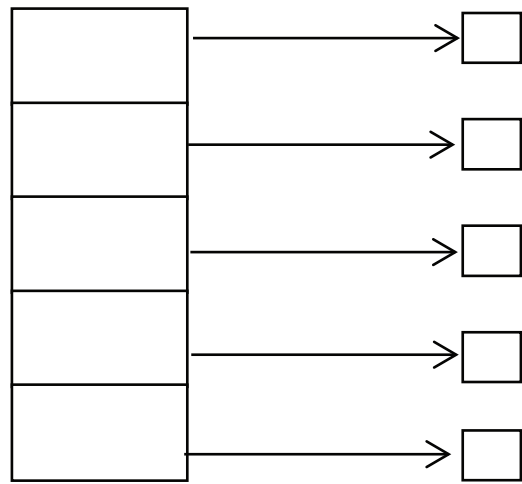
directory entry



# Indexed Allocation Method

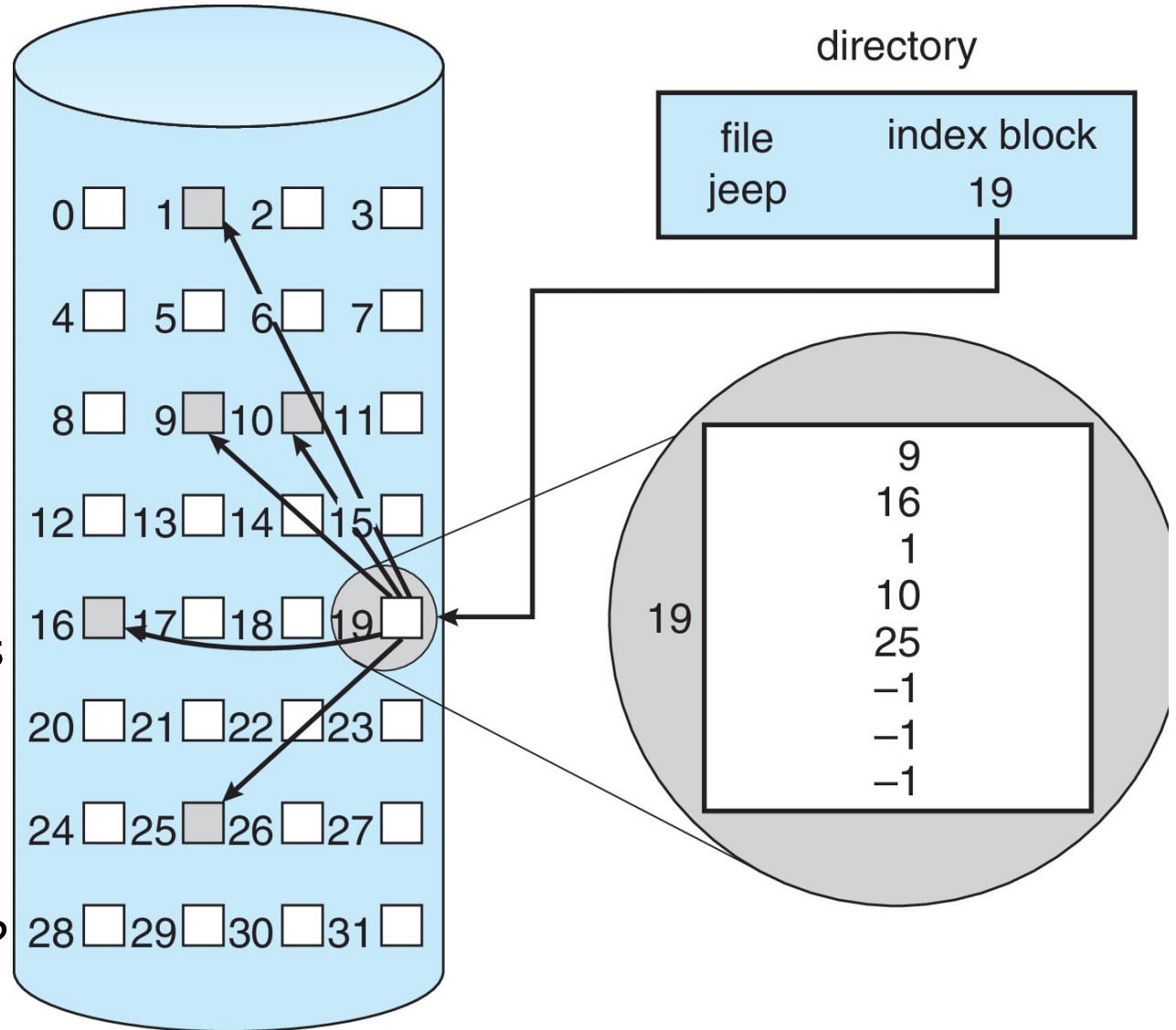
- Each file has its own **index block(s)** of pointers to its data blocks
- Logical view

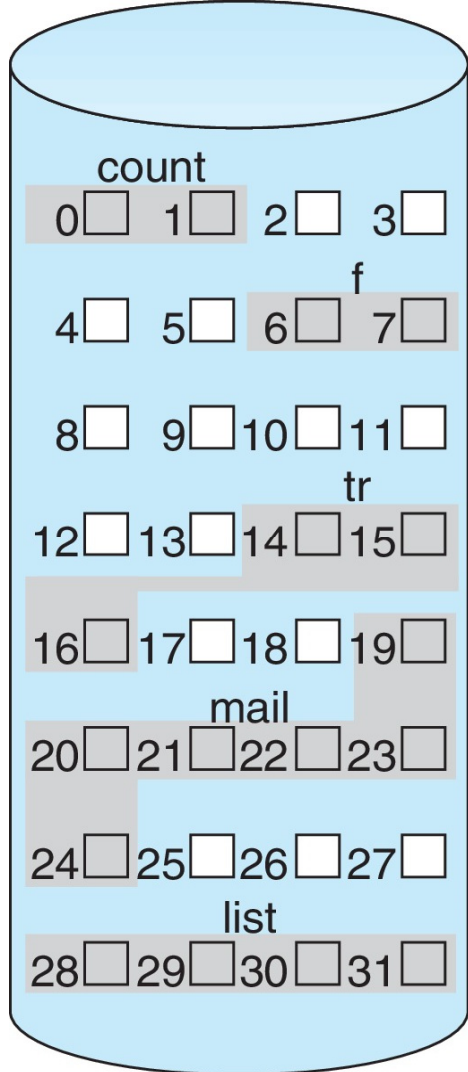
Example of Indexed Allocation



index table

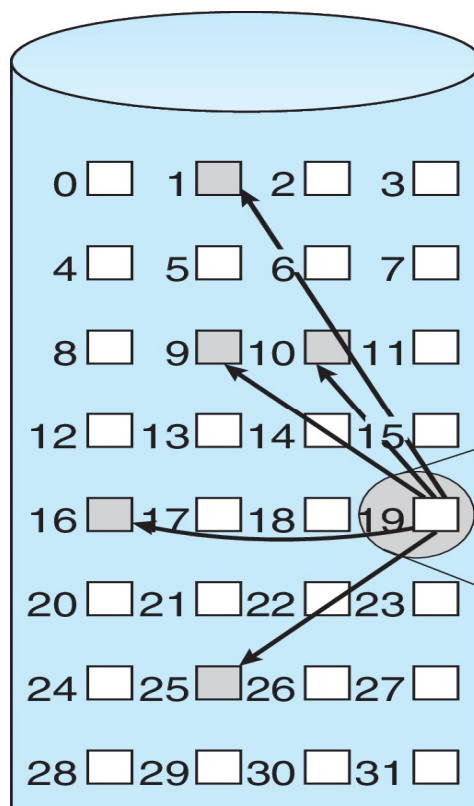
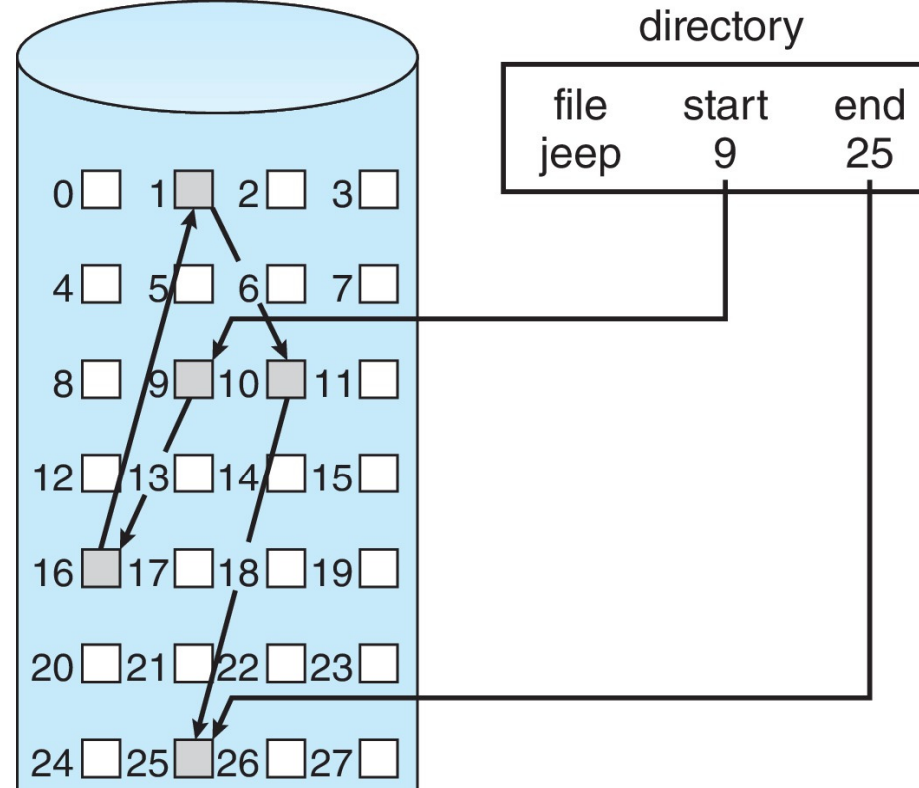
Uses paging scheme; supports direct access, no wasted space; no external fragmentation; larger pointer overhead than linked Allocn;  
How large in an index block?  
Several schemes – *linked, multi-level and combined.*





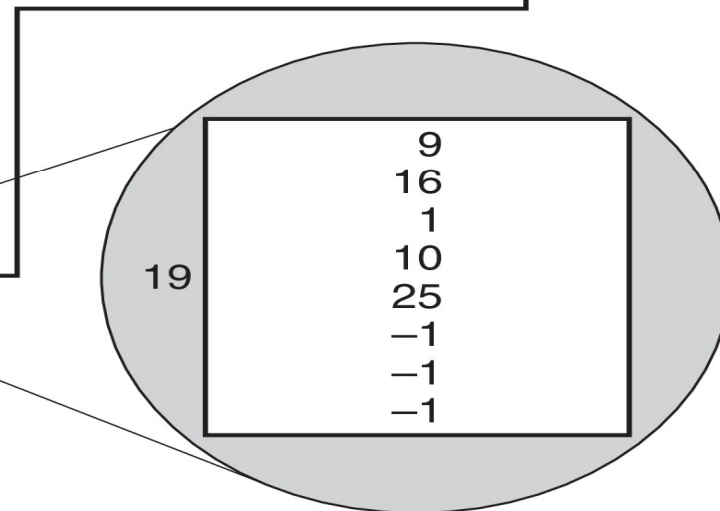
directory

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



directory

file	index block
jeep	19

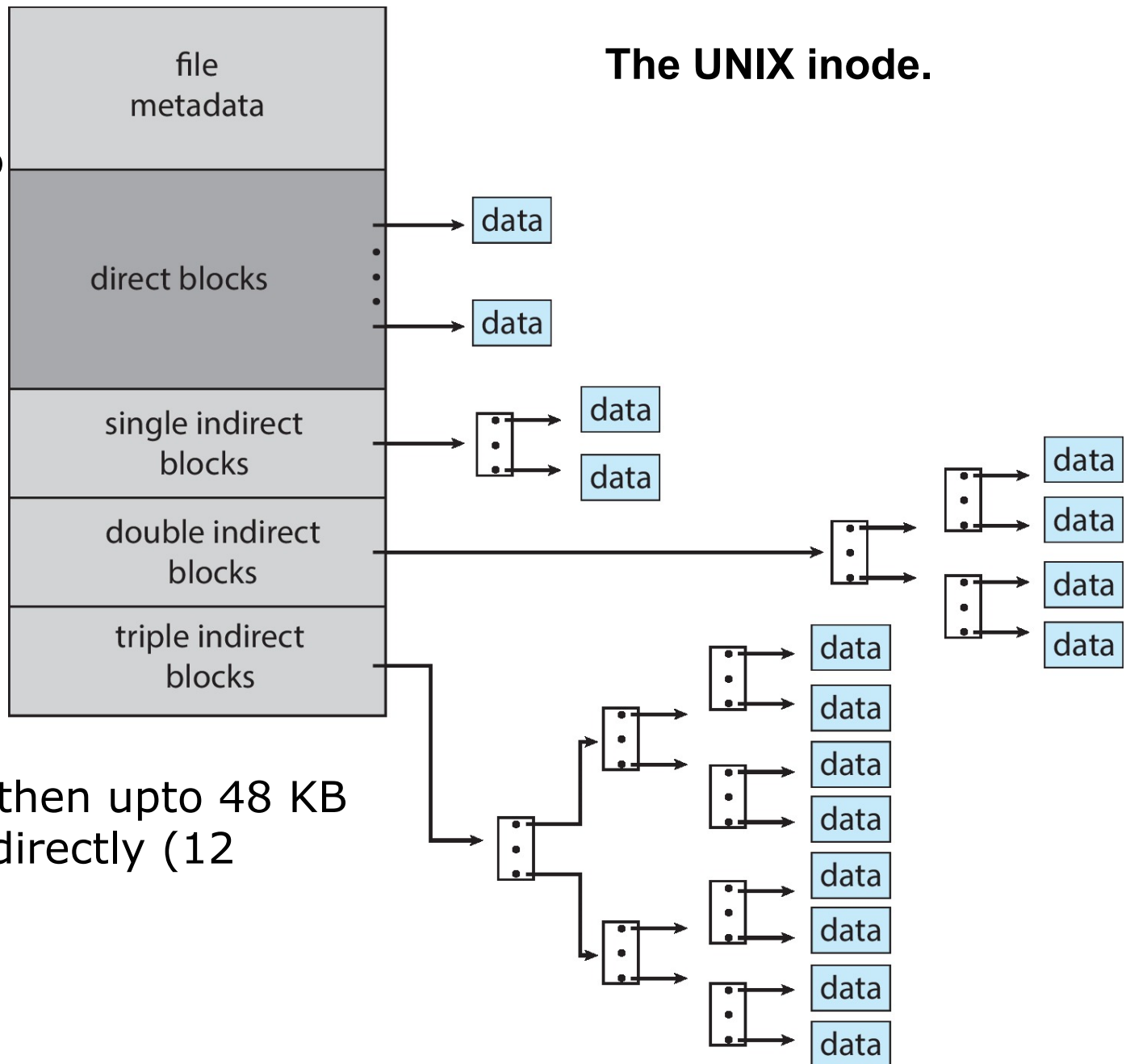


# Combined Scheme : UNIX UFS

- 4K bytes per block, 32-bit addresses – 4GB
- More index blocks than can be addressed with 32-bit file pointer (or 64/128-bit too)

If the block size is 4 KB, then upto 48 KB of data can be accessed directly (12 pointers for direct).

Rest used indirectly



# Performance

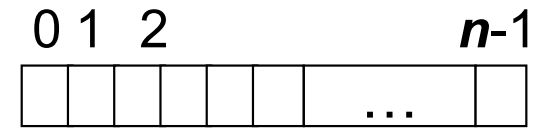
- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation
  - Select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read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

Given the disparity between CPU speed and disk speed, it is not unreasonable to add thousands of extra instructions to the operating system to save just a few disk-head movements; optimization algos used.

- For NVM, no disk head - so different algorithms and optimizations needed
  - Using old algorithm uses many CPU cycles trying to avoid non-existent head movement
  - Goal is to reduce CPU cycles and overall path needed for 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

Bit map requires extra space

Example:

CPU's have instructions to return offset within word of first “1” bit



$\text{block size} = 4\text{KB} = 2^{12} \text{ bytes}$

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

$n =$    $\text{MB}$

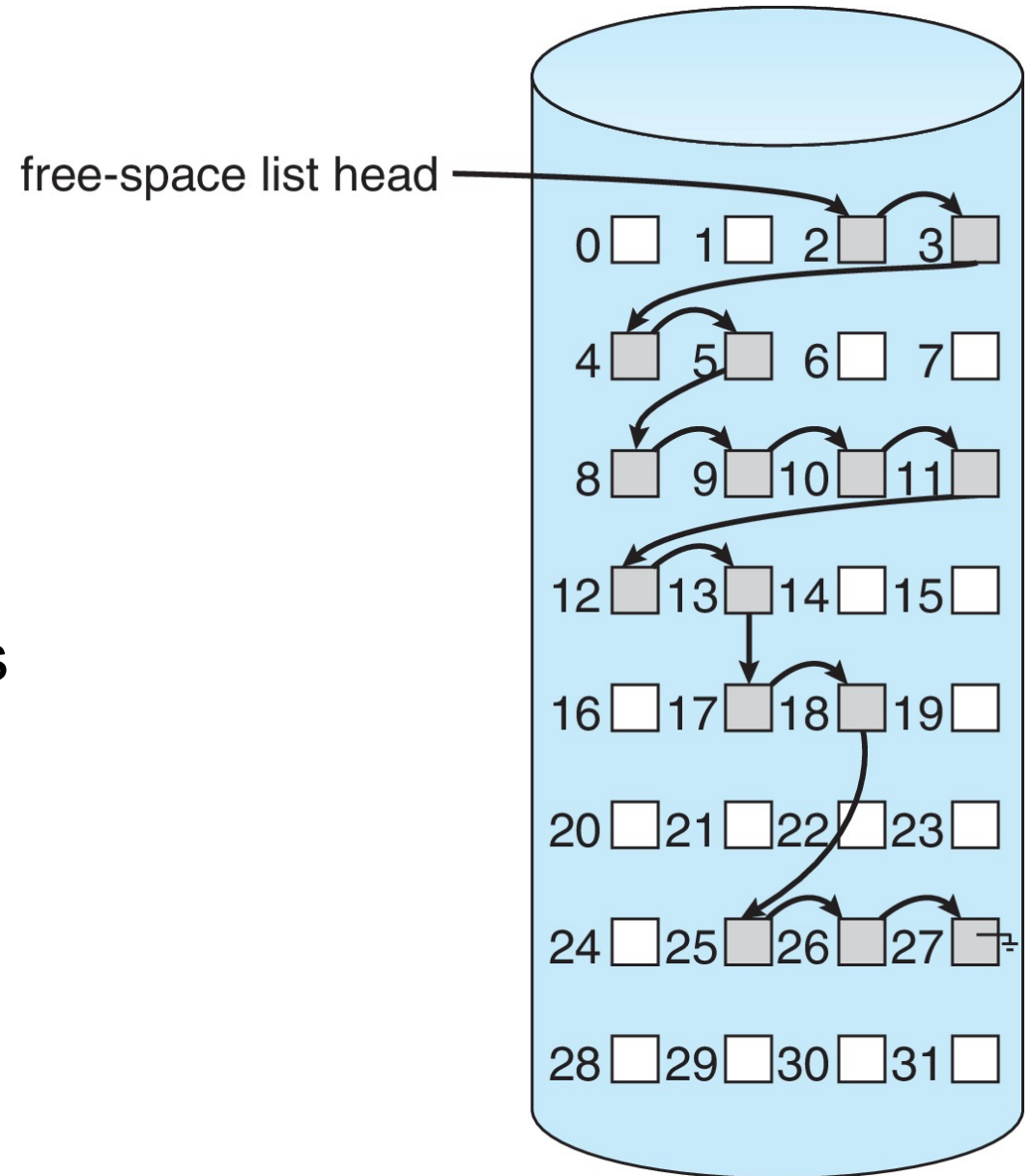
if clu   $\text{B of memory}$

 A 1.3-GB disk with 512-byte blocks would need a bitmap of over  KB to track its free blocks,

although clustering the blocks in groups of four reduces this number to around 83 KB per disk. A  4-TB disk with 8-KB blocks would require  MB) to store its bitmap.

# Linked Free Space List on Disk

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste. Linked Free Space List on Disk 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)
    - *Heard of run-length coding ?*
- 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



# Free-Space Management (Cont.)

- Space Maps
  - Used in **ZFS** (Oracle, Solaris)
  - Consider meta-data I/O on very large file systems
    - Full data structures like bit maps cannot fit in memory → thousands of I/Os; scattered blocks totaling GB
- Divides device space into **metaslab** units and manages metaslabs
  - Given volume can contain hundreds of metaslabs
- Each metaslab has associated space map
  - Uses counting algorithm
- But records to log file (log-structured file-system techniques) rather than file system
  - Log of all block activity, in time order, in counting format
- Metaslab activity → load space map into memory in balanced-tree structure, indexed by offset
  - Replay log into that structure (*the log plus the balanced tree is the free list*)
  - Combine contiguous free blocks into single entry (flush)

### Log Spacemaps

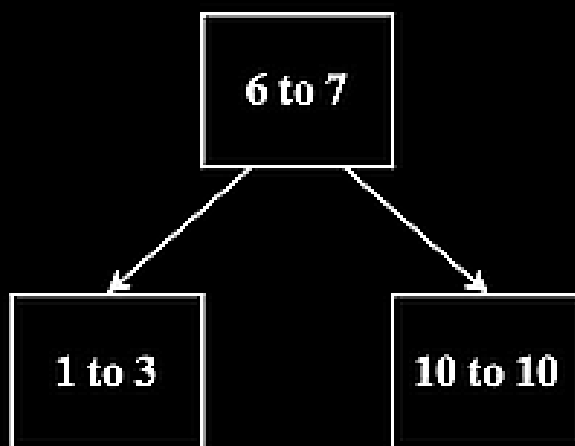
TXG	Blocks	Metaslabs flushed
10	2	1
11	6	2
12	2	2
13	4	2
14	6	10
15	4	2



### Summary

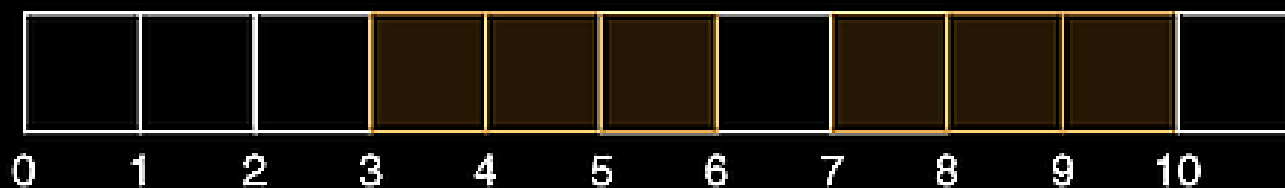
TXGs	Blocks	Metaslabs flushed
10 - 12	10	5
13 - 15	14	14

### Range Tree (AVL) In-memory



### Space Map On-disk

<b>TXG - 10</b>
ALLOCATE [1 , 6)
ALLOCATE [7 , 10)
<b>TXG - 11</b>
ALLOCATE [10 , 10]
FREE [1 , 3)
FREE [6 , 7)
FREE [10 , 10]



# TRIMing Unused Blocks

- HDDs overwrite in place so need only free list
- Blocks not treated specially when freed
  - Keeps its data but without any file pointers to it, until overwritten
- Storage devices not allowing overwrite (like NVM) suffer badly with this same algorithm
  - Must be erased before written, erases made in large chunks (blocks, composed of pages) and are slow
  - TRIM is a newer mechanism for the (ATA-based, EIDE or PATA) file system to inform the NVM storage device that a page (or block) is free
    - Can be garbage collected or if block is free, now block can be erased

# 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 (keep a file's data blocks near that file's inode block to reduce seek time)
  - 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 (also *page cache, unified virtual memory*)
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before proceeding (acknowledgement)
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** (removes a page from the buffer as soon as the next page is requested. The previous pages are not likely to be used again and waste buffer space) and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes (why?)

# 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

*memory mapping* a file, allows a part of the virtual address space to be logically associated with the file - mapping a disk block to a page (or pages) in memory. Simplifies and speeds up file access and usage. When file is closed, all the memory-mapped data are written back to the file on secondary storage and removed from the virtual memory of the process.

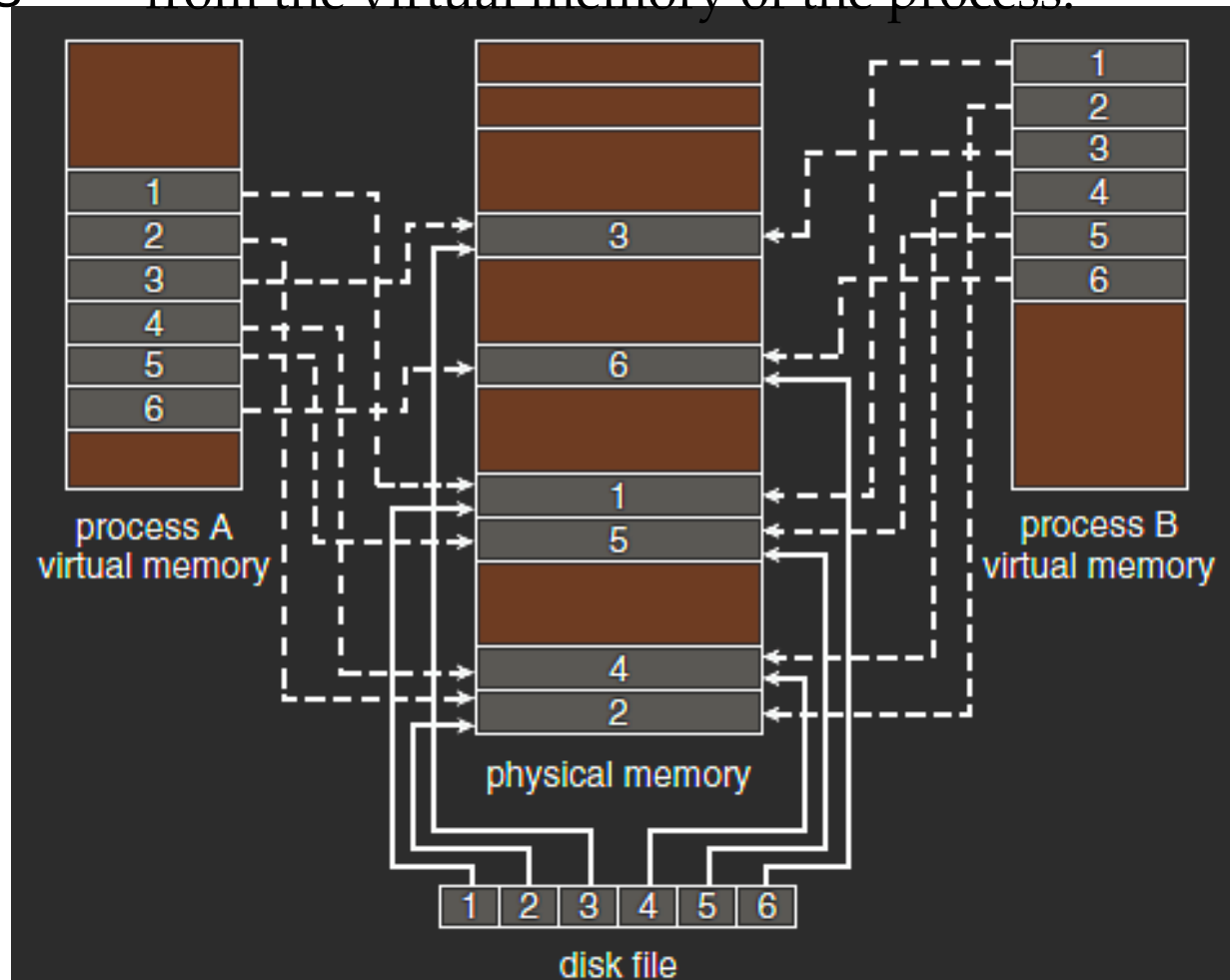
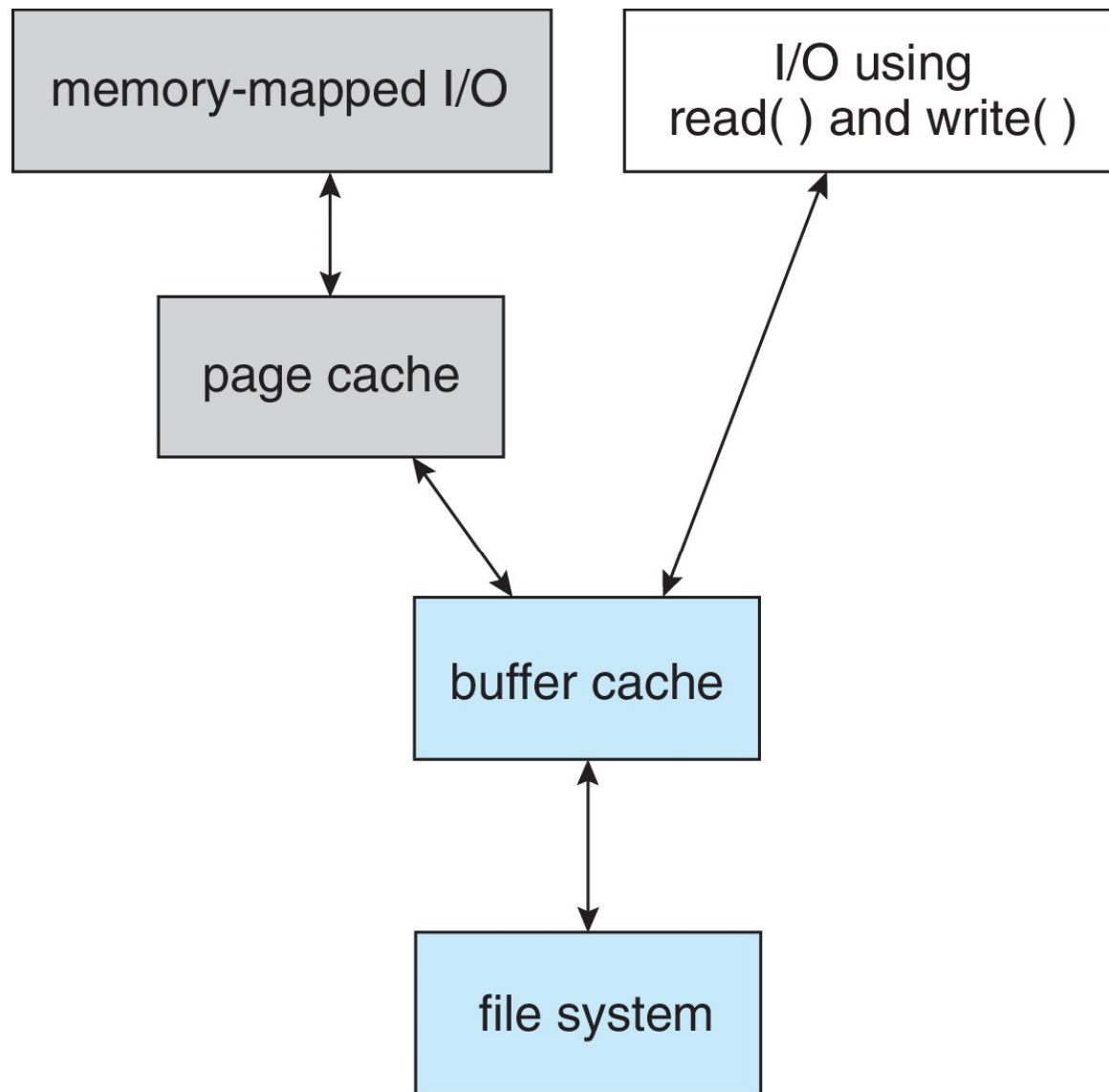


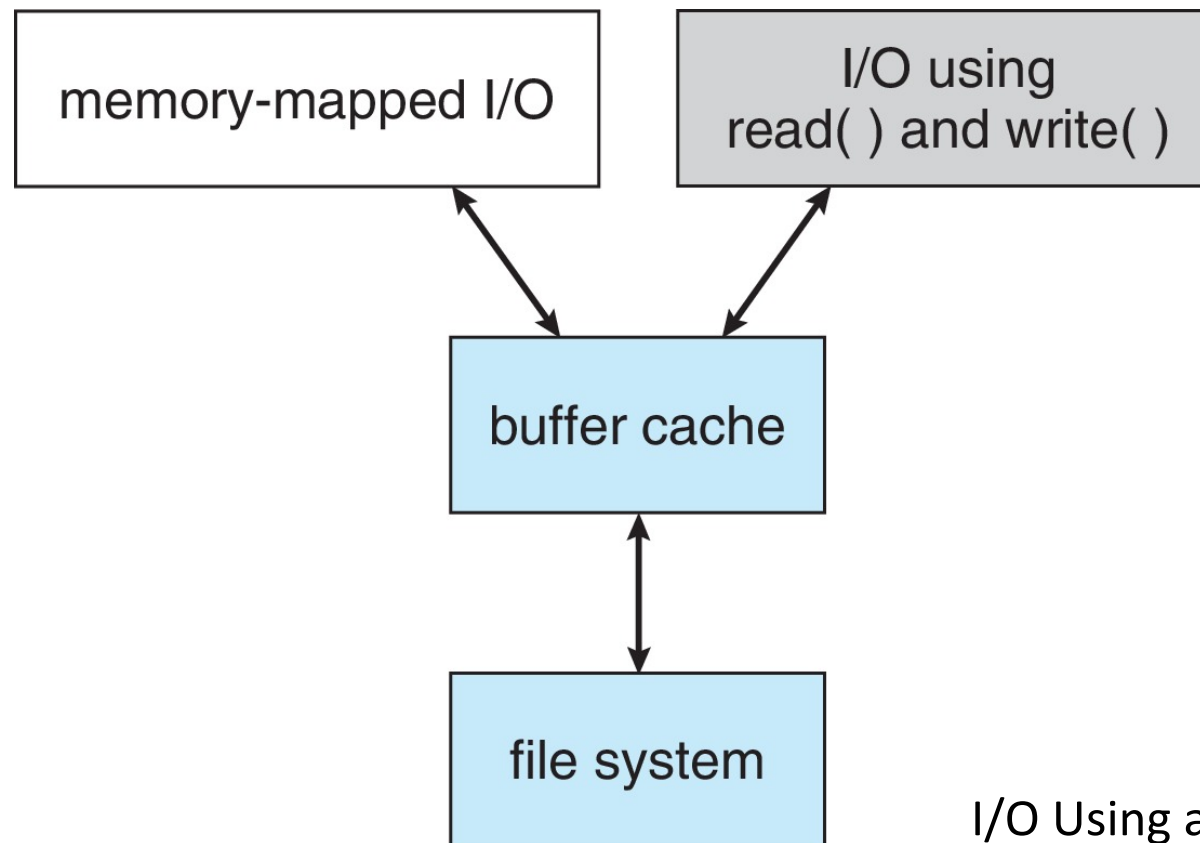
Figure 13.13 Memory-mapped files.

# 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 (process vs file) get priority, and what replacement algorithms to use?



I/O Using a Unified Buffer Cache

# Recovery

- Files and directories are kept both in main memory and on the storage volume, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency.

## 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 (circular buffer)
  - 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

