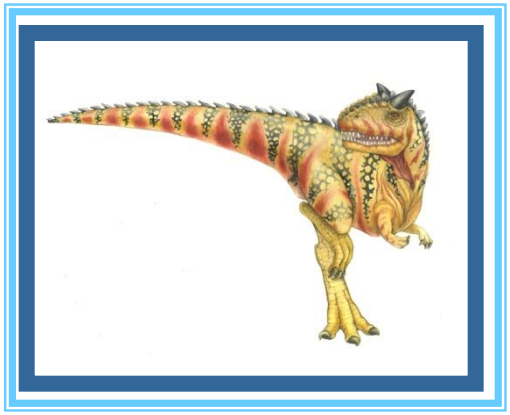
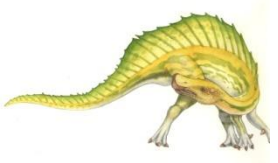
Chapter 11: Implementing File Systems



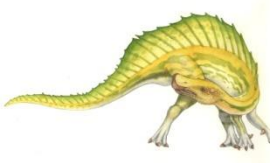
Chapter 11: Implementing File Systems

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



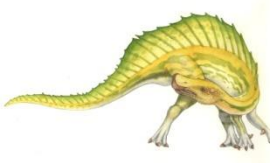
Objectives

- 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



File-System Structure

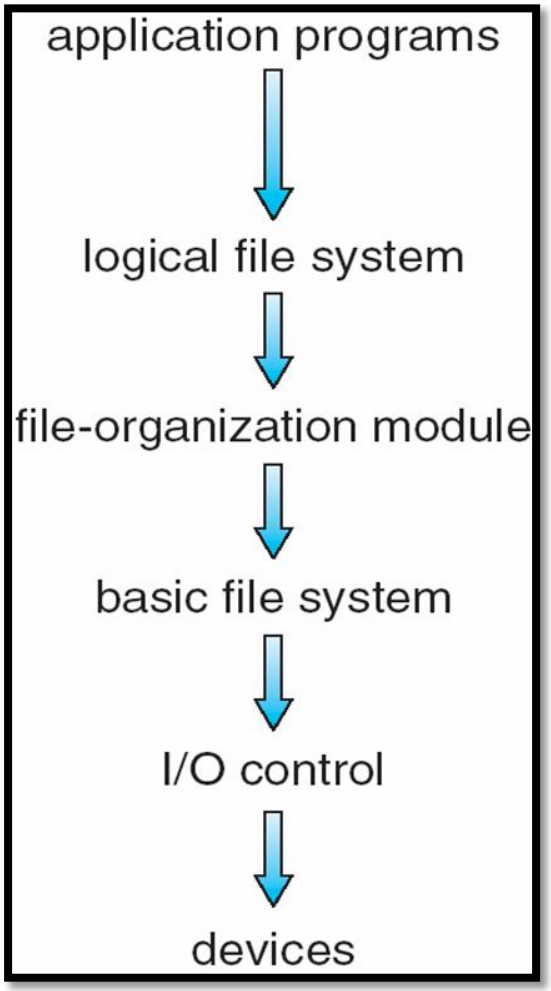
- File system resides on secondary storage (disks)
- The physical unit of transfer is a disk sector (e.g.,512 bytes).
- Logical transfer of data is in *blocks.*
- *Blocks are "chunks" or clusters of disk sectors.*
- The OS imposes a file system for efficient and convenient access to the disk.
- The file system design deals with two distinct matters:
    1. How should the file system look to the user.
    2. Creating data structures and algorithms to map the logical file system onto the physical secondary storage device.



Layered File System

A layered design abstraction

- **I/O control** :device drivers and interrupt service routines that perform the actual block transfers.
- **Basic file system** : issues generic low-level commands to device drivers.
- **File organization module** : translates logical block addresses to physical block addresses, and file space manager.
- **Logical file system** : handles metadata that includes filesystem structure (e.g., directory structure and file control blocks (FCB's).



On –disk File System structures

On-disk and in-memory structures are needed to implement a file system:

On disk:

1. **Boot control block** (per volume): needed to boot OS from a disk partition ;bootblock [UFS],Master partition boot sector[NTFS]
2. **Volume control block** (per volume)**:** holds details about partition (e.g., blocks in partition, freeblock count, …) ;superblock [UFS],Master File Table [NTFS]
3. **Directory structure** (per file system)**:** to organize files ;In [UFS] includes file names and associated inode numbers, Master File Table [NTFS]
4. **File control block (FCB)** (per file) **–** storage structure consisting of information about a file; stored in Master File Table [NTFS]

A Typical File Control Block

[Linux] inode is the term for 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 File System structures

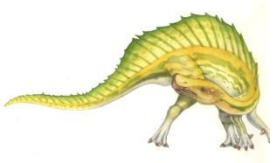
On-disk and in-memory structures needed to implement a file system:

In-memory: (mount time)

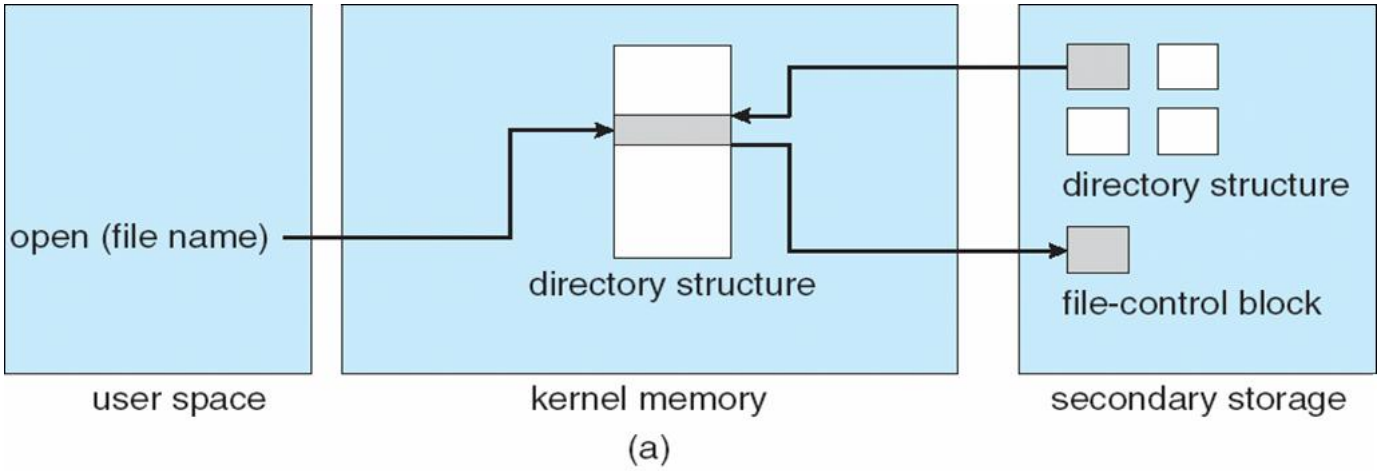
1. **In memory mount table:** information about each mounted volume
2. **In-memory directory structure:** cache holds information of recently accessed directories
3. **System-wide open-file table:** copy of FCB of each open file
4. **Per-process open-file table:** pointer to system-wide open file table.
5. **Buffers** hold file system blocks when reading or written to disk.

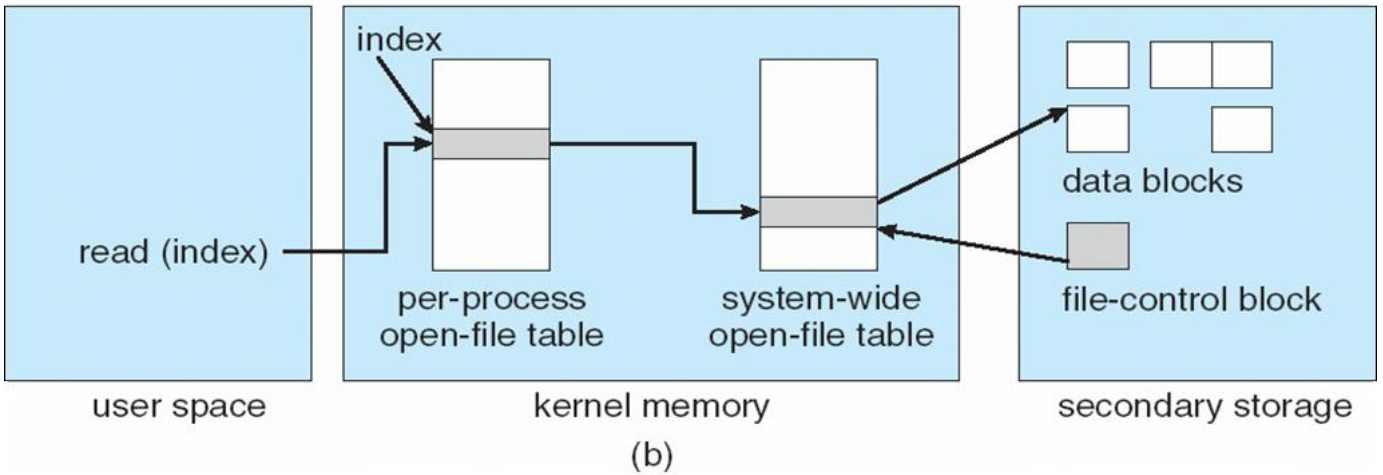
In-Memory File System Structures

- The following figure illustrates the necessary file system structures provided by the operating systems.
- Figure 12-3(a) refers to opening a file.
- Figure 12-3(b) refers to reading a file.



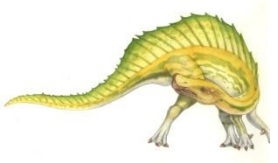
In-Memory File System Structures





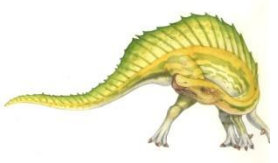
Directory Implementation

- **Linear list** of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- **Hash Table** – linear list with hash data structure.
  - decreases directory search time
  - **collisions** – situations where two file names hash to the same location
  - To resolve the collisions, a chained-overflow hash table is used. Each entry can be a linked list instead of an individual value.



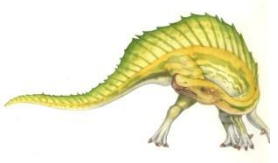
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

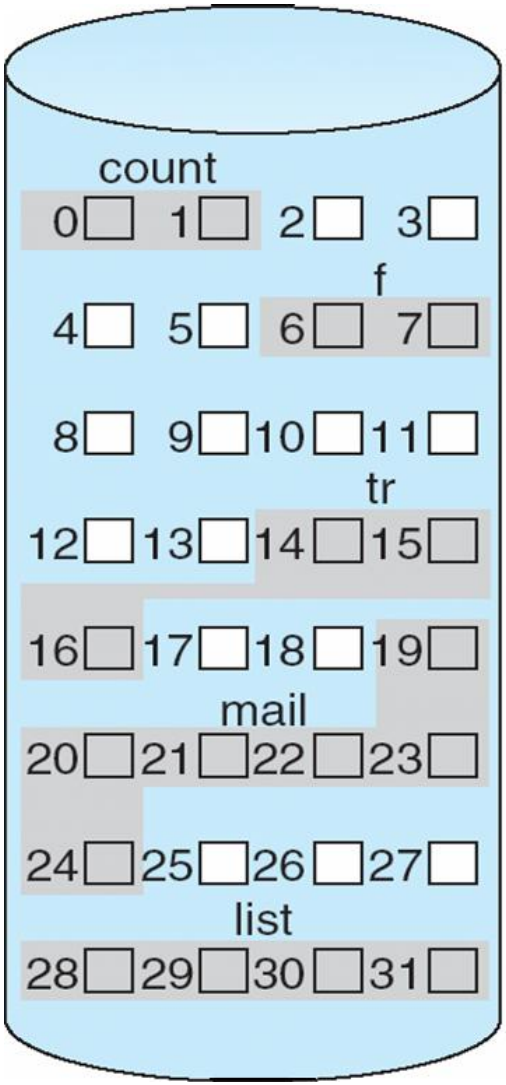


Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- External fragmentation
- Sequential or direct access
- Wasteful of space
- Files cannot grow



Contiguous Allocation of Disk Space



directory

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

Extent-Based Systems

- Many newer file systems (I.e. Veritas File System) use a modified contiguous allocation scheme
- The contiguous space can enlarge through extents to increase flexibility an decrease external fragmentation.

- Extent-based file systems allocate disk blocks in **extents**

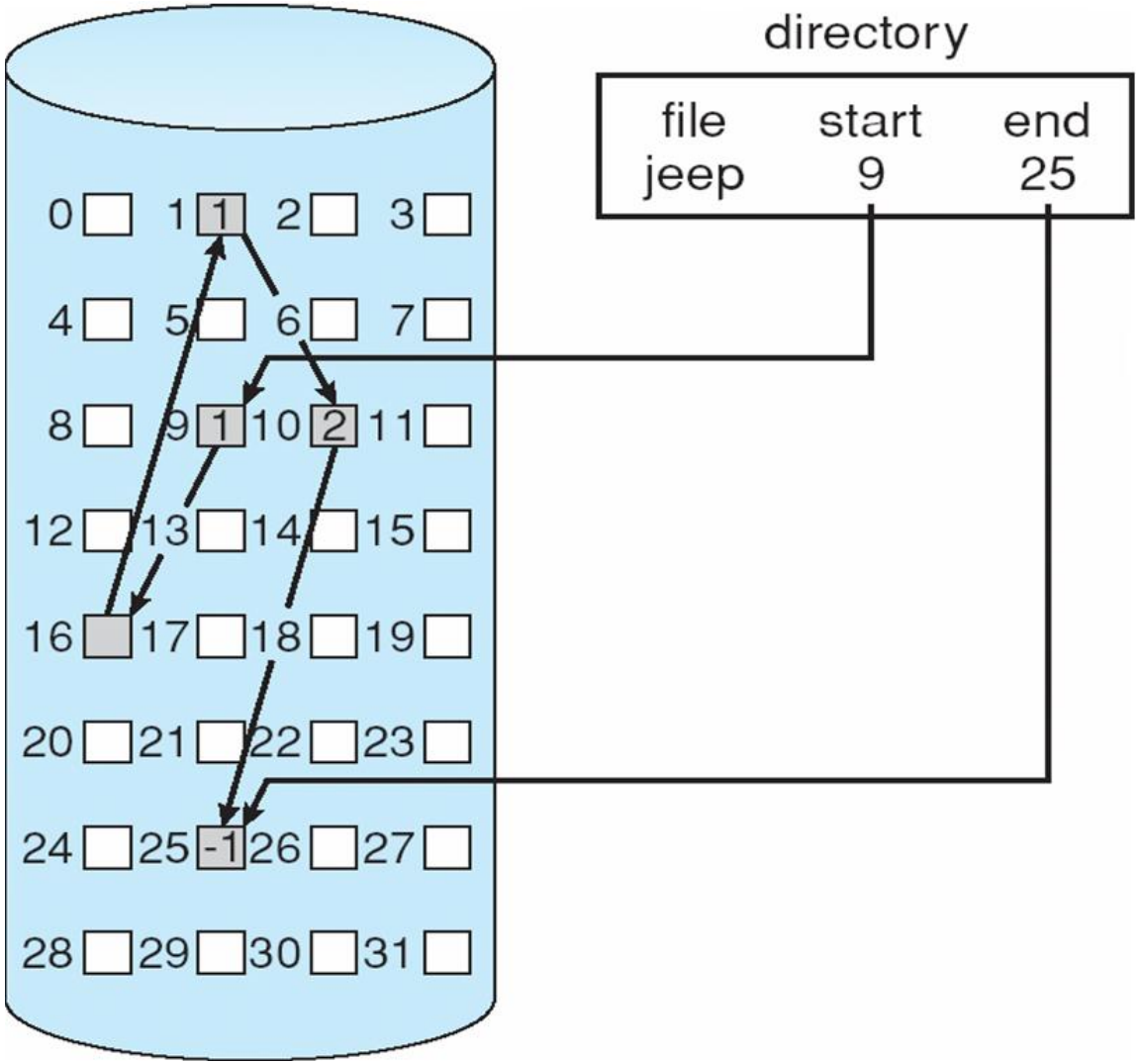
- An **extent** is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents.

- The location of a file's blocks is : starting location (block #) , length (number of blocks) , and a link to the first block of the next extent.

Linked Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file.
- Simple – need only starting address
- Free-space management system – no waste of space
- Only sequential access, No direct access
- No external fragmentation

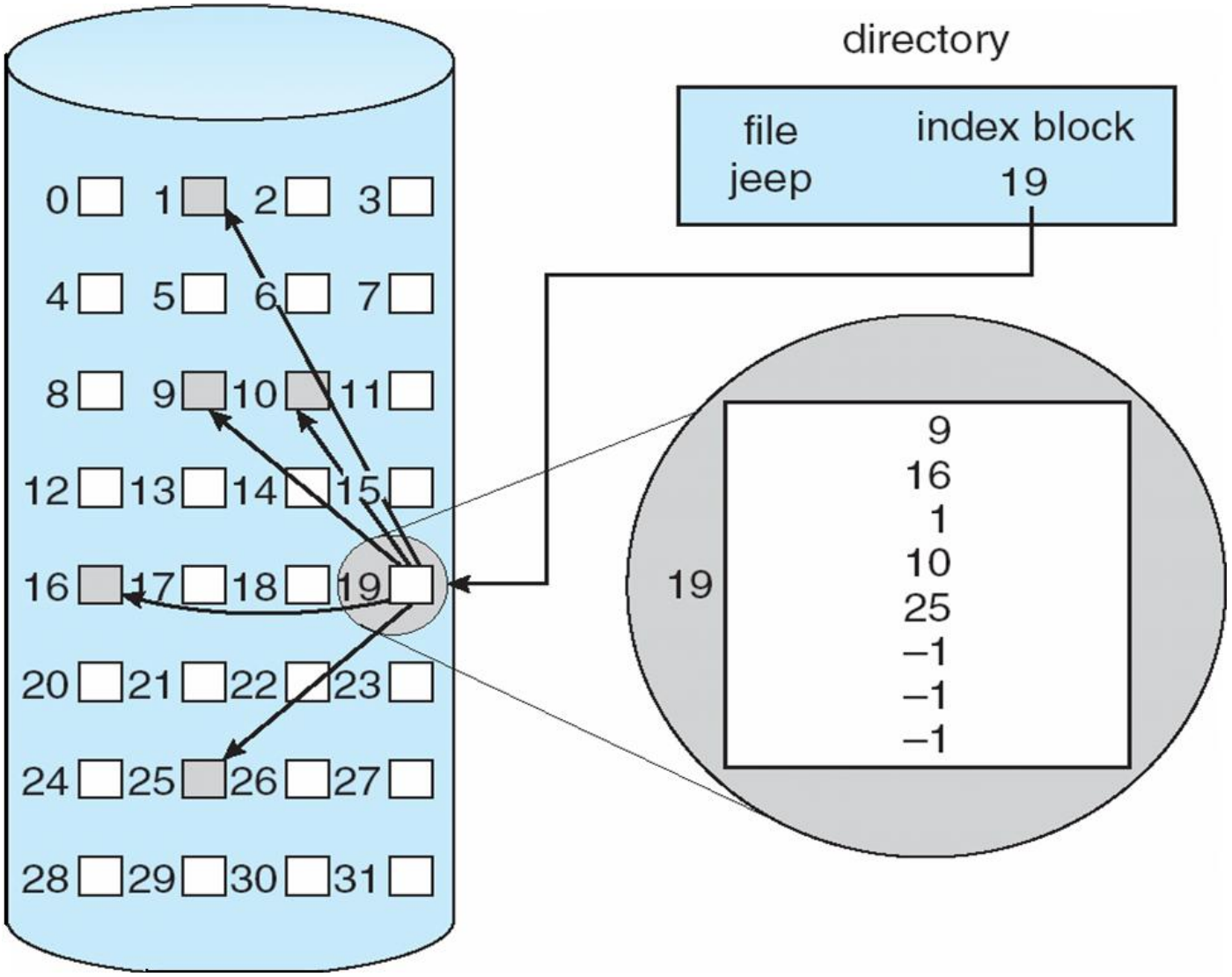
Linked Allocation



Indexed Allocation

- Brings all pointers together into the *index block.*
- Each file has its own index block, which is an array of disk-block addresses
- Need index table
- Direct access
- Dynamic access without external fragmentation, but have overhead of index block.

Example of Indexed Allocation

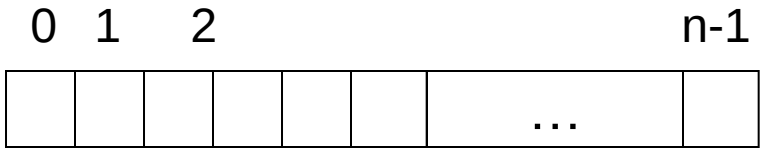


Free-Space Management

- Reusing disk space from deleted files for new files is desirable.
- To keep track of free disk space, the system maintains a **free-space list**.
- The free-space list records all free disk blocks.
- To create a file , we search the FSL for the required amount of space and allocate that space to the new file. And remove it from the list.
- When the file is deleted, its disk space is added to the FSL.
- FSL implementation in two methods:
  - Bit Vector
  - Linked List

Free-Space Management (cont.)

- **Bit vector** (*n* blocks) each block is represented by 1 bit



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

For example,

if the disk had 10 blocks, and blocks 2, 4, 5, and 8 were free, while blocks 0, 1, 3, 6, 7, and 9 were in use, the **bit vector** would be represented as: **1101001101**

Advantages: Simplicity, and Efficiency in finding the first free block (Direct Access)

Disadvantages: waste of space because it is Inefficient unless the entire vector is kept in main memory.

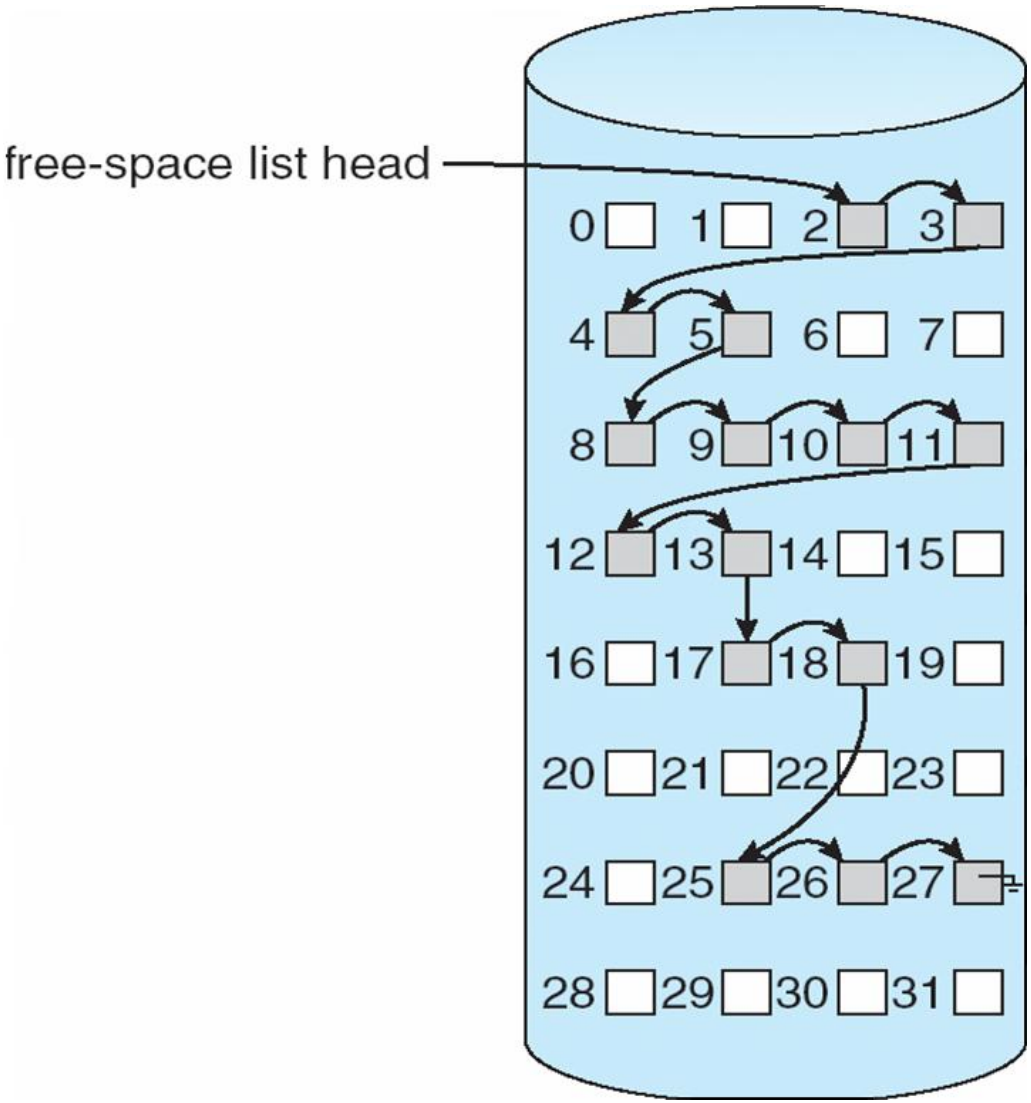
Free-Space Management (Cont.)

- **Linked list** (Linked List of Free Blocks)
  - The address (block number) of the first free block is kept in a designated place in memory.
  - The first free block contains a pointer to the next free block, it contains a pointer to the next free block, and so forth.
  - Can add a free block to the beginning of the free list.
  - Can remove a free block from the beginning of the free list.

  - ***Advantage:*** No waste of space
  - ***Disadvantage:*** Cannot get contiguous space easily

    (sequential access)

Linked Free Space List on Disk



End of Chapter 11