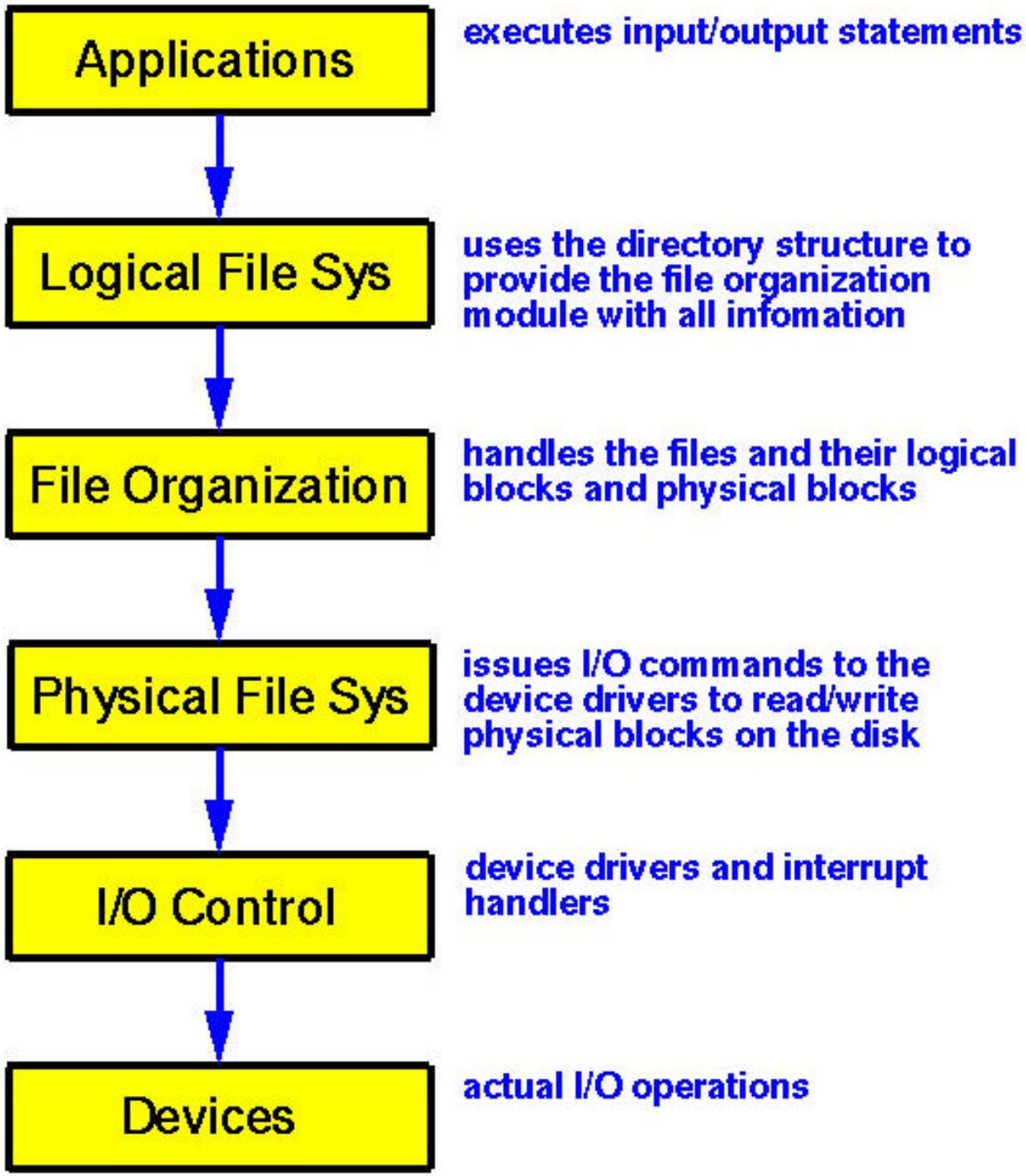
Part III
Storage Management
Chapter 11: File System Implementation



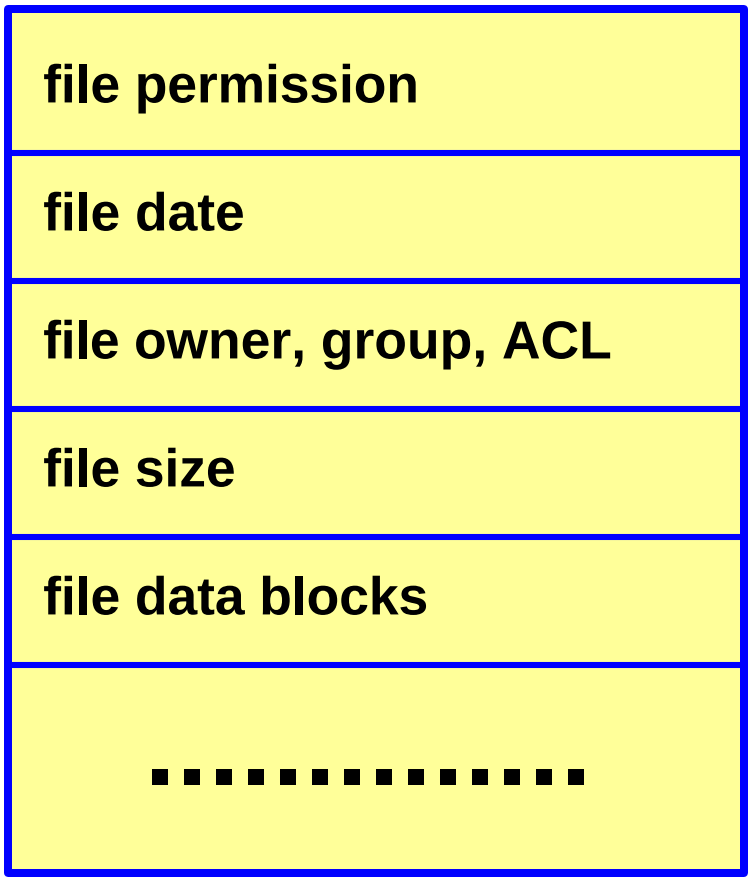
Layered File System

Overview: 1/4

- A file system has on-disk and in-memory information.
- A disk may contain the following for implementing a file system on it:
  - A boot control block per volume
  - A partition control block per volume
  - A directory structure per file system
  - A file control block per file
- In-memory information include
  - An in-memory partition table
  - An in-memory directory structure
  - The system-wide open-file table
  - The per-process open-file table

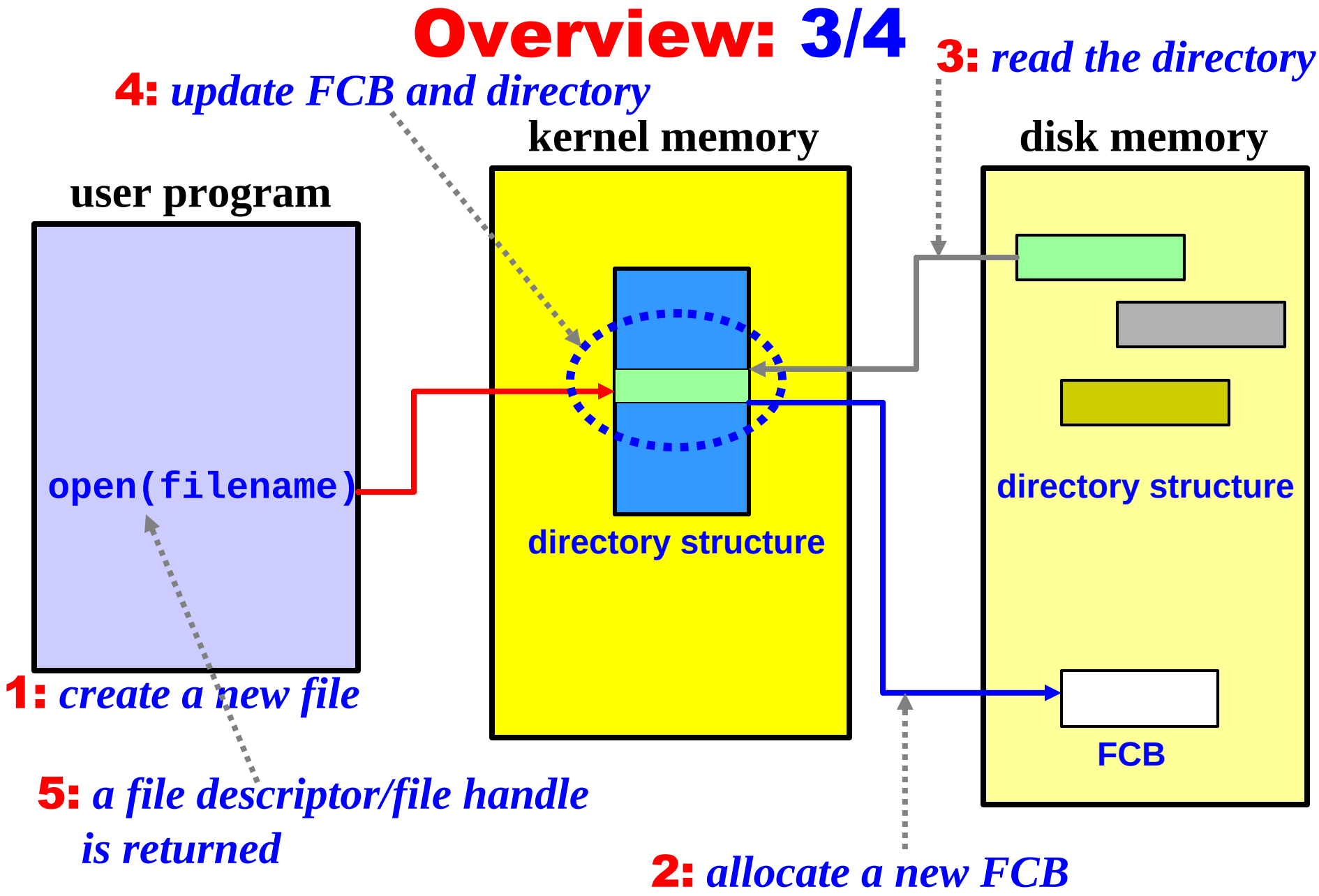
Overview: 2/4

a typical file control block FCB

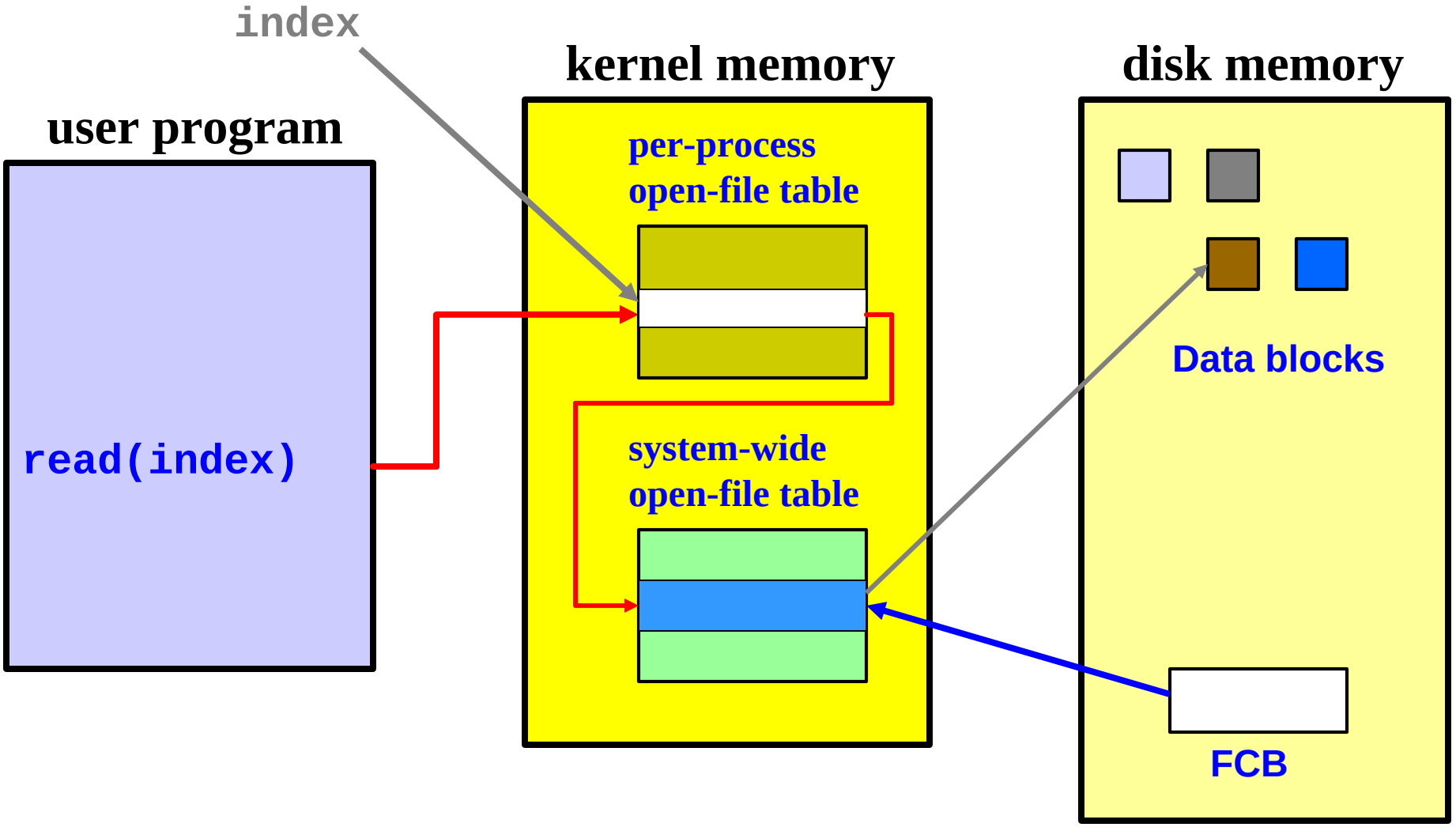


- A FCB, *f*ile *c*ontrol *b*lock, contains the details of a file.
- In Unix, a FCB is called an i-node.

Overview: 3/4


4: update FCB and directory
3: read the directory
kernel memory
disk memory
user program
open(filename)
directory structure
directory structure
FCB
1: create a new file
5: a file descriptor/file handle is returned
2: allocate a new FCB

Overview: 4/4

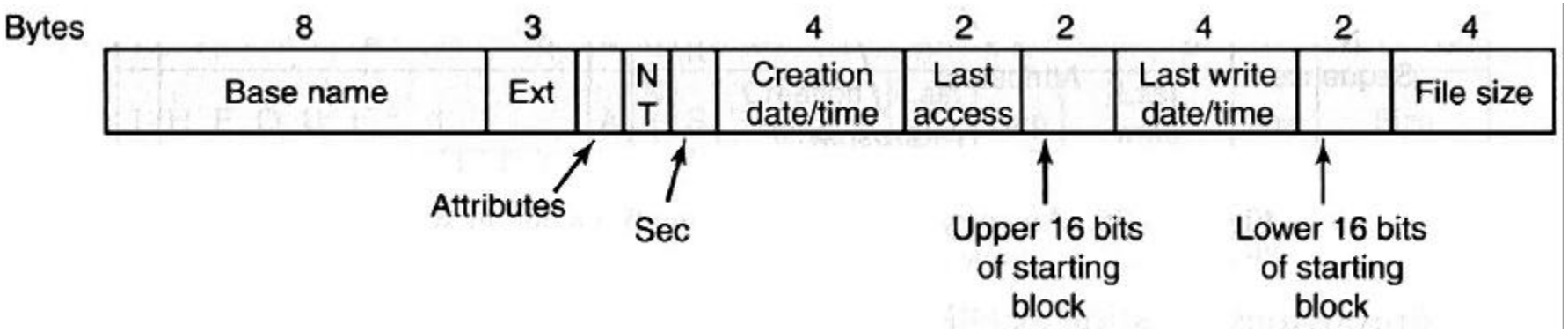

index
kernel memory
disk memory
user program
per-process
open-file table
Data blocks
read(index)
system-wide
open-file table
FCB

Directory Implementation

- A File directory is usually implemented as a linked-list or a tree (*e.g.*, B-tree), a hash table with chaining, or the combination of both.
- The problem with the linked-list approach is the poor performance in searching for a file. Directory search is a very frequently performed operation.
- The hash table approach speeds up search; however, we must deal with the problem of collisions. Thus, chaining is necessary.

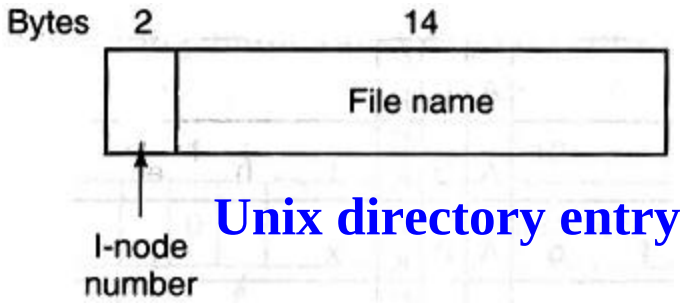
Directory Entries: 1/2

- A directory is simply a file!
- A directory entry may be very simple like the one used by MS-DOS. Or, it may be quite complex like the Unix *i-node*.



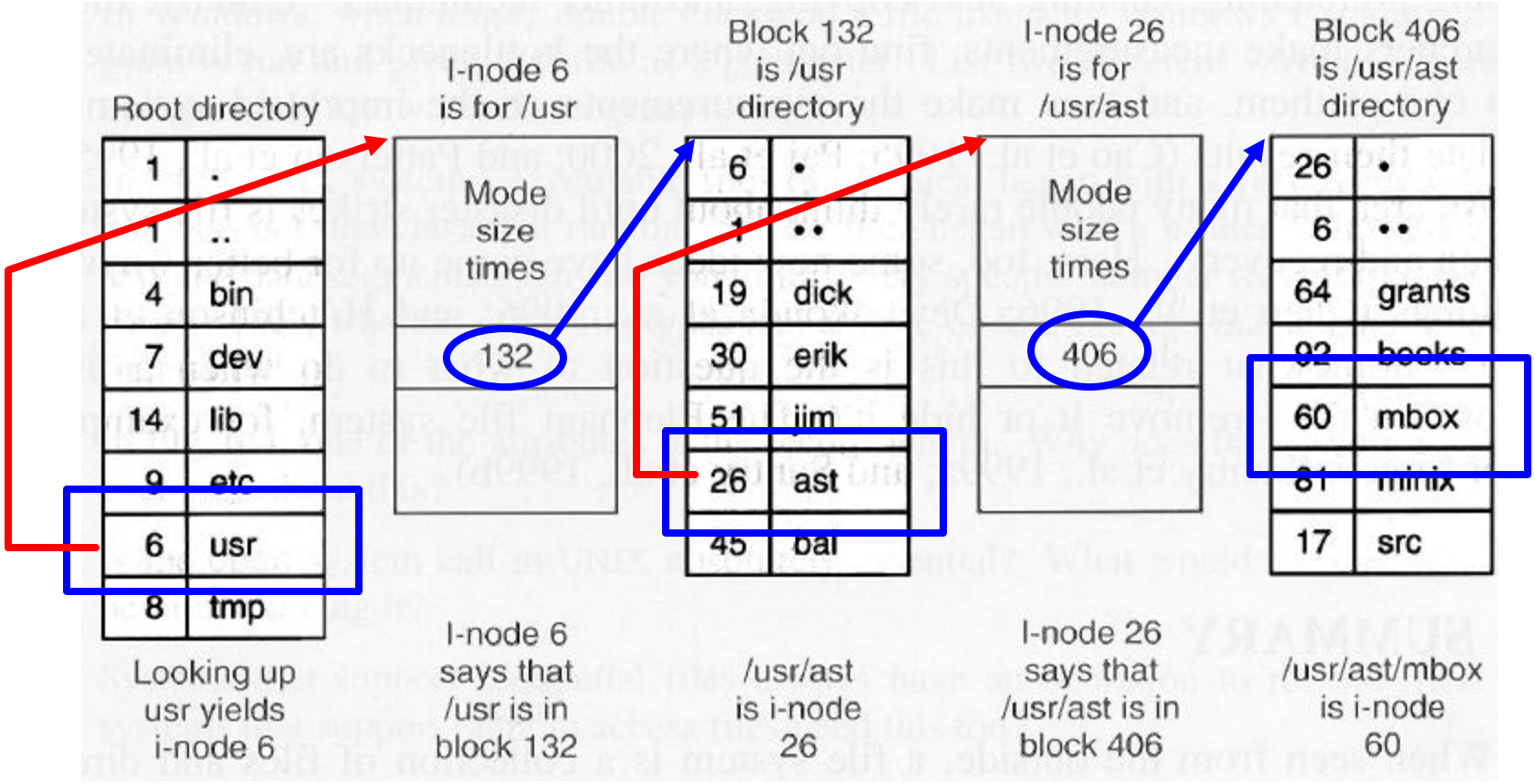
extended MS-DOS directory entry used in Windows 98

Directory Entries: 2/2



Unix directory entry

find /usr/ast/mbox



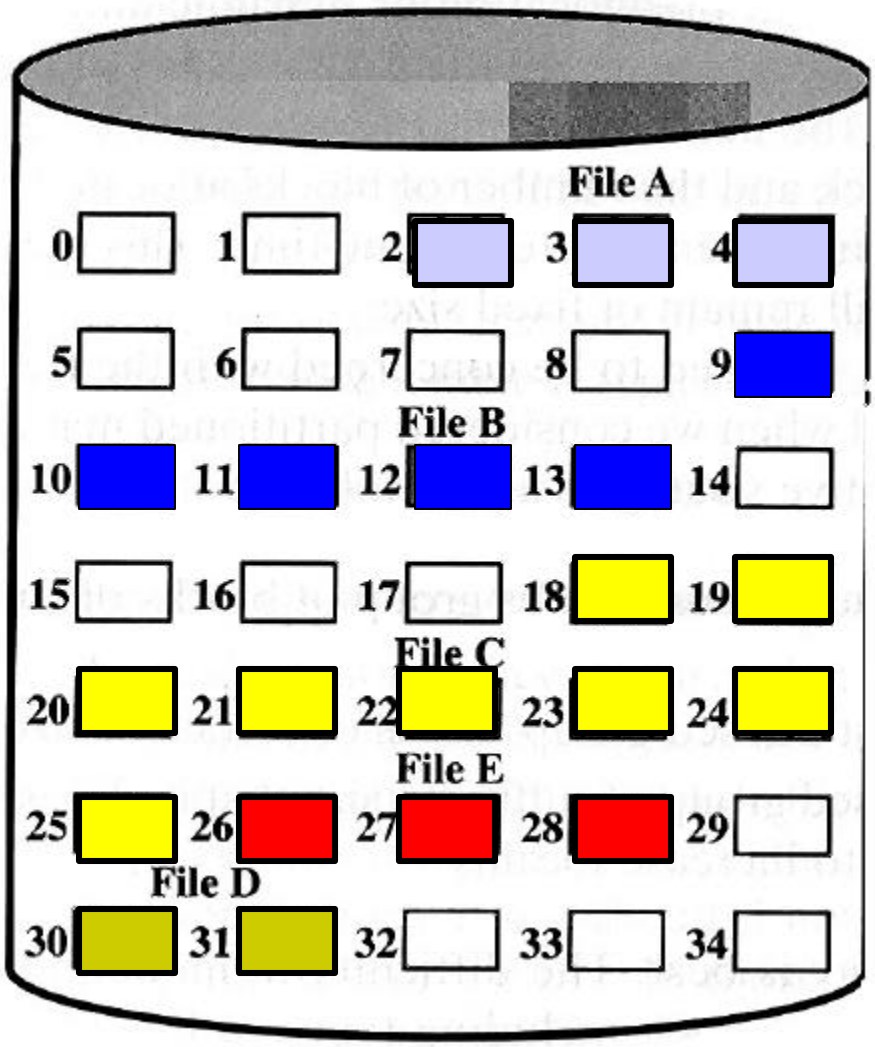
File Allocation Methods

- There are three typical file space allocation methods:
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation

Contiguous Allocation: 1/3

- With the contiguous allocation method, a user must indicate the file size before creating the file.
- Then, the operating system searches the disk to find contiguous disk blocks for the file.
- The directory entry is easy. It contains the initial disk address of this file and the number of disk blocks.
- Therefore, if the initial address is b and the number of blocks is n, the file will occupy blocks b, $b+1$, $b+2$, …, $b+n-1$.

Contiguous Allocation: 2/3



directory

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Since blocks are allocated contiguously, external fragmentation may occur. Thus, compaction may be needed.

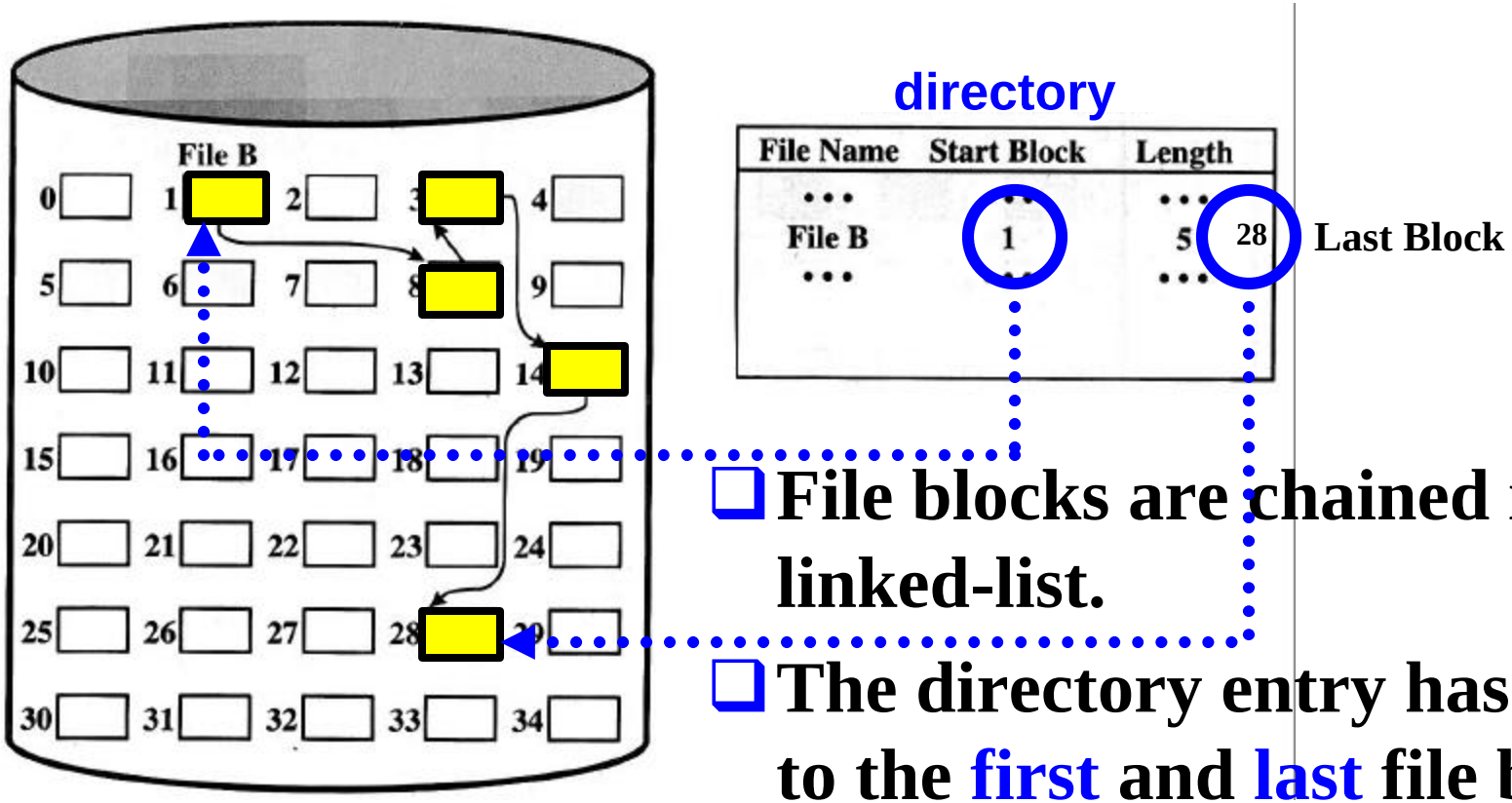
Contiguous Allocation: 3/3

- Contiguous allocation is easy to implement.
- Its disadvantages are
  - It can be considered as a form of dynamic memory allocation, and external fragmentation may occur and compaction may be needed.
  - It is difficult to estimate the file size. The size of a file may grow at run time and may be larger than the specified number of allocated blocks. In this case, the OS must move the blocks in order to provide mode space. In some systems, this is simply an error.

Linked Allocation: 1/3

- With the linked allocation approach, disk blocks of a file are chained together with a linked-list.
- The directory entry of a file contains a pointer to the first block and a pointer to the last block.
- To create a file, we create a new directory entry and the pointers are initialized to nil.
- When a write occurs, a new disk block is allocated and chained to the end of the list.

Linked Allocation: 2/3

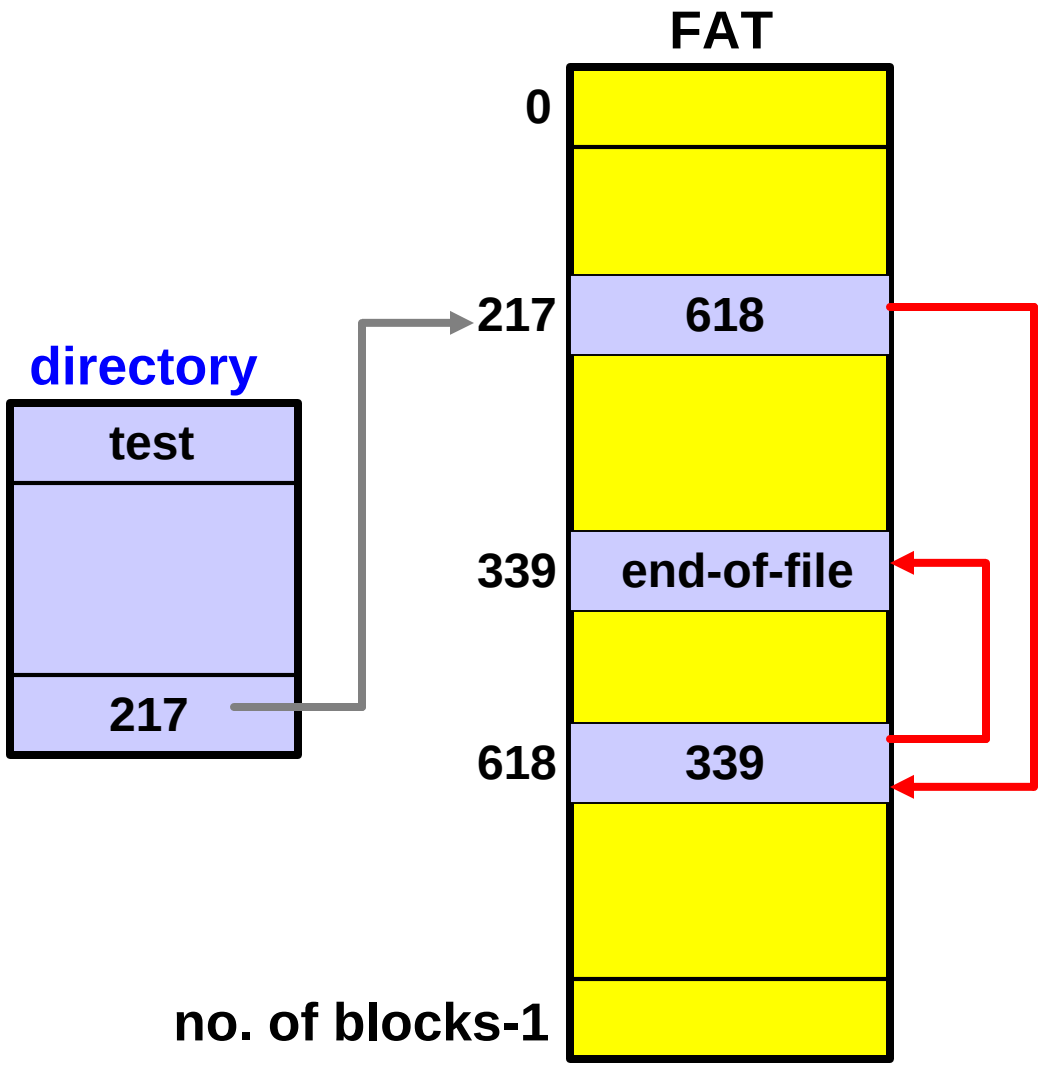


- File blocks are chained into a linked-list.
- The directory entry has pointers to the first and last file blocks.
- Append is difficult to do without the last pointer.

Linked Allocation: 3/3

- **Advantages:**
  - **File size does not have to be specified.**
  - **No external fragmentation.**
- **Disadvantages:**
  - **It does sequential access efficiently and is not for direct access**
  - **Each block contains a pointer, wasting space**
  - **Blocks scatter everywhere and a large number of disk seeks may be necessary**
  - **Reliability: what if a pointer is lost or damaged?**

File Allocation Table (FAT)

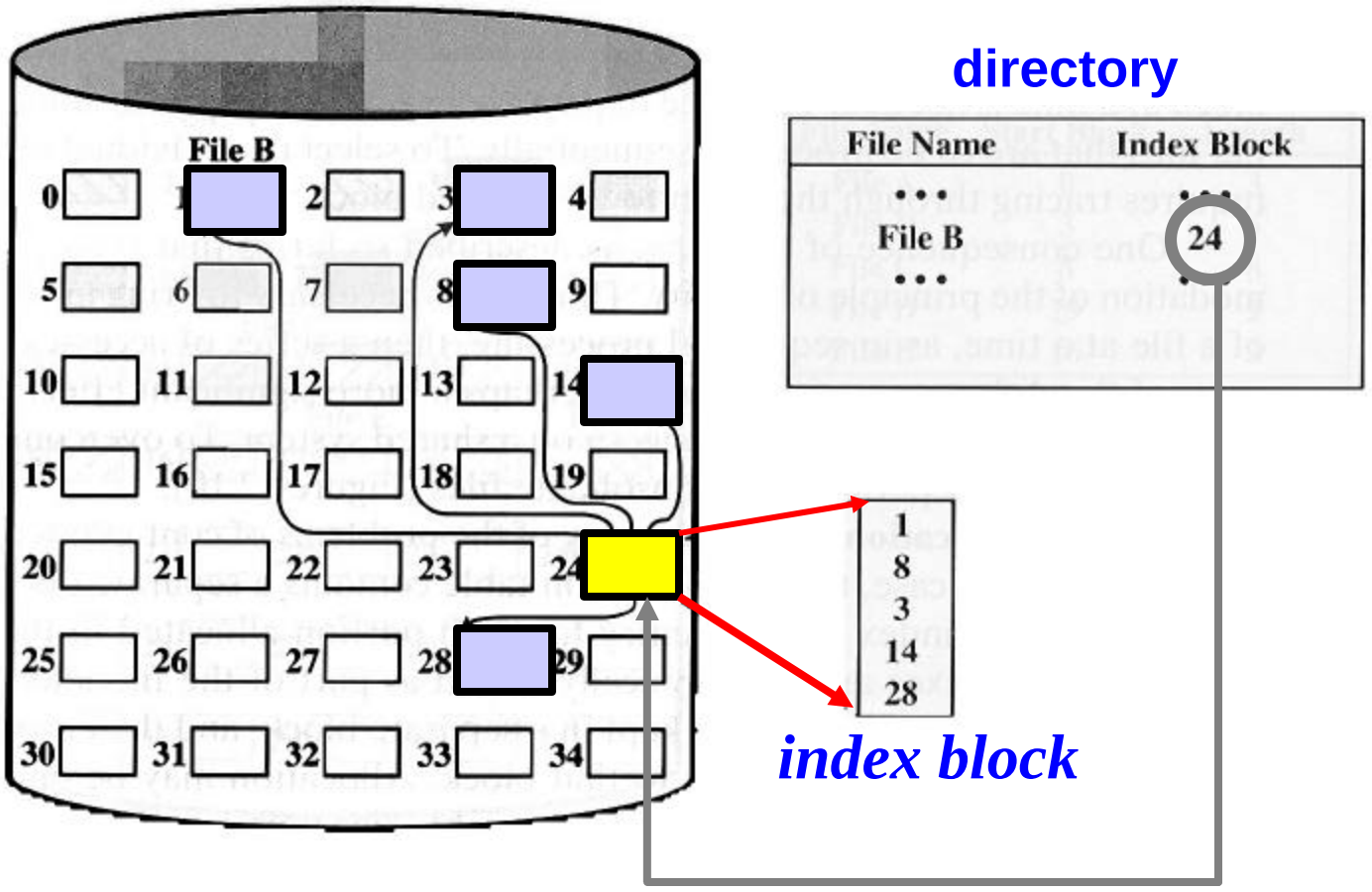


- This is a variation of the linked allocation by pulling all pointers into a table, the file allocation table (FAT).
- Large no. of disk seeks.
- Can do direct access.
- FAT needs space.
- The left diagram shows file `test` has its first block at 217, followed by 618, 339 (end of file).
- What if FAT is damaged? We all know it well!

Indexed Allocation: 1/4

- Each file has an *index block* that is an array of disk block addresses.
- The *i*-th entry in the index block points to the *i*-th block of the file.
- A file's directory entry contains a pointer to its index. Hence, the index block of an indexed allocation plays the same role as the page table.
- Index allocation supports both sequential and direct access without external fragmentation.

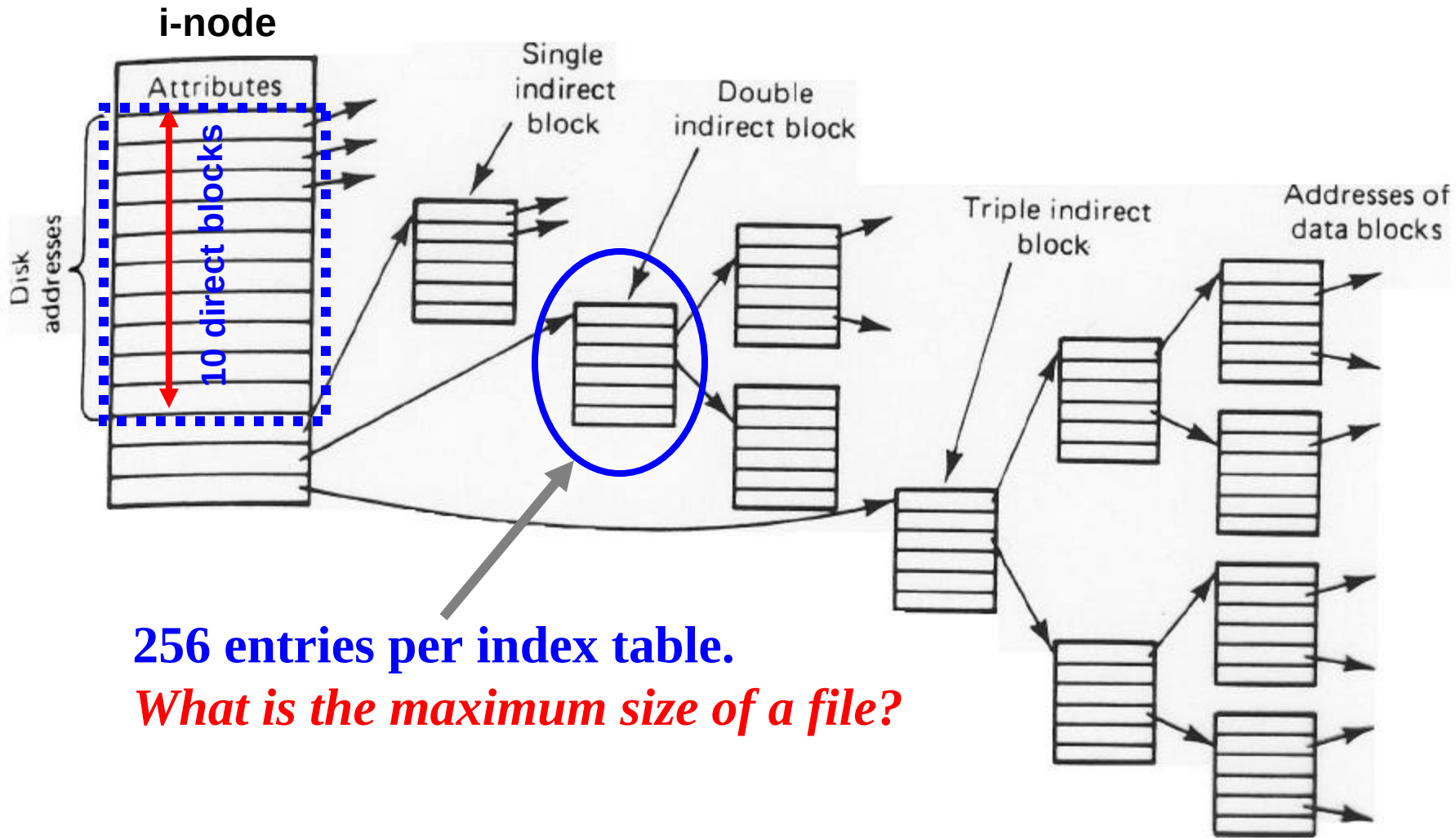
Indexed Allocation: 2/4



Indexed Allocation: 3/4

- The indexed allocation suffers from wasted space. The index block may not be fully used (*i.e.*, internal fragmentation).
- The number of entries of an index table determines the size of a file. To overcome this problem, we can
  - Have multiple index blocks and chain them into a linked-list
  - Have multiple index blocks, but make them a tree just like the indexed access method
  - A combination of both

Indexed Allocation: 4/4



256 entries per index table.

What is the maximum size of a file?

Free Space Management

- How do we keep track free blocks on a disk?
- A free-list is maintained. When a new block is requested, we search this list to find one.
- The following are commonly used techniques:
  - Bit Vector
  - Linked List
  - Linked List + Grouping
  - Linked List+Address+Count

Bit Vector

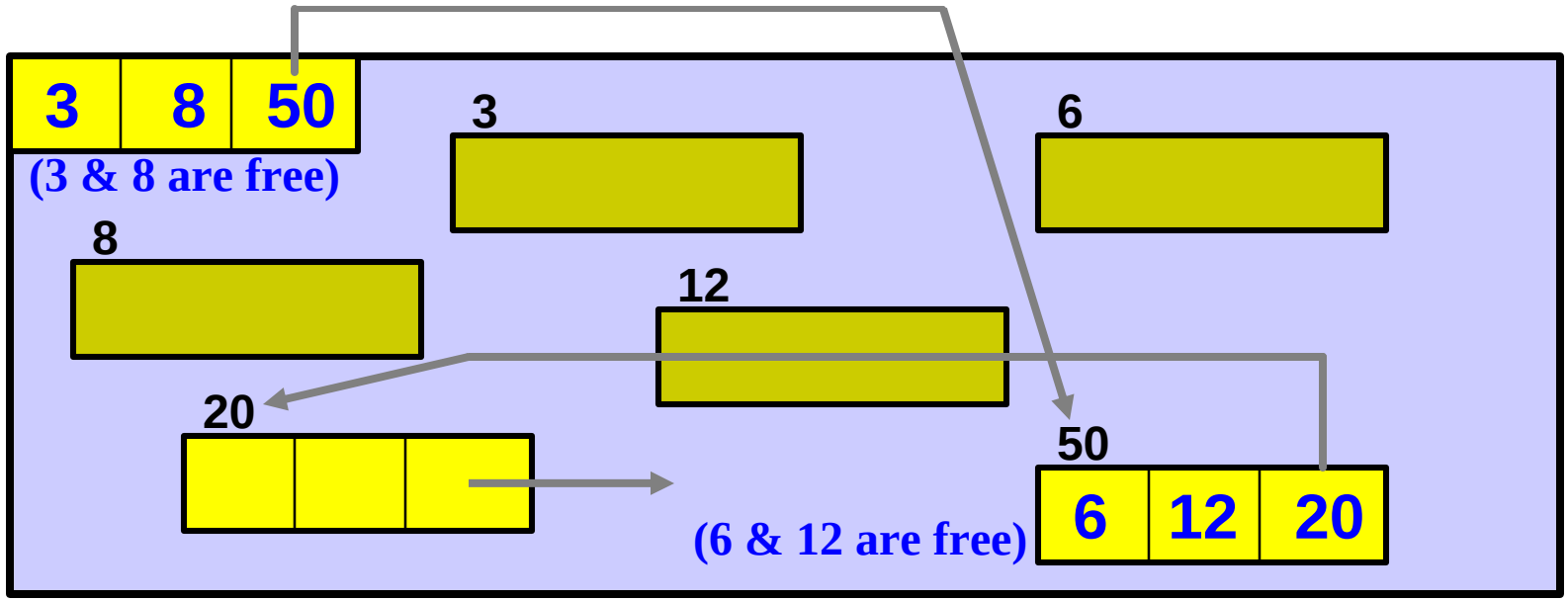
- Each block is represented by a bit in a table. Thus, if there are *n* disk blocks, the table has *n* bits.
- If a block is free, its corresponding bit is 1.
- When a block is needed, the table is searched. If a 1 bit is found in position *k*, block *k* is free.
- If the disk capacity is small, the whole bit vector can be stored in memory. For a large disk, this bit vector will consume too much memory.
- We could group a few blocks into a cluster and allocate clusters. This saves space and may cause internal fragmentation.
- Another possibility is the use of a summary table.

Linked List

- Like the linked allocation method, free blocks can be chained into a linked list.
- When a free block is needed, the first in the chain is allocated.
- However, this method has the same disadvantages of the linked allocation method.
- We can use a FAT for the disk and chain the free block pointers together. Keep in mind that the FAT may be very large and consume space if it is stored in memory.

Grouping

- The first free block contains the addresses of *n* other free blocks.
- For each group, the first *n*-1 blocks are actually free and the last (*i.e.*, *n*-th) block contains the addresses of the next group.
- In this way, we can quickly locate free blocks.



Address + Counting

- We can make the list short with the following trick:
  - Blocks are often allocated and freed in groups
  - We can store the address of the first free block and the number of the following *n* free blocks.