File System Implementation

Chapter 12

Adapted from the slides given by Silberschatz, Galvin, and Gagne
In "Operating System Concepts", 10th edition
John Wiley and Sons, Inc 2018.

Dr.Siwar Rekik

Outline of this lecture

In this lecture, we will discuss the following:

- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management

Objectives

- To describe the details of implementing local file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

File System Implementation

- The implementation must
  - Keep track of the files and directories stored in the file system, i.e. which disk blocks/sectors store the file data.
  - Keep track of the free blocks/sectors in the disk.
- Structures used for implementing FS:

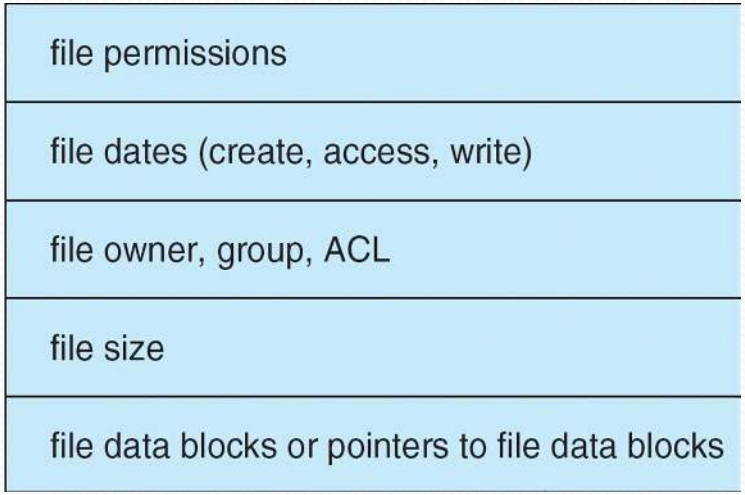
1. on-disk structure – fast,
2. in-memory structure

on-disk File-System Implementation

- There are various **on disk data structures** that are used to implement a file system. This structure may vary depending upon the OS. The *FS may contain info* about
- **Boot control block (Boot Sector)** 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 (partitions) details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- **Directory structure** is used to organize **all** the files - starting with the root directory
  - In UNIX, it includes inode numbers associated to file names.

On-disk File-System Implementation (Cont.)

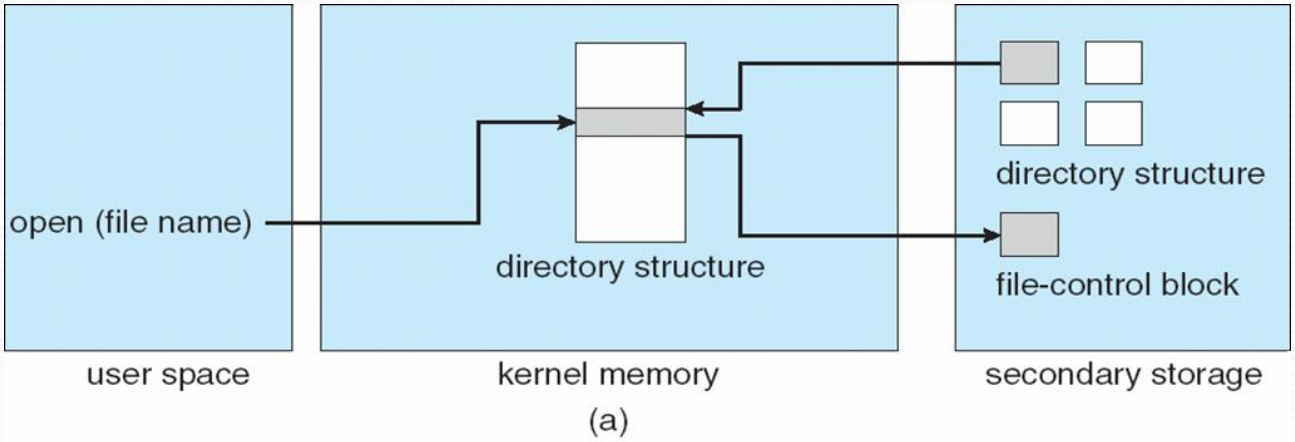
- Per-file **File Control Block** (**FCB/inodes in most unix systems**) contains many details about the file (i.e. name, attributes associated with the file, size, location, owner…)
  - A typical FCB is shown below:



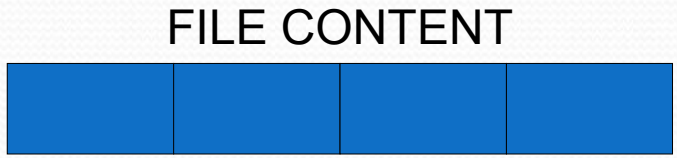
In-Memory File System Structures

- A **mount table** – contains information about each mounted volume- list of all the devices which are being mounted to the system.
- A **directory structure cache** – info about **recently** accessed directories by the CPU.
- A **system-wide open file table** contains a copy of the FCB of **all open file** in the system and other info.
- A **per-process open-file table** (1 process file info) contains a pointer to the appropriate entry in the system-wide open-file table and other info.
- Buffers that hold file-system blocks as they are being read and written to the disk.

In-Memory File System Structures



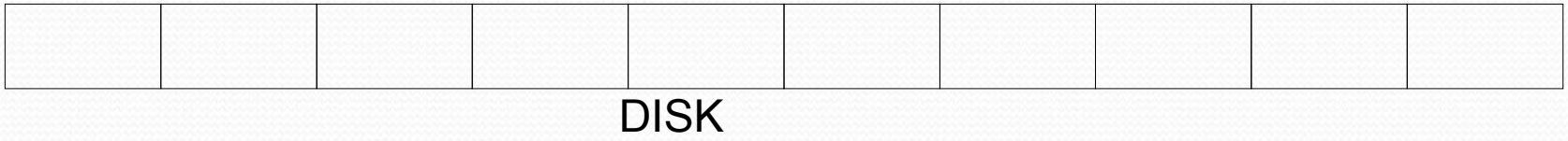
Allocation Methods



A file is viewed as a sequence of **logical** blocks (data blocks)

Mapping

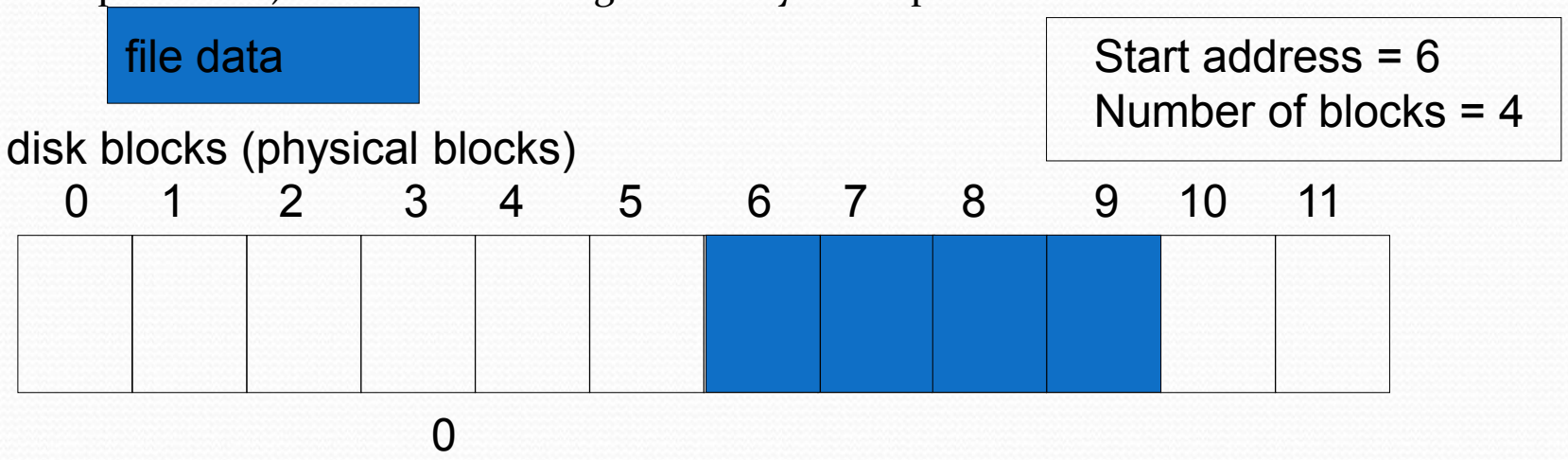
A disk is viewed as a sequence of **physical** blocks



Allocation Methods - Contiguous

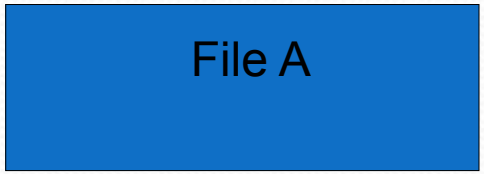
- An allocation method refers to how to allocate space to files so that the disk is utilized effectively and files can be accessed quickly:

1. **Contiguous allocation** – each file occupies set of contiguous blocks
   - Simple – only starting location (block #) and length (number of blocks) are required to find out the *disk data blocks of file*
   - Problems - Files cannot grow
   - finding space for new file, knowing file size (the size of an output file may be difficult to estimate), external fragmentation, need for
     - **compaction off-line** (**downtime,** normal system operation cannot be permitted) or **on-line** during normal system operation



Example

offset 0



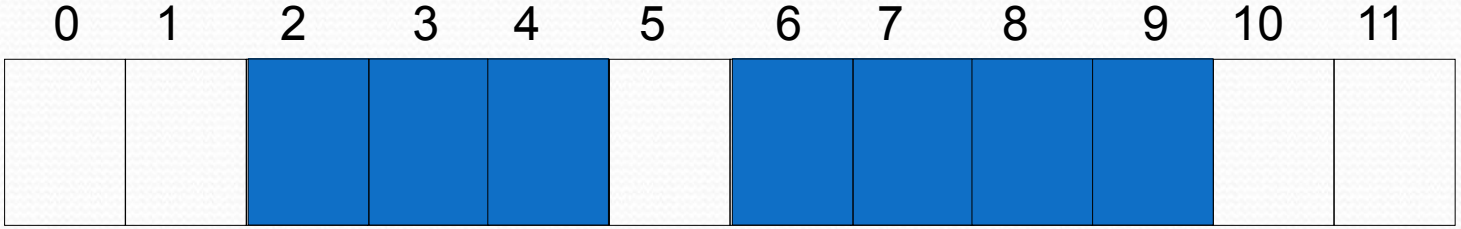
offset 0



File A: start=6, size_in_disk_blocks=4

File B: start=2, size_in_disk_blocks=3

disk blocks (physical blocks)



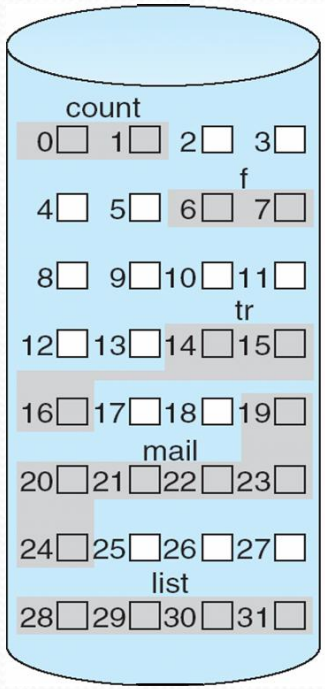
Contiguous Allocation

- LA: logical address into a file: file offset (i.e. address of a byte in file) (first byte has address o)
- Mapping from logical (file) address to physical (disk) address

LA/DiskBlockSize

Q = LA / DiskBlockSize

R=LA Mod DiskBlockSize



directory

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

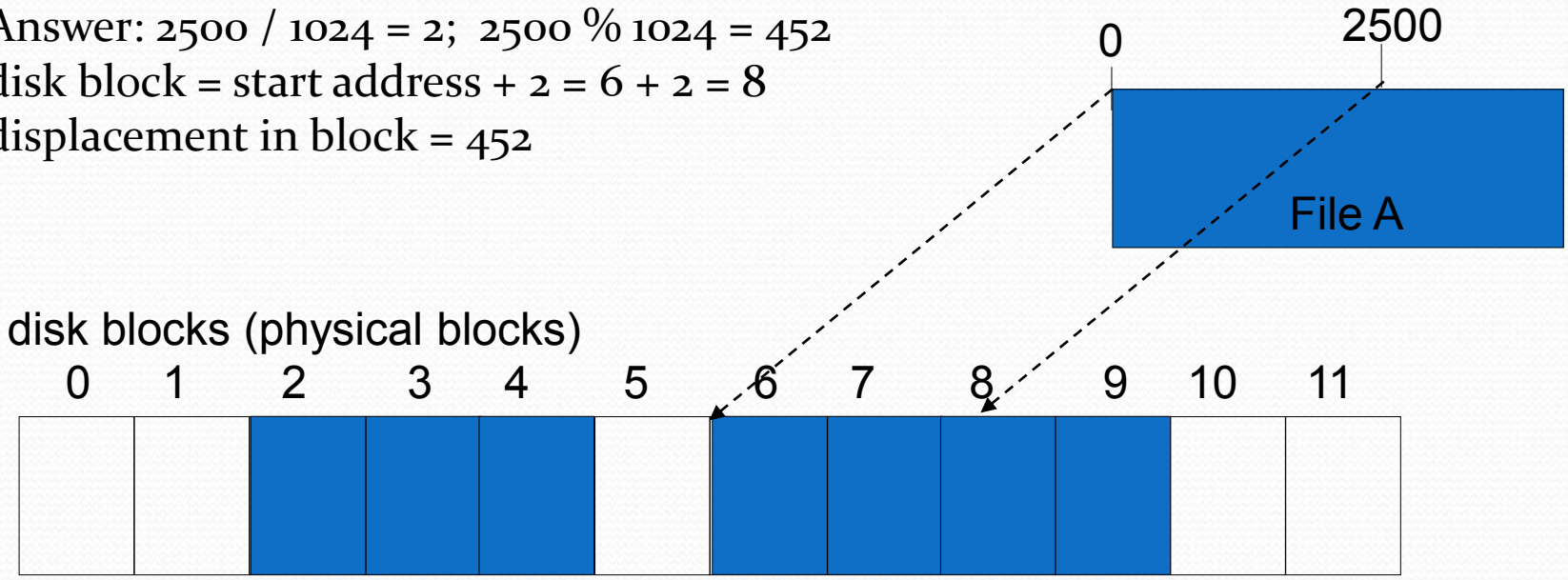
Block to be accessed = Q + starting disk block number (address)

Displacement into block = R

Example

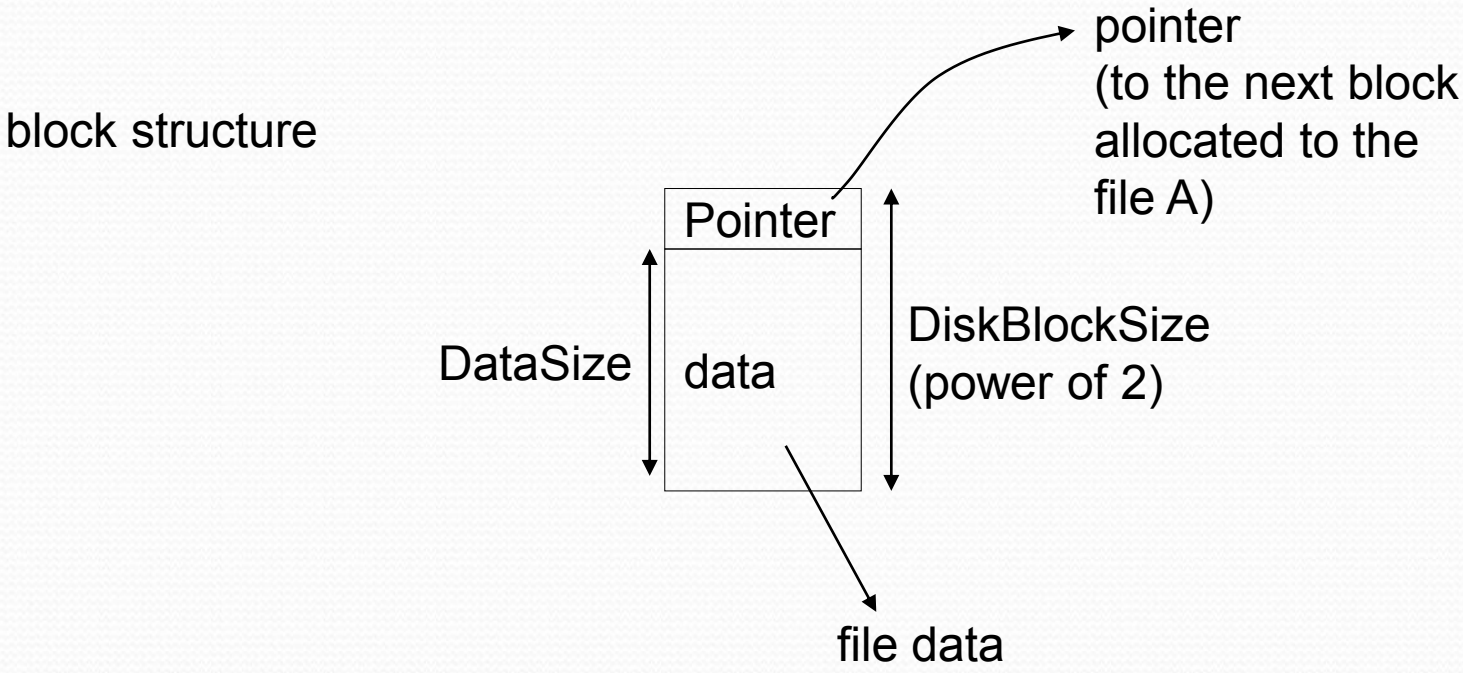
- Assume block size = 1024 bytes
- Which disk block contains the byte 0 of file A (LA = 0)? What is the displacement inside that block?
  - Answer : disk block = 6, displacement (disk block offset) = 0
- Which disk block contains the byte at LA (at file offset) 2500? In other words, where is LA 2500 mapped in disk?

Answer: 2500 / 1024 = 2; 2500 % 1024 = 452
disk block = start address + 2 = 6 + 2 = 8
displacement in block = 452



Allocation Methods – Linked

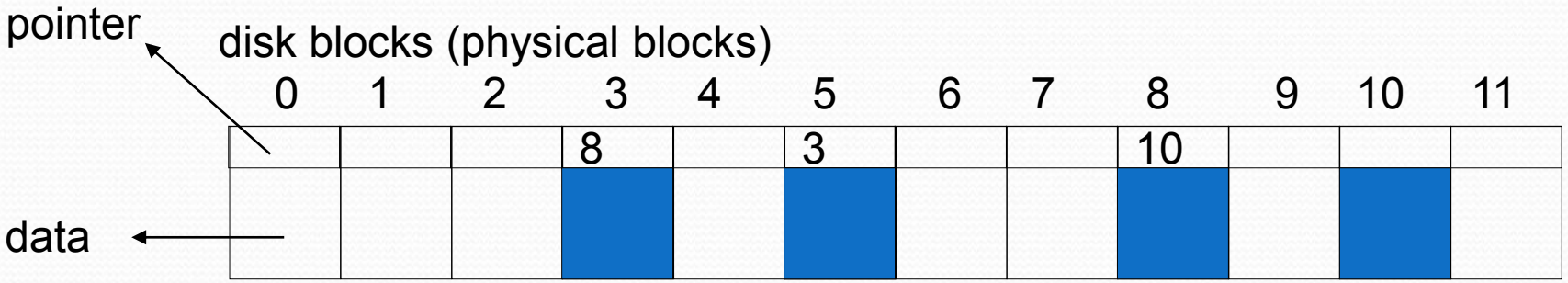
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



File data size in a disk block is no longer a power of 2

Linked Allocation

File A



Linked Allocation

2. Linked allocation – solves all problems of contiguous allocation.

- No external fragmentation
- No compaction, external fragmentation
- Each block contains pointer to next block
- File ends at nil pointer (the end-of-list pointer value)
- Free space management system called when new block needed
- Locating a block can take many I/Os and disk seeks
- can be used only for sequential access file (to find the ith block of a file, we must start at the begining of that file & follow the pointers until we get to the ith block).
- The space required for the pointers. i.e. Each file requires more space that it would otherwise



- Improve efficiency by clustering blocks into groups but **increases internal fragmentation** (More space is wasted when a cluster is partially full than when the block is partially full.)
- Reliability can be a problem

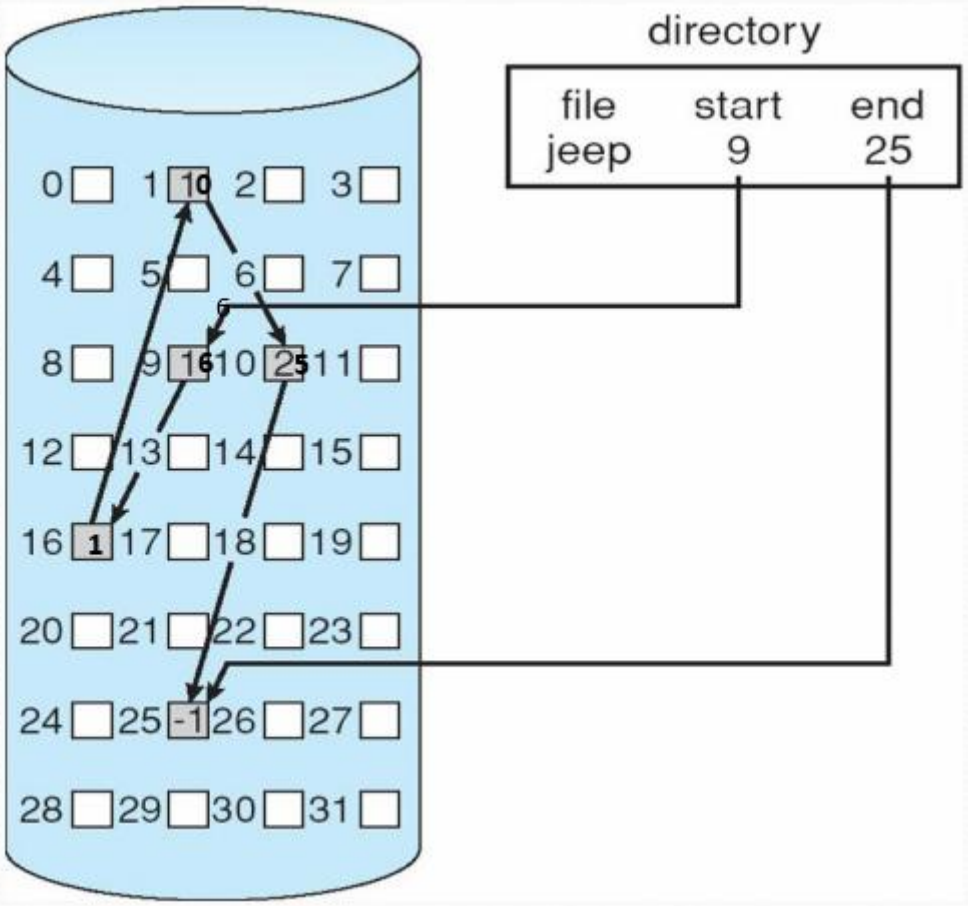
Linked Allocation (Cont.)

Mapping Algorithm

Logical Address (LA) / (BlockSize-**PointerSize**) → Q, R

Block to be accessed = the Qth disk block in the linked chain of disk blocks representing the file.
Displacement into disk block = R **+ PointerSize**

Linked Allocation



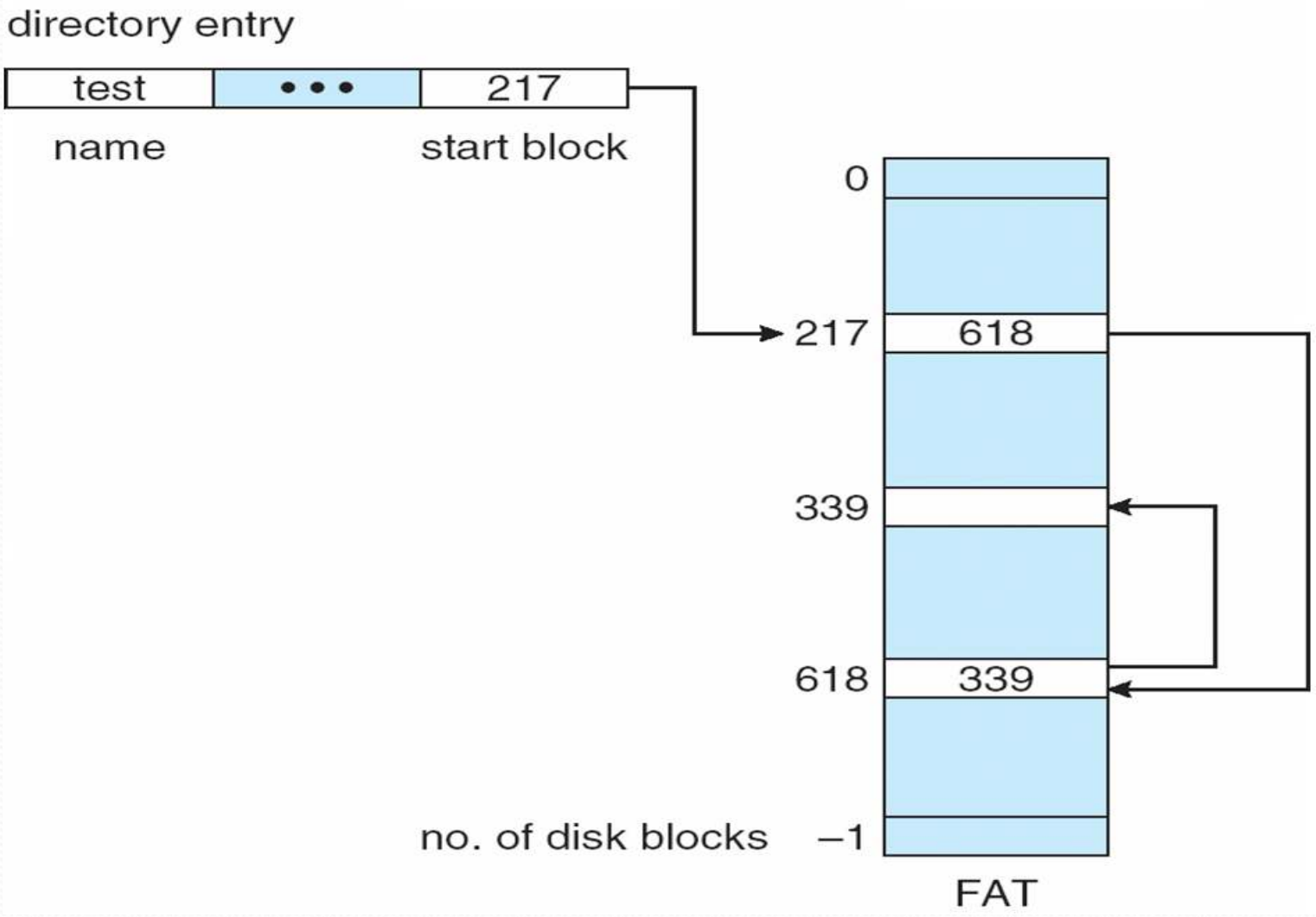
Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation on linked allocation – disk-space allocation used by MS-DOS and OS/2.
  - Simple but efficient method of disk space allocation
  - A section of disk at the Beginning of volume has table, which has 1 entry for each disk block & indexed by block no
  - Much like a linked list, but faster on disk and cacheable
  - Pointers (i.e. disk data blocks numbers) are kept in a table (FAT)
  - Data Block does not hold a pointer; hence data size in a disk block is a power of 2.

Disadvantages:

- Linked allocation cannot support efficient direct access since the pointers to the blocks are scattered with the block themselves all over the disk.
- **Solution:** bringing all the pointers together into 1 location: **the index block**

File-Allocation Table

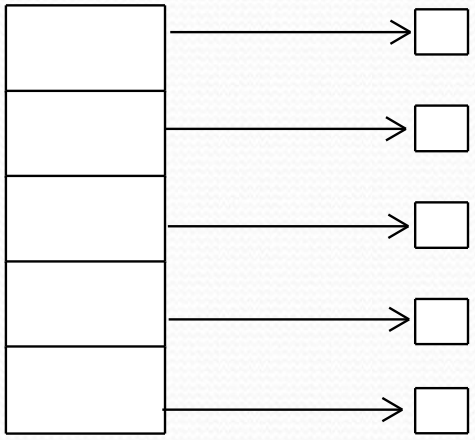


Allocation Methods - Indexed

3. Indexed allocation

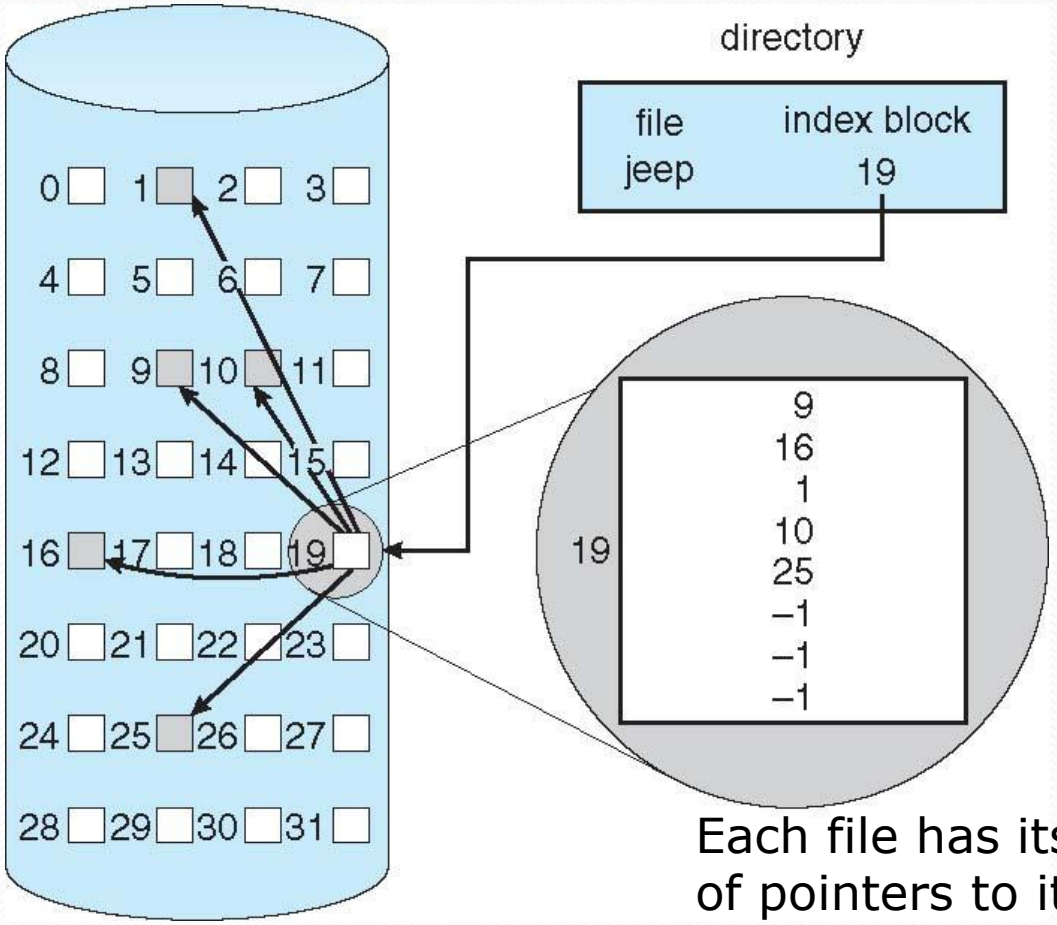
- Solve the problem of linked allocation (No Random access)
- Each file has its own **index block**(s) of pointers to its data blocks, , which is an array of disk-block addresses.

- Logical view



- When the **file is created**, all pointers in the index block are set to null (-1). When the *ith block is first written, a block is obtained from the free-space* manager, and its address is put in the *ith index-block entry.*

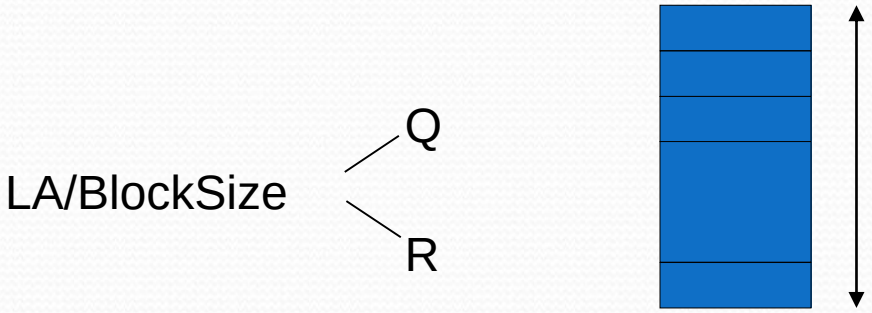
Example of Indexed Allocation



Each file has its own **index block** of pointers to its own data blocks

Indexed Allocation (Cont.)

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

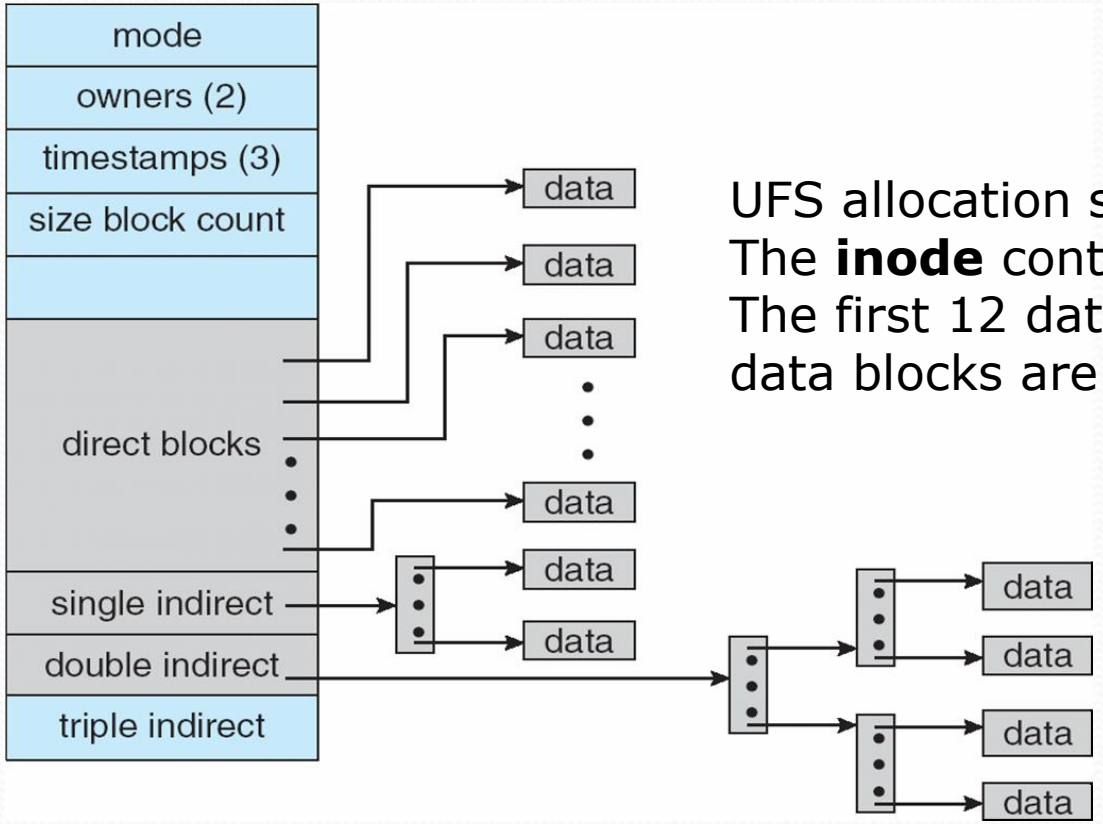


Q = displacement into index table (logical block number)

R = displacement into block (offset)

Combined Scheme: UNIX UFS

4K bytes per block, 32-bit addresses



UFS allocation scheme is index. The **inode** contains the addresses of The first 12 data blocks. Remaining data blocks are addressed indirectly

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

Free Space Management

- File system maintains **free-space list** to track available blocks/clusters
  - (Using term "block" for simplicity)
- How can we keep track of free blocks of the disk?
  - Which blocks are free?

- We need this info when we want to allocate a new block to a file.
  - Allocate a block that is free.

- There are several methods to keep track of free blocks:

  - Bit vector (bitmap) method
  - Linked list method
  - Grouping
  - Counting

Free-Space Management: 1) Bit Vector

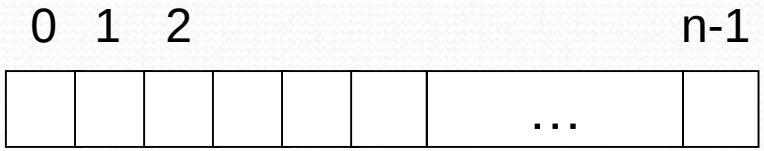
- The free-space list is implemented as a bit map or bit vector.
- Each block is represented by 1 bit.
  - If the block is free, the bit is 1;
  - If the block is allocated, the bit is 0.

- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18,25,26, and 27 are free and the rest of the blocks are allocated.

- The free-space bit map would be 00111100111111000110000001110000
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.

Free-Space Management: 1) Bit Vector (Cont.)

- Implementation: Bit vector (n blocks)



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

Block number calculation

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

0-value words has all bits 0

Where:

- A **bit vector** tracks free/allocated blocks.
- A **word** = group of bits (e.g., 8, 16, or 32 bits depending on system architecture)
- A **0-value word** means all bits in that word are zero (i.e., all blocks in that word are allocated)

Example:

Suppose that we have:

Each word = **8 bits**
Bit vector =
Word 0: 00000000 → All bits 0 → 1st word = 0-value
Word 1: 00000000 → All bits 0 → 2nd word = 0-value
Word 2: 00010000 → Not all 0s → this word has a free block

Step-by-step calculation:

- **Bits per word** = 8
- **Number of 0-value words** before we find a free block = 2
- **Offset of first 1 bit** in Word 2 = 3 (since it's the 3rd bit, counting from the **left**, index 3) -> Offset = 3

Block number = 8*2 + 3 = 19

The **first free block** is block **19**.

Free-Space Management: 1) Bit Vector (Cont.)

- Easy to get contiguous files

- Inefficient unless the entire vector is kept in memory (possible for smaller disks but not for larger ones)
- Bit map requires extra space
  - Example:

Given the following:

- **Block Size:** 4 KB=4096 bytes
- **Disk Size:** 1 TB=1024^4=1,099,511,627,776 bytes

Calculation:

To manage a disk with a bitmap:

1. Calculate the number of blocks on the disk:

Number of Blocks=Block Size/Disk Size

=1,099,511,627,776 / 4096

= 268,435,456 blocks

2. Each block requires **1 bit** in the bitmap to indicate whether it is free or used.

3. Convert the number of bits required for the bitmap into bytes:

Bitmap Size (in bytes)= Number of Blocks / 8 =268,435,456 / 8

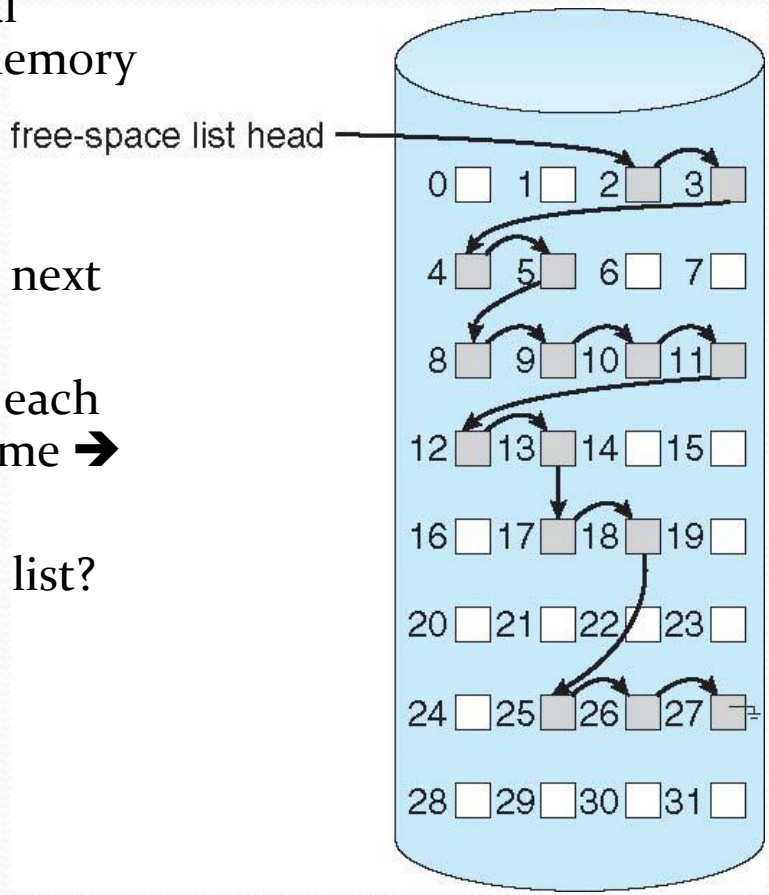
= 33,554,432bytes = 32MB

The space required for the bitmap to manage a 1 TB disk with a 4 KB block size is 32 MB.

Free-Space Management: 2) Linked List

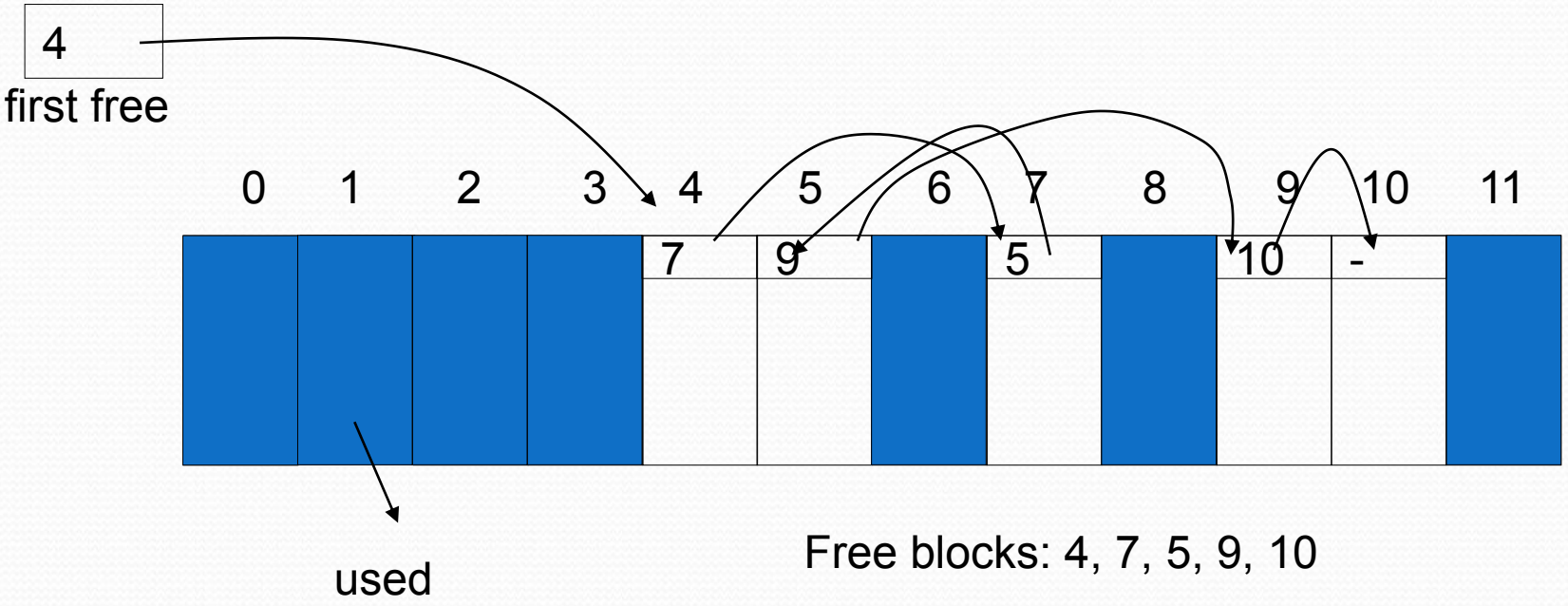
2. Linked list (free list)

- link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory
- No waste of space
- Cannot get contiguous space easily
- The first block contains a pointer to the next free disk block, and so on.
- To traverse the entire list, we must read each block, which requires substantial I/O time ➔ not efficient scheme.
- How many free blocks are in this linked list?
  - 15



Free-Space Management: 2) Linked List(cont.)

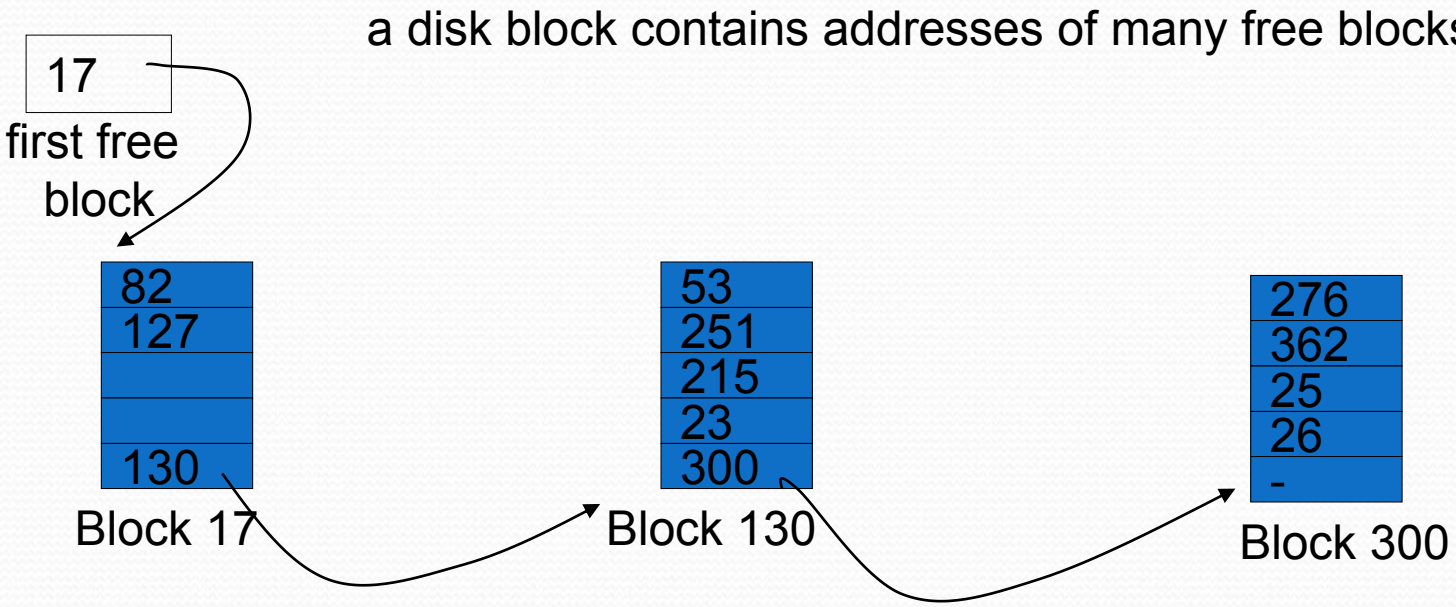
- Each free block has pointer to the next free block
- We keep a pointer to the first free block somewhere (like superblock)
- Features:



Free-Space Management: 3) Grouping

- A modification of the free-list approach is to store the addresses of n free blocks in the first free block.
- The first n-1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Free-Space Management: 3) Grouping



Free blocks are:
82 127 53 251 215 23 276 362 25 26

The last block contains the addresses of another n free blocks, and so on

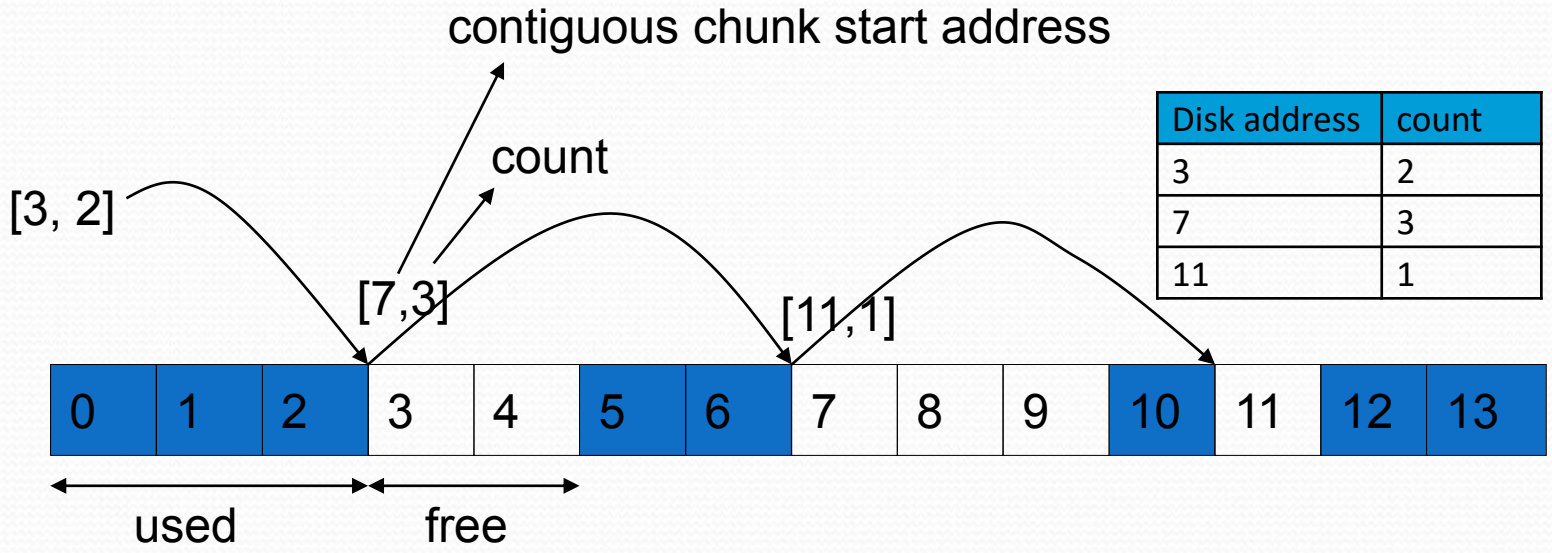
a block containing free block pointers will be free when those blocks are used.

Free-Space Management: 4) Counting

- rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.
- Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

Free-Space Management: 4) Counting

- Besides the free block pointer, keep a counter saying how many block are free contiguously after that free block



Disk address	count
3	2
7	3
11	1

End of Chapter 11