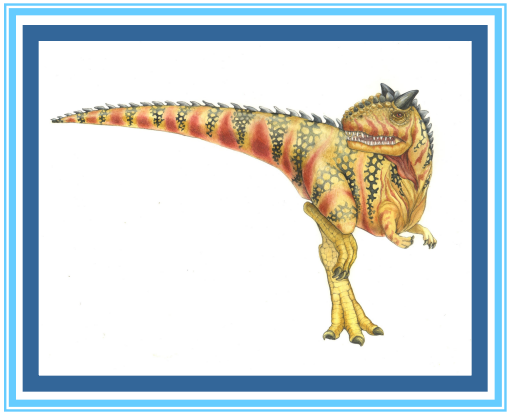
File System Implementation



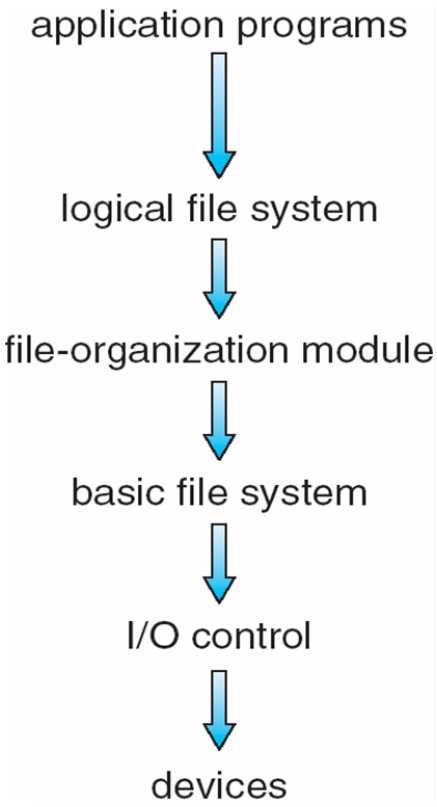
Background: Disk addressing

- We have hard drive/ssd/thumb drives in our computer and we want to use them via File systems
  - Very old way: physical addressing,
    - to get the data the hard drive need to go to 5th track of 20th sector
    - OS needed to know physical properties of the drive
    - Varies from hdd to ssd to any other technology
    - There is need for abstraction
  - Disk exports its data as a logical array of blocks [0…N]
  - These blocks are the smallest unit of transfer
  - This is called Logical Block addressing (LBA)
  - We will call these blocks on secondary storage as “physical blocks”

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 system organized into layers

Layered File System



- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device

File-System Implementation

- 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
- **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

File-System Implementation (Cont.)

- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structures

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

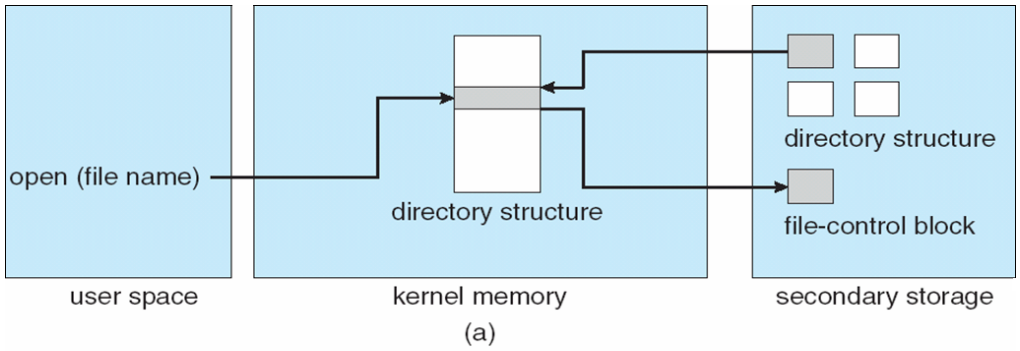
How does create() work?

- data structures needed (for all operations):
  - An in-memory mount table (FS mounts, mount points, FS types)
  - An in-memory directory-structure cache for recently accessed directories
  - system-wide open file table
  - per-process open file table
  - Buffers to hold file-system blocks when they are being read from disk or written to disk

- Mechanism for create():
  - Application calls logical file system which creates a FCB / get a free FCB
  - OS reads corresponding (to that FCB) directory in memory
  - updates it with the new file name and FCB, and writes it back to the disk

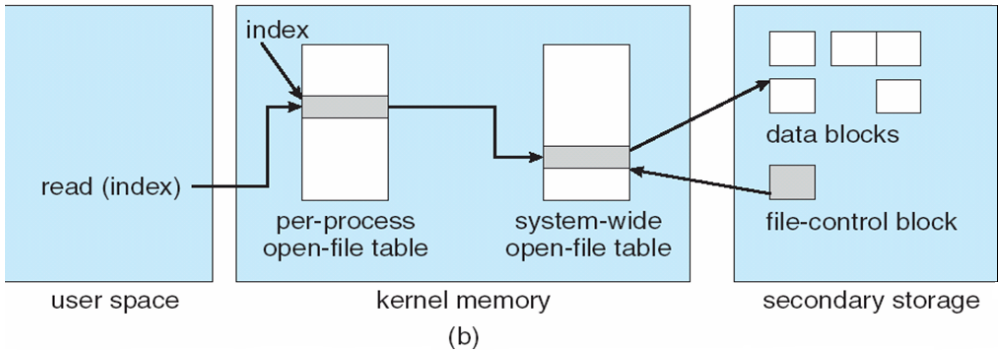
How does open() work?

- Mechanism for open():
  - open() call passes a file name to the logical file system
  - Searches system-wide open-file table to check if the file is already in use by another process
    - Yes: a per-process open-file table entry created pointing to existing system-wide open-file table
    - No: directory structure is searched for the given file name (use directory caches). Once file is found, FCB is copied into a system-wide open-file table (with #process that called open()). Then same as the "Yes" case.
  - return a pointer to entry in per-process open-file table (the “file descriptor”)



How does read() work?

- Mechanism for read():
  - read() call passes the file descriptor (index in per-process open-file table)
  - The per-process open-file table entry at index points to the system-wide open-file table entry
  - The system-wide open-file table entry contain the FCB
  - Pass this information to the file-organization module and the layers below



(b)

How does close() work?

- Mechanism for close():
  - per-process open-file table entry removed
  - system-wide open-file table-entry's open count is decreased by 1
  - If the system-wide open-file table-entry's open count is 0
    - updated metadata is copied back to secondary storage directory
    - the system-wide open-file table entry is removed

Generalized principles

- open(), close(), read(), write() etc. are nice understandable system calls
  - OS like Unix/Linux provide similar system calls also for other things like networking or basically any device your connect
  - system-wide open-file table-entries are not only FCB (i.e., inode) but also similar information for network connections and devices
  - Thus the adage: "Everything is a file" in Unix/Linux
  - More: https://en.wikipedia.org/wiki/Everything_is_a_file
- Caching in FS is super important
  - Most OS keep *all* information about an open file in memory, except for the actual data blocks
  - in-memory buffer caches for both read() and write()
  - Average FS cache hit rate of Unix is 85%!

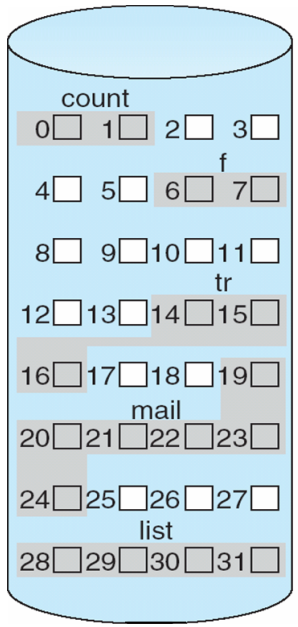
Directory Implementation

- **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 1 - Contiguous

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
  - Best performance in most cases
  - Simple – only starting location (block #) and length (number of blocks) are required

Contiguous Allocation: Schematic



directory

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

Contiguous Allocation: Mapping

- Mapping from logical address (LA) to physical
  - Logical address: relative to file, e.g., access LA'th byte in the file
  - Physical address: the actual physical block number on storage

$$Q = LA \,//\, 512$$

$$R = LA \bmod 512$$

Physical Block to be accessed = Q + starting address
Displacement into block = R

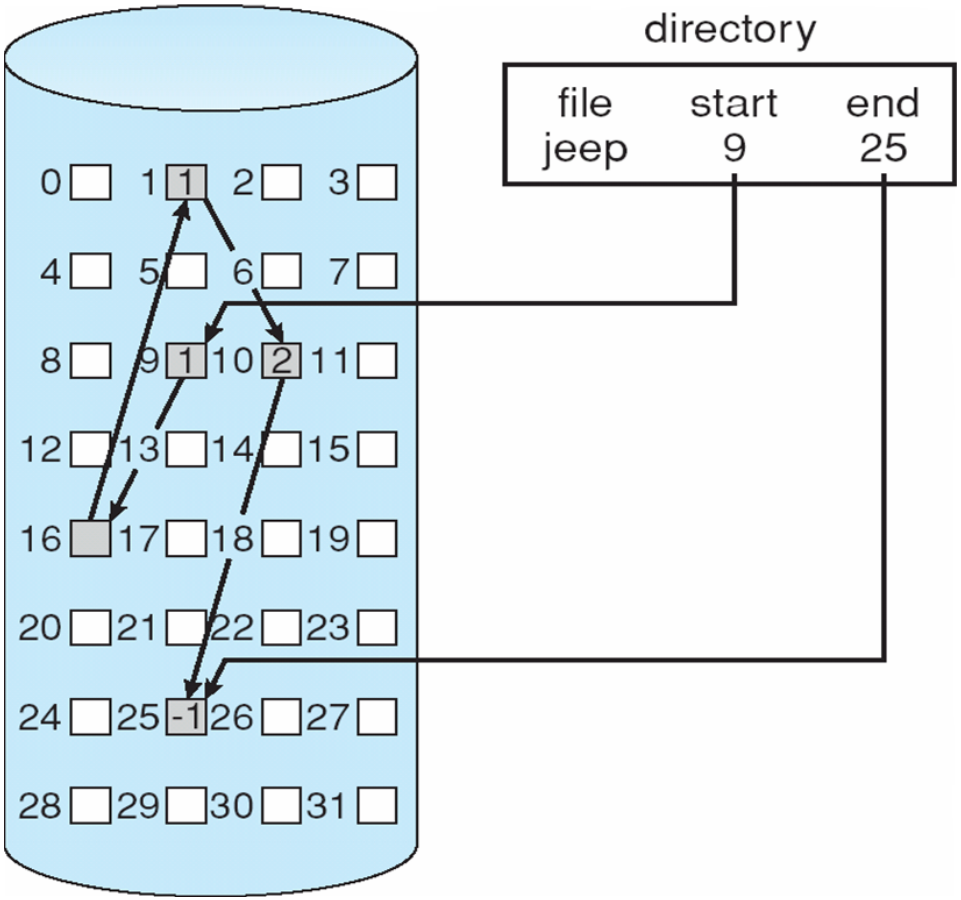
Problems with contiguous allocation

- **Contiguous allocation** – problems
  - Finding space for file
    - external fragmentation,
    - Need for **compaction off-line** (**downtime**) or **on-line**
  - Knowing size of the file
    - Size of file might increase after creation
    - However there might be no contiguous space to increase
    - Terminating programs is costly
    - Overestimation of space – still costly
    - Find another larger space and copy current content – slow

Allocation Method 2 - Linked

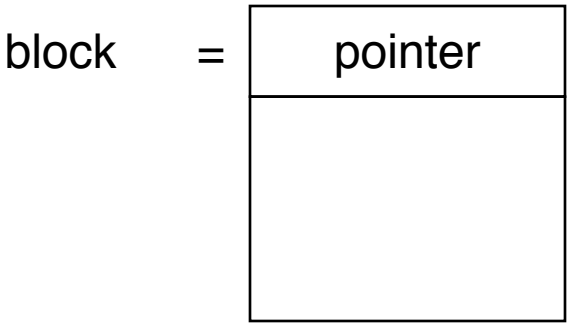
- **Linked allocation** – each file a linked list of blocks
  - File ends at nil pointer
  - No external fragmentation
  - Each block contains pointer to next block
  - Free space management system called when new block needed
  - No compaction, external fragmentation
  - Issues
    - Linear search -- Locating a block can take many I/Os and disk seeks
    - Reliability
  - Improve efficiency by clustering blocks into groups but increases internal fragmentation

Linked Allocation



Linked Allocation

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



- Mapping from logical to physical

$$Q = LA \,//\, 508$$

$$R = LA \bmod 508$$

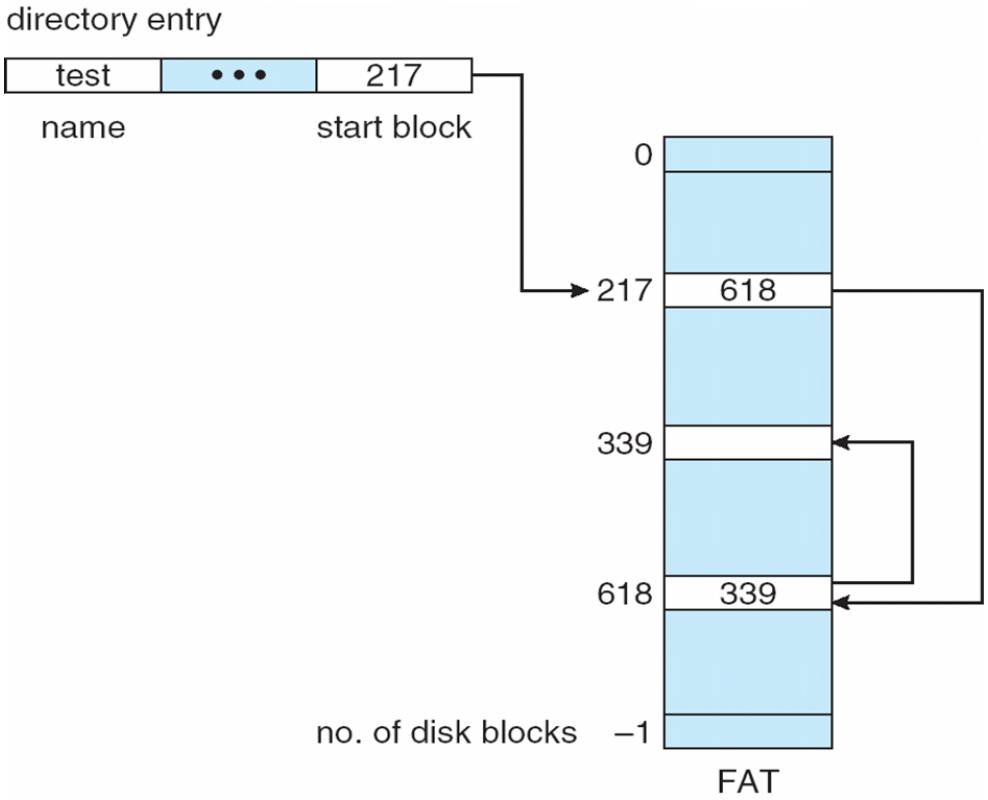
Block to be accessed is the Qth block in the linked chain of blocks representing the file.

Displacement into block = R + 4

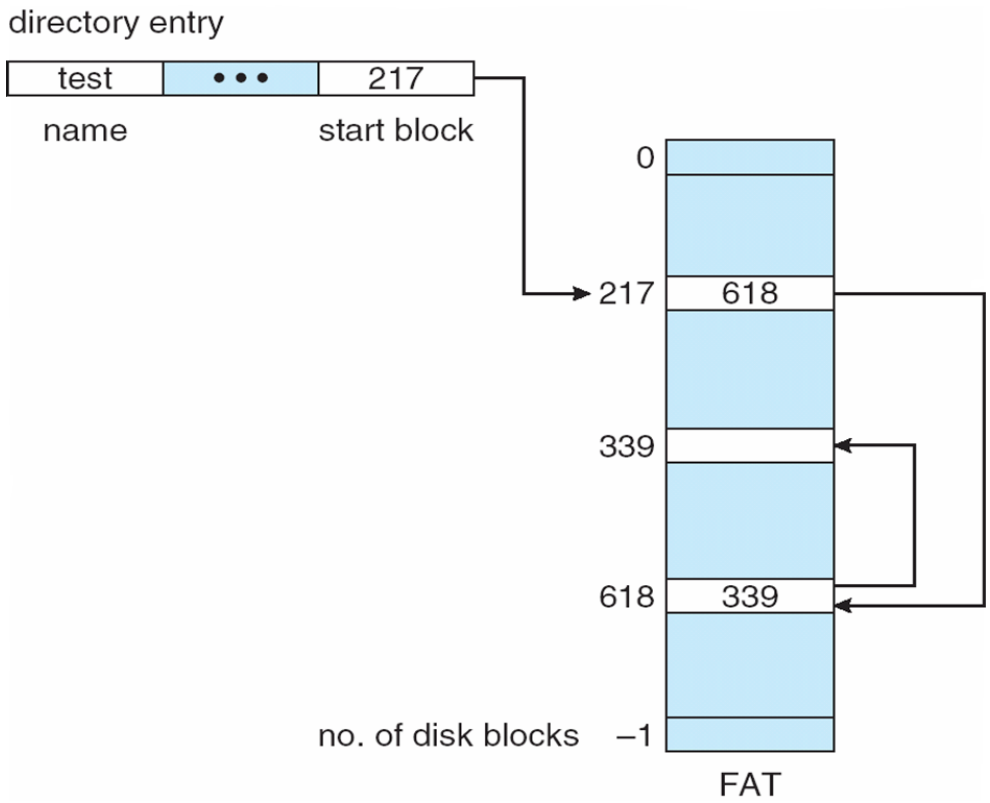
Allocation Methods – Linked (Cont.)

- FAT (File Allocation Table) variation
  - Beginning of volume has table, indexed by block number
  - Much like a linked list, but faster on disk and cacheable
  - New block allocation simple

File-Allocation Table



File-Allocation Table

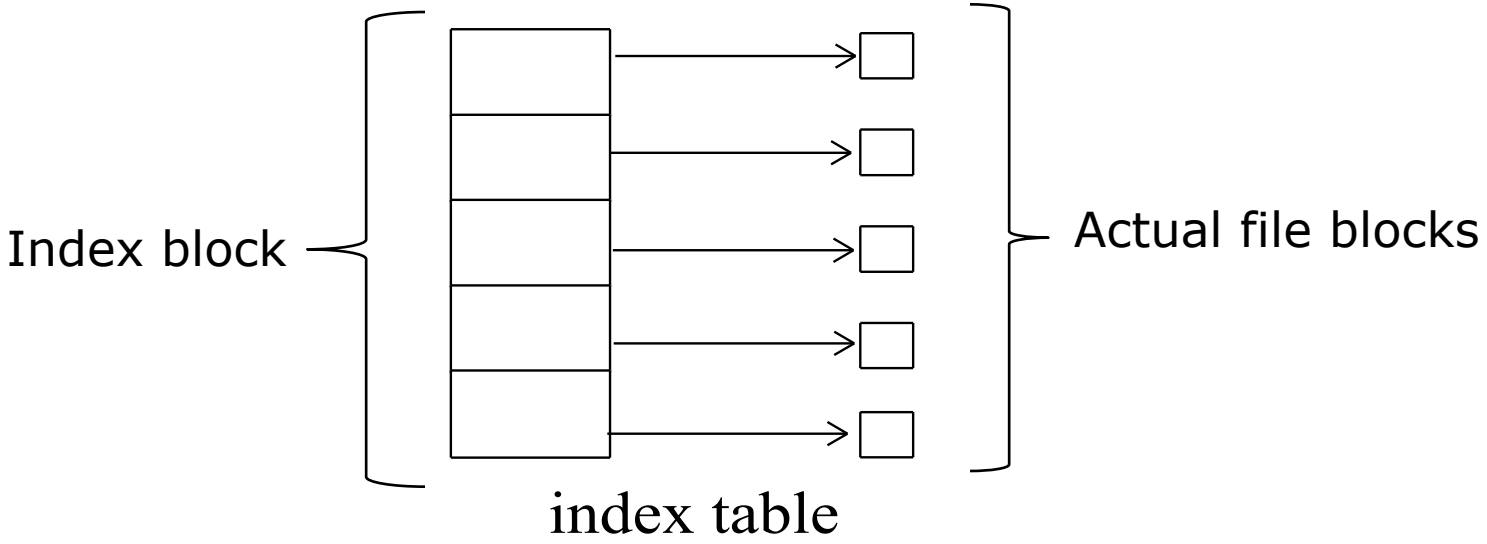


Question: you have a 512 GB hard disk and each block size is 4 KB. If your File system contains FAT, what is the smallest amount of memory used for FAT?

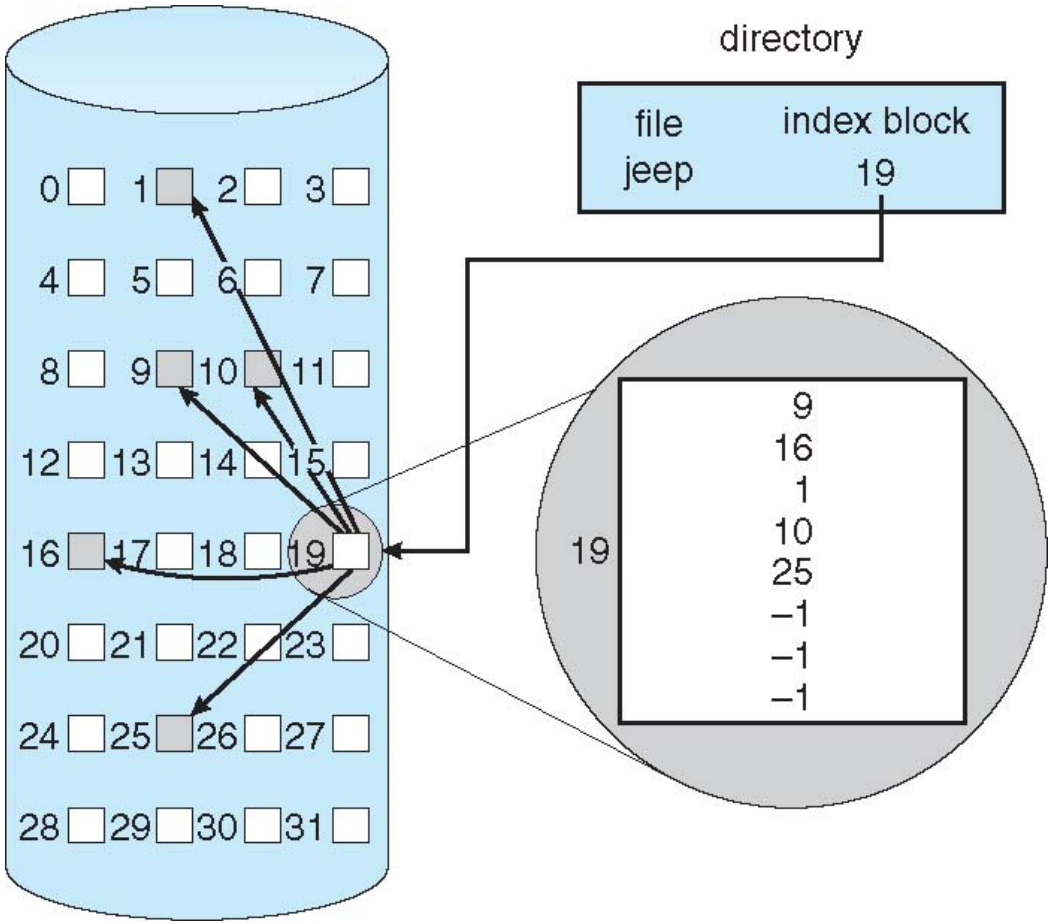
Allocation Method 3 - Indexed

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

- Logical view



Example of Indexed Allocation



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

$$Q = LA \,//\, 512$$

$$R = LA \bmod 512$$

Q = displacement into index table
R = displacement into block

Indexed Allocation : Linked scheme

- Create an index block as a disk block
- Then if necessary add a pointer to the end to another new index block
- Linked scheme – Link blocks of index table (no limit on size)

Indexed Allocation : Multilevel index

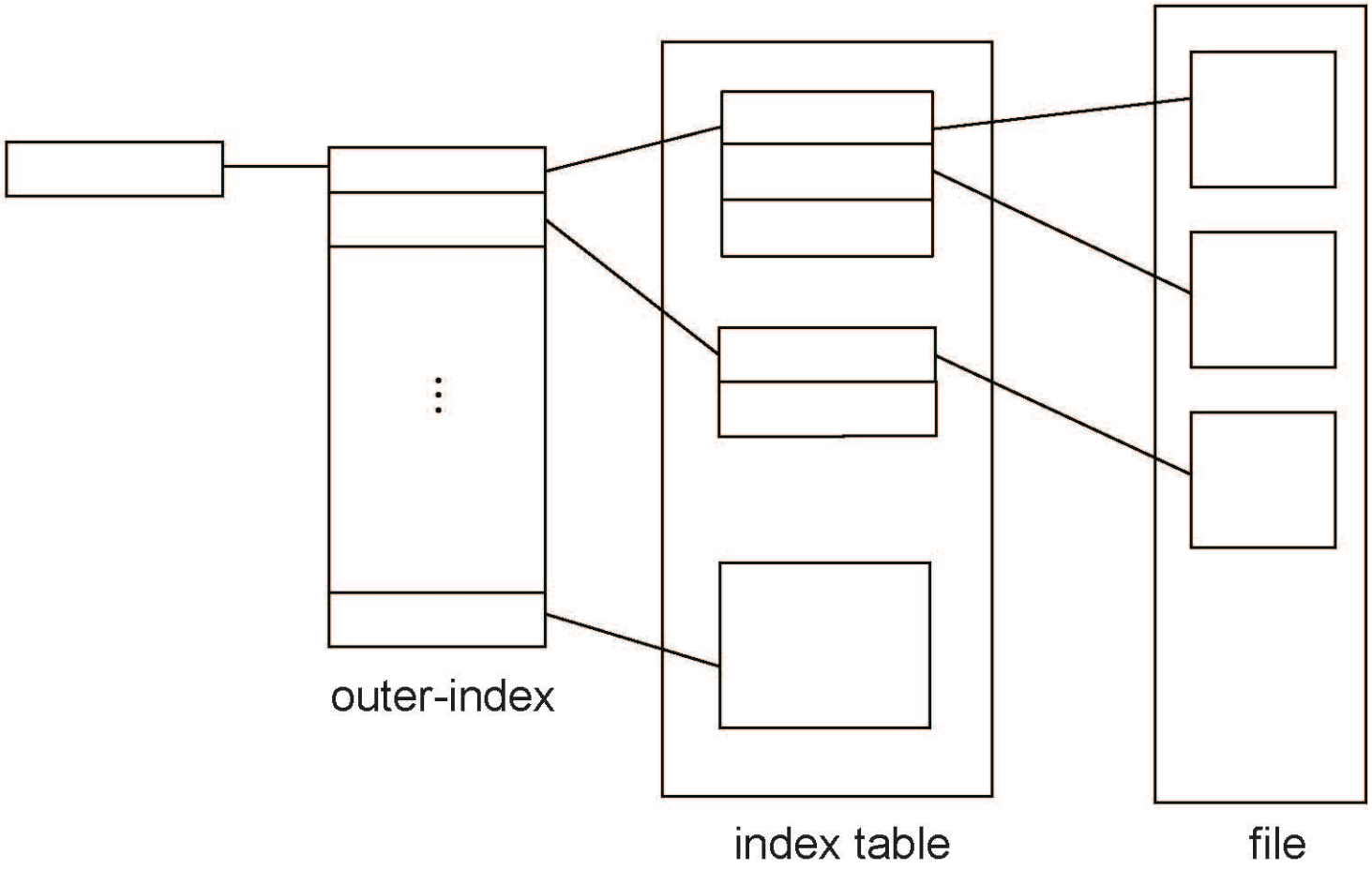
- First-level index block is a disk block which holds address for second-level index block
- If level = 2, then second-level index points to data blocks
- What is the maximum size of files for two level index with 4kB sized blocks and 4 byte pointers?
  - 4kB block of first index= 4 x 2^10 bytes = 2^10 pointers
  - Each pointer points to a second-level index block
  - So, total #entries in all second level tables = 2^10 x 2^10
  - Each entry in each second level table points to a 4 KB block
  - So maximum file size = 2^10 x 2^10 x 4 KB = 4 GB

Indexed Allocation : Multilevel index

- First-level index block is a disk block which holds address for second-level index block
- If level = 2, then second-level index points to data blocks
- What is the maximum size of files for two level index with 4kB sized blocks and 4 byte pointers?
  - 4kB block of first index= 4 x 2^10 bytes = 2^10 pointers
  - Each pointer points to a second-level index block
  - So, total #entries in all second level tables = 2^10 x 2^10
  - Each entry in each second level table points to a 4 KB block
  - So maximum file size = 2^10 x 2^10 x 4 KB = 4 GB

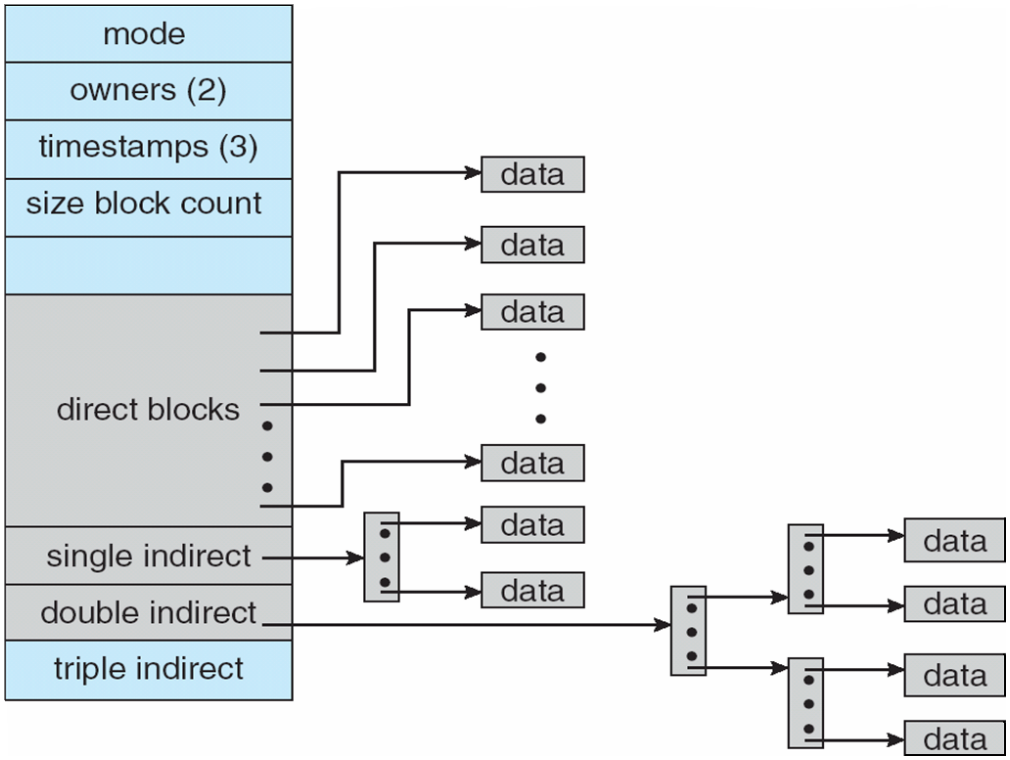
- Exercise: What would be the max. file size for n level index with 4m byte block sizes and 4 byte pointer sizes? [Ans: 4 x m^(n+1)]
- Exercise: Find the logical file specific byte to physical block mapping in this two–level scheme. Assume a block size of 512 bytes and pointer size of 4 bytes. [Hint: First find the block number in 2nd index table]

Indexed Allocation : multilevel index



Indexed allocation: Combined Scheme:

Used in UNIX
4K bytes per block, 32-bit addresses



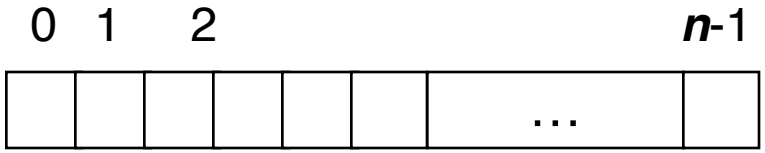
More index blocks than can be addressed with 32-bit file pointer

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
  - A hybrid is more preferable: contiguous for small files and indexed for large files

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)



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

Block number calculation

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

CPUs have instructions to return offset within word of first "1" bit

Free-Space Management (Cont.)

- Bit map requires extra space
  - Example:

    block size = 4KB = 2^{12} bytes

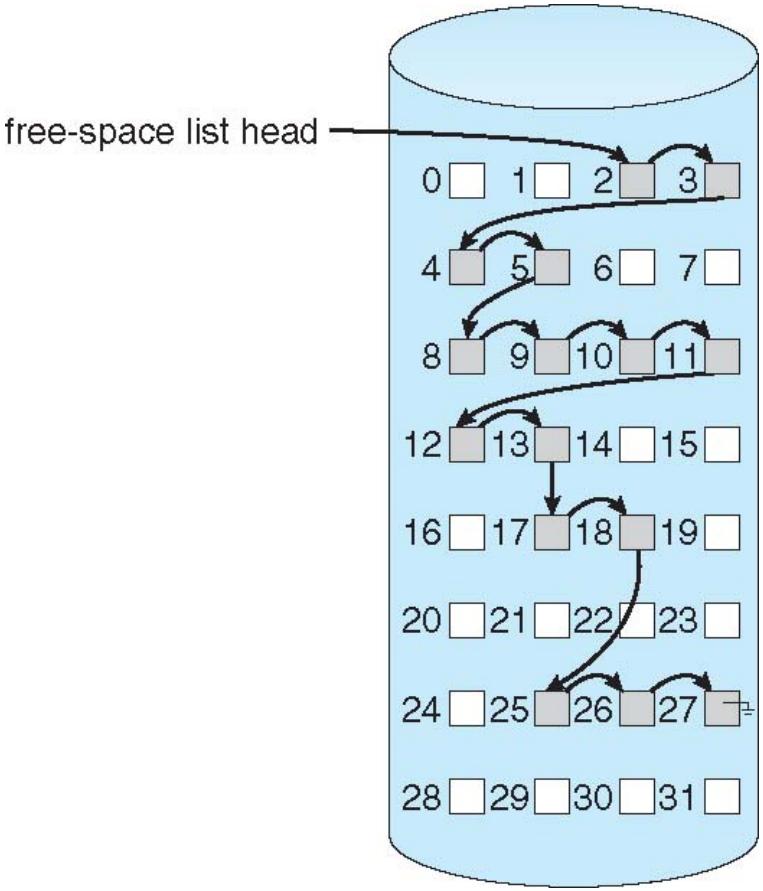
    disk size = 2^{40} bytes (1 terabyte)

    $n = 2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

    if clusters of 4 blocks -> 8MB of memory
- Easy to get contiguous files

Linked Free Space List on Disk

- Linked list (free list)
  - No waste of space
  - Expensive to traverse the list
  - No need to traverse the entire list (if # free blocks recorded)



Efficiency and Performance

- Performance
  - **Buffer cache** – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - **Free-behind** and **read-ahead** – techniques to optimize sequential access
  - Reads frequently slower than writes