CS420: Operating Systems

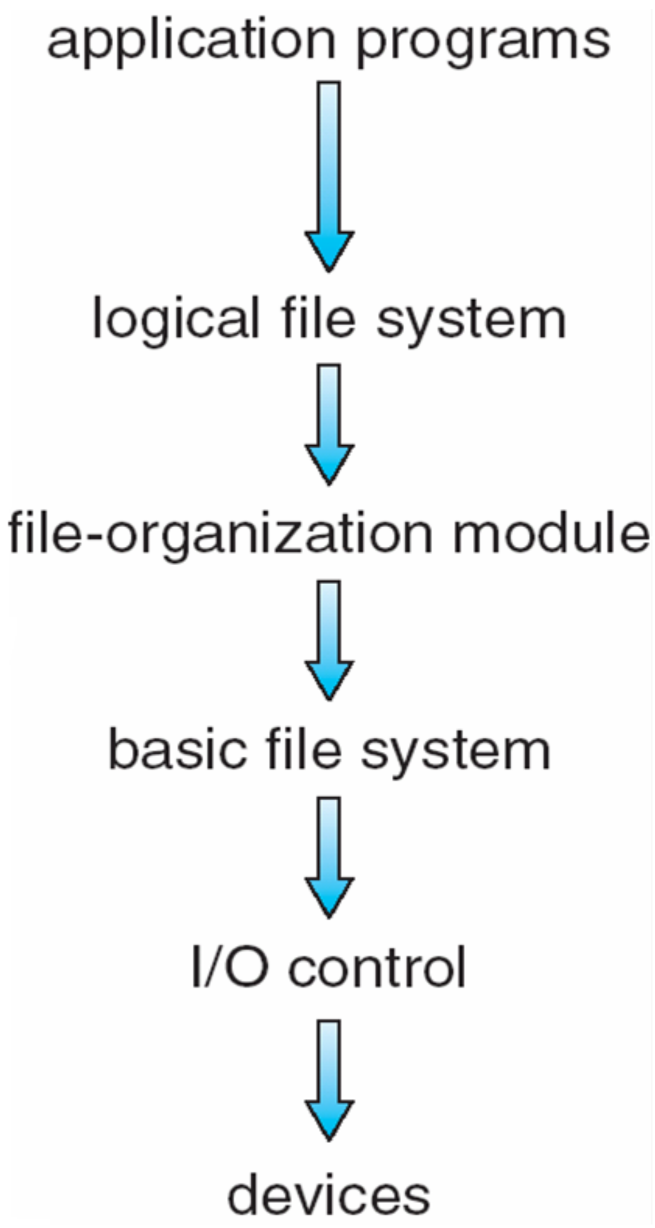
File-System Implementation

James Moscola
Department of Physical Sciences
York College of Pennsylvania

Based on Operating System Concepts, 9th Edition by Silberschatz, Galvin, Gagne

File-System Structure

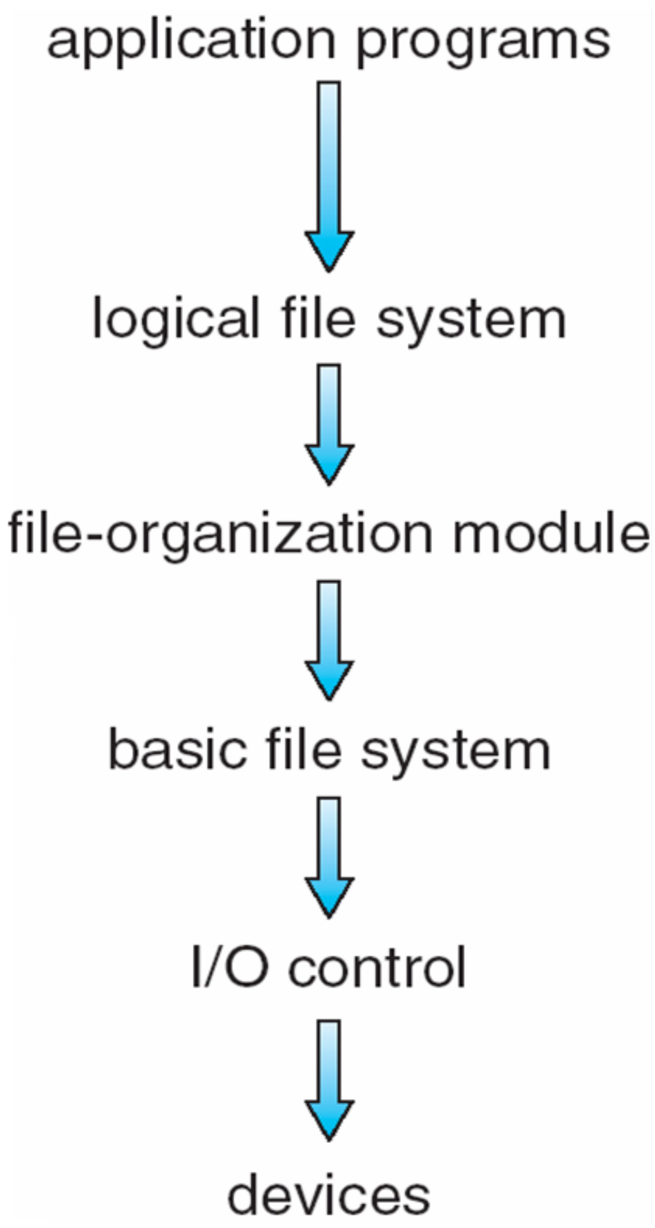
- **File system resides on secondary storage (disks)**
  - Provides a user interface to storage, mapping logical file blocks to physical storage blocks (disk I/O is performed in blocks, usually 512 byte blocks)
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- **File system organized into layers**



File System Layers

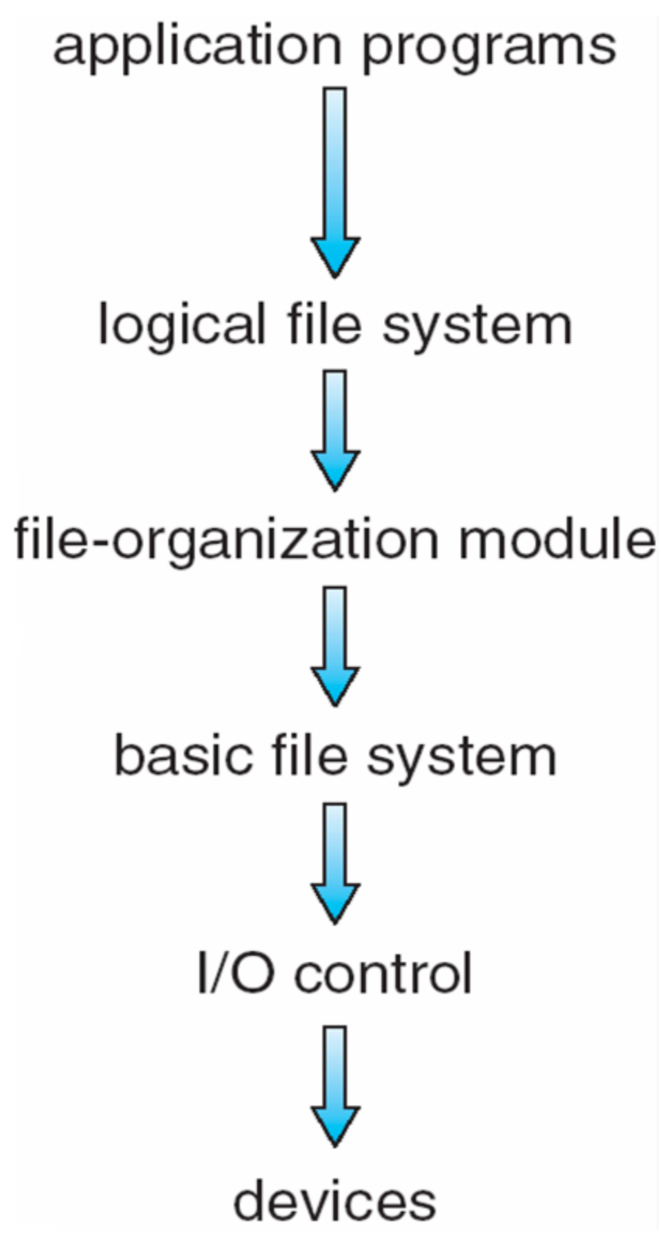
- **I/O control layer - consists of device drivers and interrupt handlers to transfer information between main memory and the disk**
  - A device driver can be thought of as a translator
  - Its input consists of high-level commands such as "retrieve block 123"
  - Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system (e.g. "read drive1, cylinder 72, track 2, sector 10, into memory location 1060")

- **Basic file system layer - gives commands to the device driver to read/write physical block on the disk**
  - Also manages memory buffers and caches (allocation, freeing, replacement of buffers)
    - Buffers hold data in transit
    - Caches hold frequently used data



File System Layers (Cont.)

- **File organization module - knows about files, logical blocks, and physical blocks**
  - Translates logical block # to physical block # (logical blocks are numbered from 0 through N)
  - Manages free space, disk allocation



- **Logical file system layer - manages metadata information (file-system structure)**
  - Translates symbolic file name into file number, file handle, location by maintaining file control blocks (inodes in Unix)
  - Manages the file-system directory
  - Manages protection and security of the file-system

File-System Implementation

- **We have system calls at the API level (e.g. `open()`, `close()`, `read()`, `write()`), but how do we implement their functions?**
  - Use a combination of on-disk and in-memory structures
- **On-disk 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
  - Volume control block (superblock, 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
  - Per-file File Control Block (FCB) contains many details about the file
    - Permissions, inode number, file size, dates
    - NTFS stores info in master file table using relational DB structures

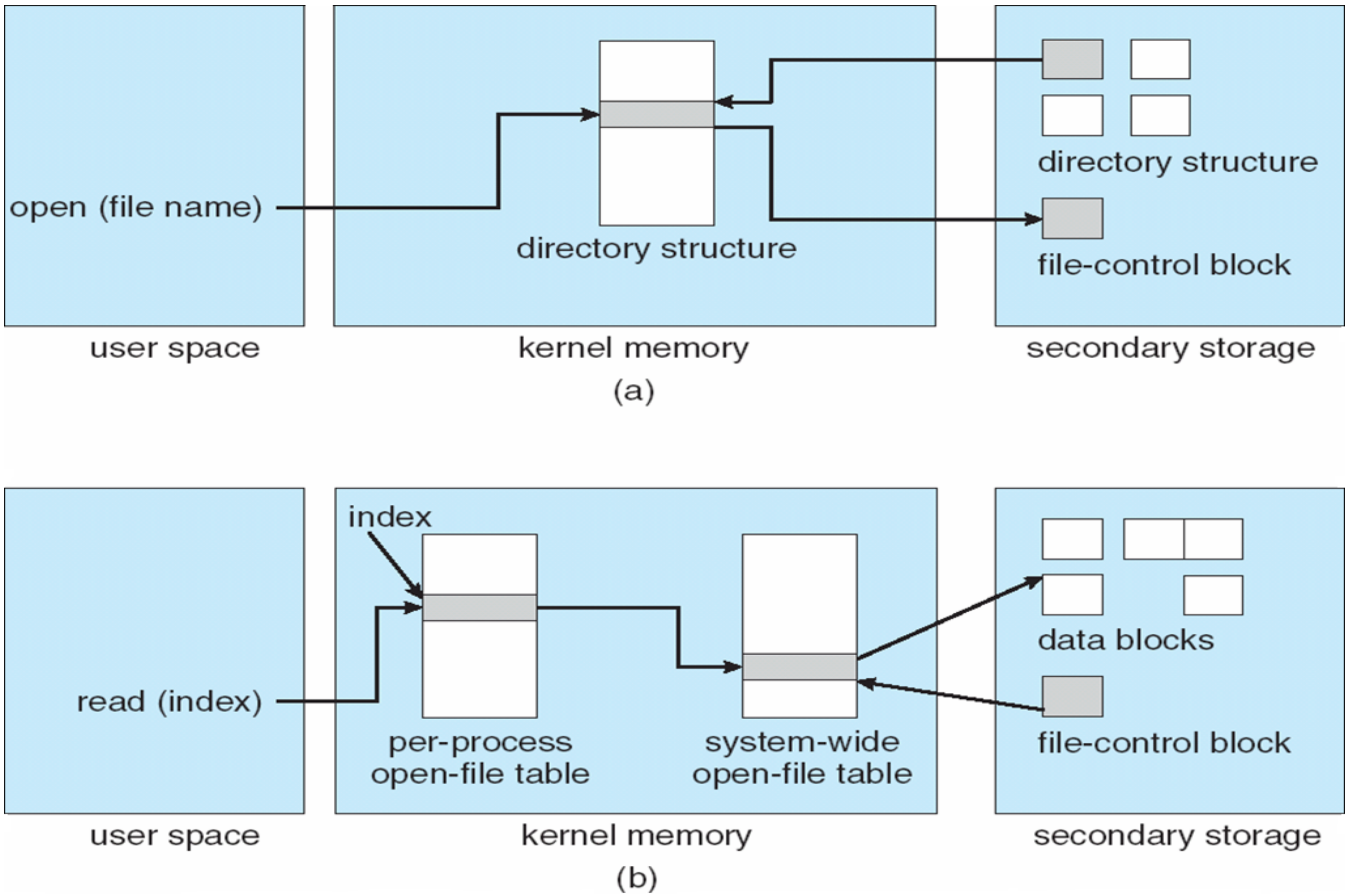
Typical File Control Block

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

- **In addition to on-disk structures, several file system structures are maintained in memory as well**
  - A mount table that stores file system mounts, mount points, file system types
  - Cached portions of the directory structure
  - A system-wide open-file table that contains a copy of the FCB for each open file
  - A per-process open-file table that contains a pointer to the appropriate entry in the system-wide open-file table
  - Buffers for assisting in the reading/writing of information from/to disk

In-Memory File System Structures



Partitions and Mounting

- **Partition can be a volume containing a file system or simply contain raw data – just a sequence of blocks with no file system**
- **A boot block can point to the boot volume or boot loader blocks that contain enough code to know how to load the kernel from the volume**
  - Might load a boot manager rather than a kernel if multiple multi-booting
  - The boot block doesn’t adhere to the file-system of the operating
    - Need to read boot block before OS is even loaded!
    - Typically, boot block is just loaded into memory and executed
- **A root partition contains the OS and is mounted at boot time**
  - Other partitions may also have file-systems and can mount automatically or manually
- **At mount time, the file system consistency checked**
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

Directory Implementation

- **Directory maintains a symbolic list of file names with a pointers to the data blocks**
- **Different algorithms can be used for 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 (use chaining for collision resolution)

Allocation Methods

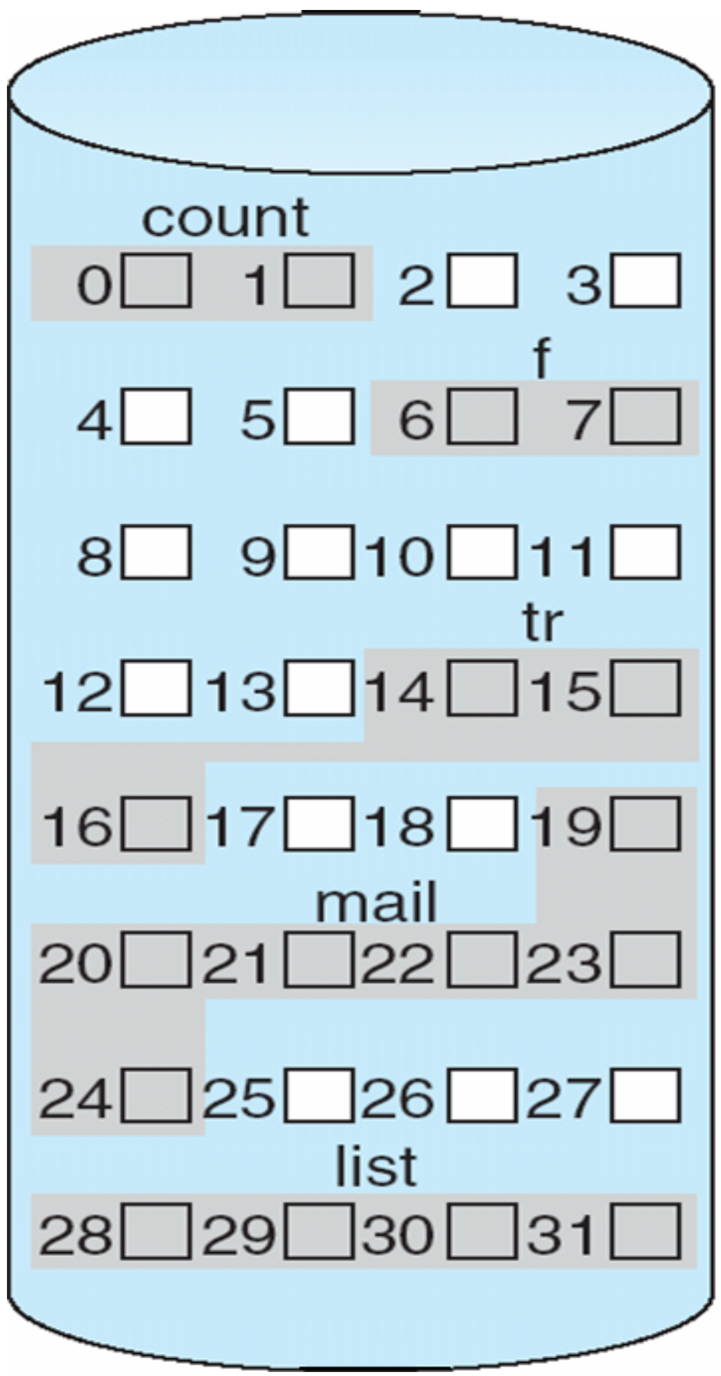
- **An allocation method refers to how disk blocks are allocated to files**
  - Want to utilize disk space efficiently
  - Want files to be accessed quickly

- **Three main methods currently used for disk allocation**
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation

Allocation Methods - Contiguous Allocation

- **Contiguous allocation – each file occupies a set of contiguous blocks on disk**
  - Best performance in most cases (minimal seek time)
  - Simple – the directory entry for each file need only contain the starting location (block #) and the length (number of blocks) for each file
  - Supports both sequential and direct access
  - Problems with contiguous allocation include:
    - Finding space for new files
    - Must know file size at time of creation
    - External fragmentation
    - Need for compaction off-line (downtime) or on-line

Contiguous Allocation of Disk Space

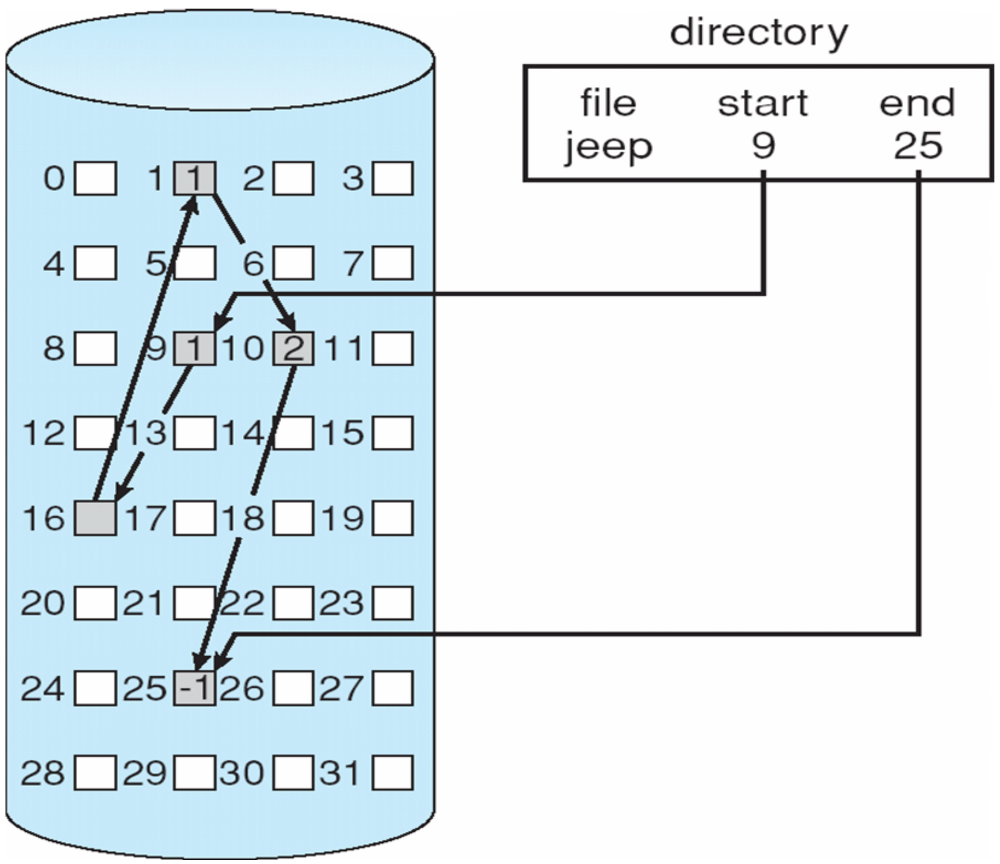


directory

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

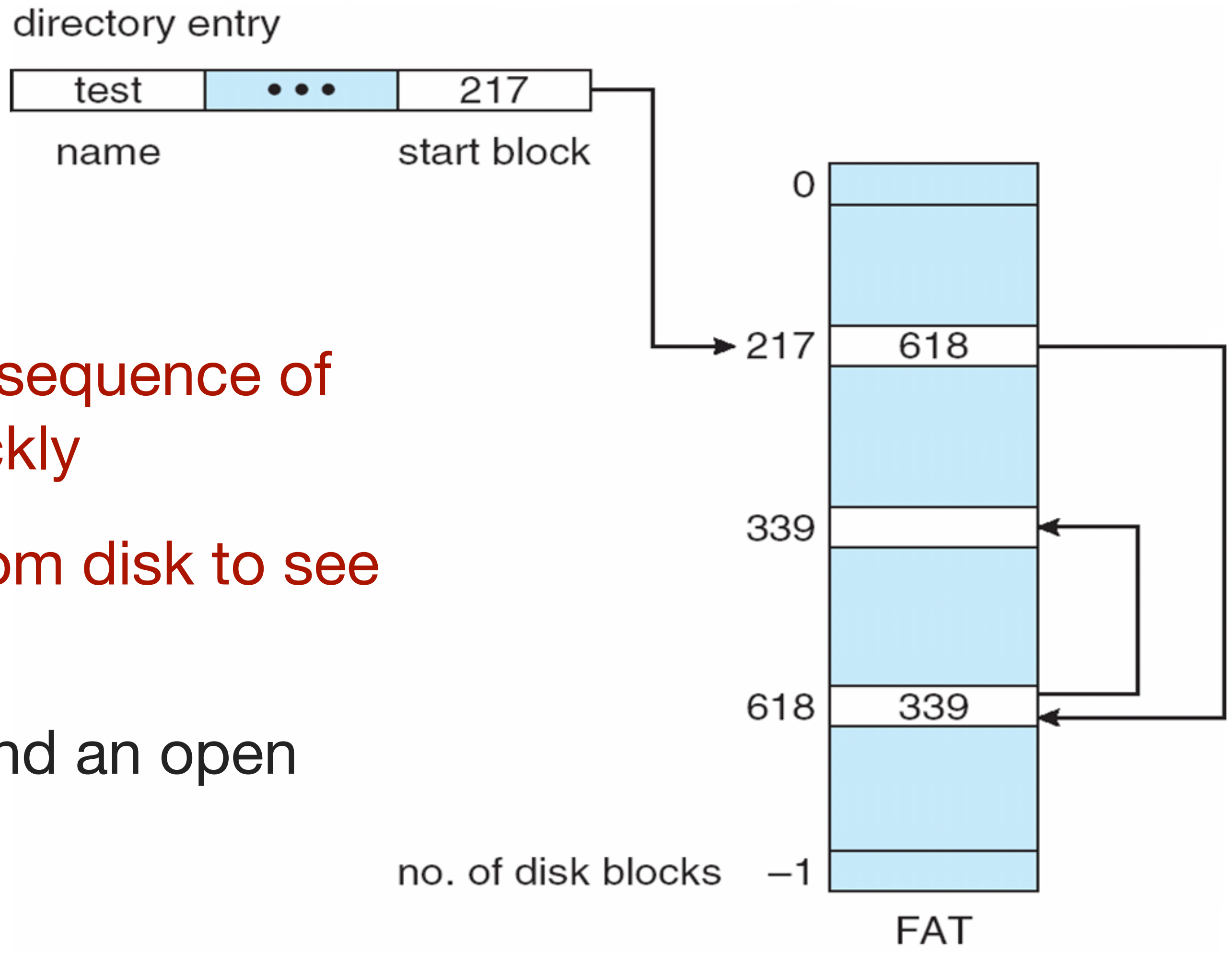
Allocation Methods - Linked Allocation

- **Linked allocation – each file is a linked list of blocks**
  - Each block contains pointer to next block
  - Free space management system called when new block needed
  - No external fragmentation, thus no compaction necessary
  - Can improve efficiency by creating clusters of blocks
    - This increases internal fragmentation
  - Reliability can be a problem
    - What happens if one of the links is corrupted?



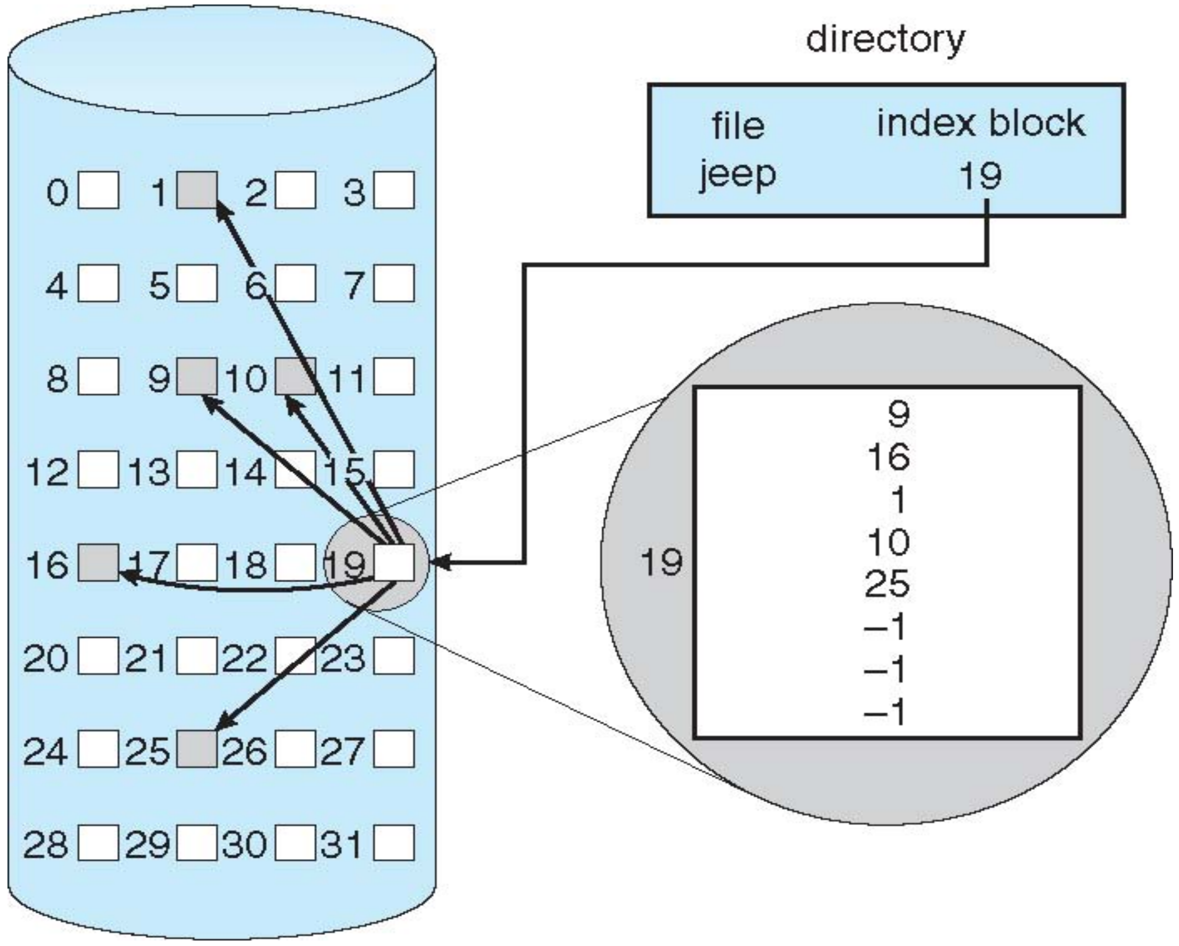
Allocation Methods - Linked Allocation with FAT

- **FAT (File Allocation Table) variation**
  - Beginning of a FAT volume has table, indexed by block number
  - Linked list of block numbers for files
    - Can be cached in memory so that sequence of blocks can be read from more quickly
    - Don't have to wait to read block from disk to see where next block is located
  - New block allocation simple, simply find an open slot in FAT



Allocation Methods - Indexed Allocation

- **Indexed allocation – each file has its own index block (or blocks) of pointers to its data blocks**
  - Directory contains the address of the index block
  - Location i in the index points to block i of the file
  - Supports direct access (like continuous allocation) without external fragmentation
  - May waste disk resources
    - An entire index block must be created, even for small files that only require a few pointers



Allocation Methods - Indexed Allocation (Cont.)

- **How large should the index be?**
  - If too large, then wasting a lot of space (*every* file has an index block)
  - If too small, then may not have enough space to index all of the blocks required for very large files

- **Several schemes are available to allow index blocks to be small enough so as not to be too wasteful, but 'expandable' so large files can be handled**

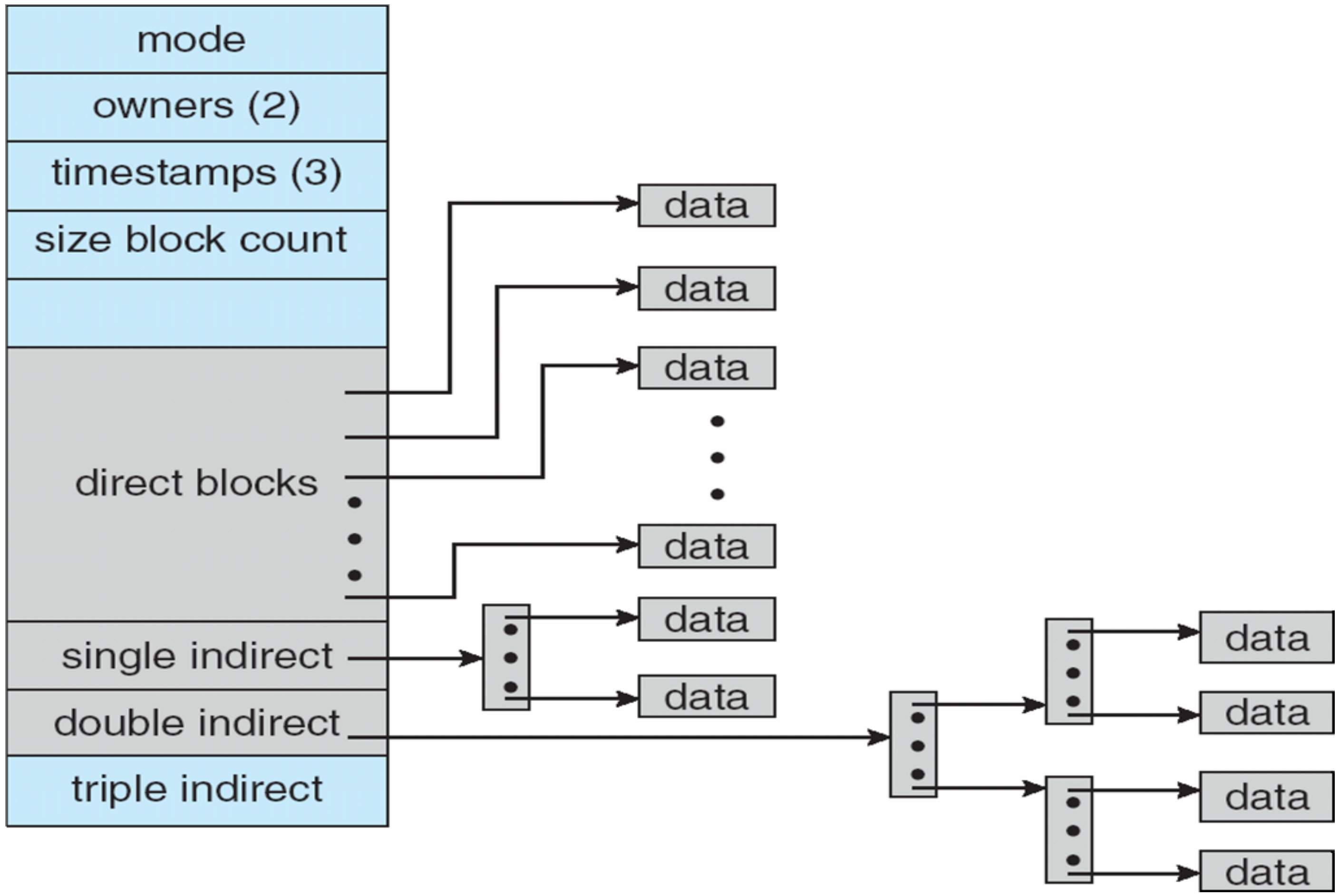
Allocation Methods - Indexed Allocation (Cont.)

- **Linked scheme - allow index blocked to be linked together to handle large files**
  - If the file is larger than can be handled by a single index block, the last word of an index block is used to store a pointer to the next linked block
  - If the file is small enough so as to require only a single index block, then the last word of the index block is a null pointer

- **Multilevel index - a first-level index block points to a set of second-level index blocks which, in turn, point to the data blocks for the file**
  - A tree structure of index blocks that can be expanded by adding additional levels

Allocation Methods - Indexed Allocation (Cont.)

- **A Combined scheme - reserves some space in the first index block to point directly to data block, while others point to multilevel indexes**
  - For small files only the first index is required which points directly to data block
  - Larger files may require some single-indirect indexes
  - Even larger files may require double or triple-indirect indexes



Free-Space Management

- **File system maintains free-space list to track unallocated blocks/clusters**

- **Free-space list can be implemented in a variety of ways**
  - Bit Vector
  - Linked List
  - Linked List with Grouping
  - Contiguous Block Counting

Free-Space Management - Bit Vector

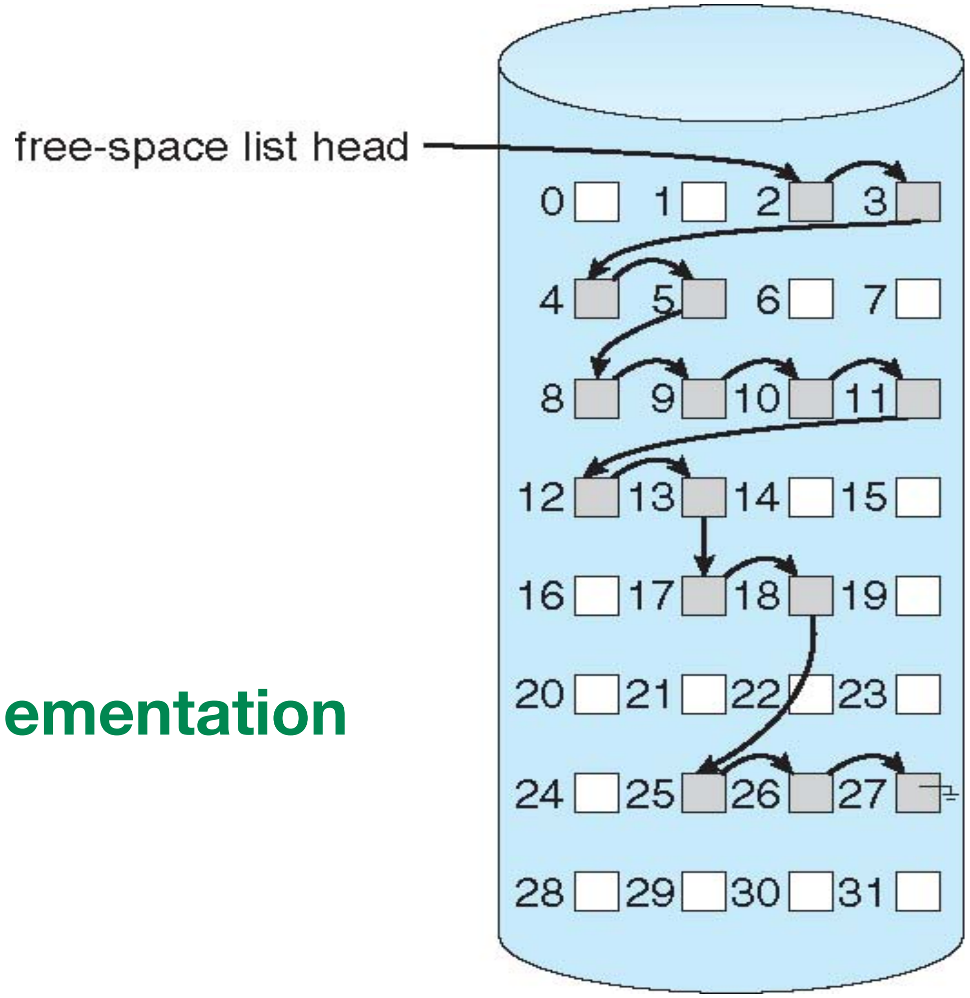
- **Utilize a bit vector in which each bit represents a block**
  - A ‘1’ represents a free block, a ‘0’ represented a block that has already been allocated
- **When looking for a free block, find the first ‘1’ in the bit vector**
  - Most modern CPUs have a single cycle instruction for this that operates on words
- **Bit vector requires extra space for storage**
  - May need to keep in main memory for performance (write to disk occasionally for permanent storage)

Free-Space Management - Linked List

- **A linked list can be used to maintain the free-list**
  - A head pointer identifies the first free block
  - Each block contains a pointer to the next free block

- **When a block is needed, simply get a block from the head of the linked list**
  - Cannot get contiguous space easily since free blocks are spread across disk

- **No wasted space like with the bit vector implementation**



Free-Space Management - Linked List with Grouping

- **In the linked list implementation, getting a large group of free blocks requires traversing the linked list to find each**

- **Rather than having only a reference to a single unallocated block in each linked list node, each node can contain a list of free blocks**

- **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)**

Free-Space Management - Contiguous Block Counting

- **Space is frequently used and freed contiguously, with contiguous-allocation or clustering**
    - When freeing a contiguous section of blocks, no need to maintain information about all of them
    - Keep address of first free block in a contiguous section of free space and a count of following free blocks
    - Free space list then has entries containing addresses and counts

Recovery

- **Crashes, bugs, power outages, etc. may leave the file system in an inconsistent state**
  - Example: FCB is written for a new file, but file isn't added to the directory

- **Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
  - One possibility is to scan the metadata for the file system to search for inconsistencies
    - Can be slow and sometimes fails
  - Associate a status bit with each file, set that bit prior to making changes to the file, unset that bit only when all changes are complete
    - If any status bits are set when checking for consistency, then a correction may be needed

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**
  - 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**