File System Implementation

Disk partition, volume, and mounting; file storage allocation methods; contiguous allocation; linked allocation and FAT; indexed allocation; multiple-indexed allocation and UFS

OS7: 1/3/2016
Textbook (SGG): Ch. 11.2.1-11.2.2; 11.4
(Assigned reading for this module will *not* be included in the quiz on Feb 29.)

Adding new disks and volumes

- You buy a new disk
- You format it into one or more partitions, each with a file system
  - File system is OS specific
  - Windows: NTFS; Mac OS/X: HFS Plus; FAT32 is compatible with both (but loses some performance)
- Each partition with a file system is called a *volume*
- You mount each volume onto a larger file system name space (directory structure)

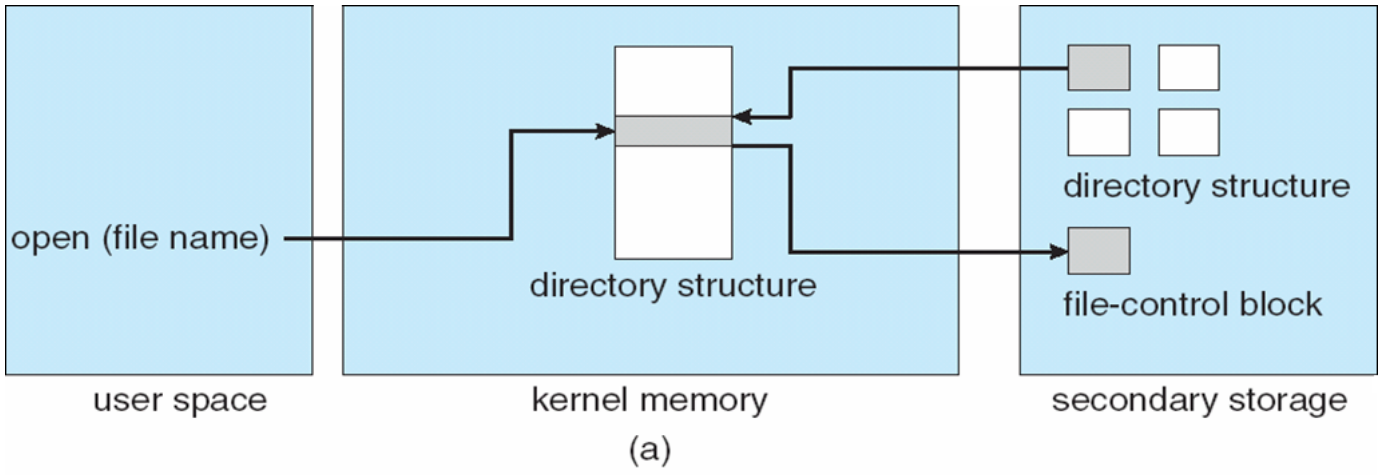
Disk blocks in partition

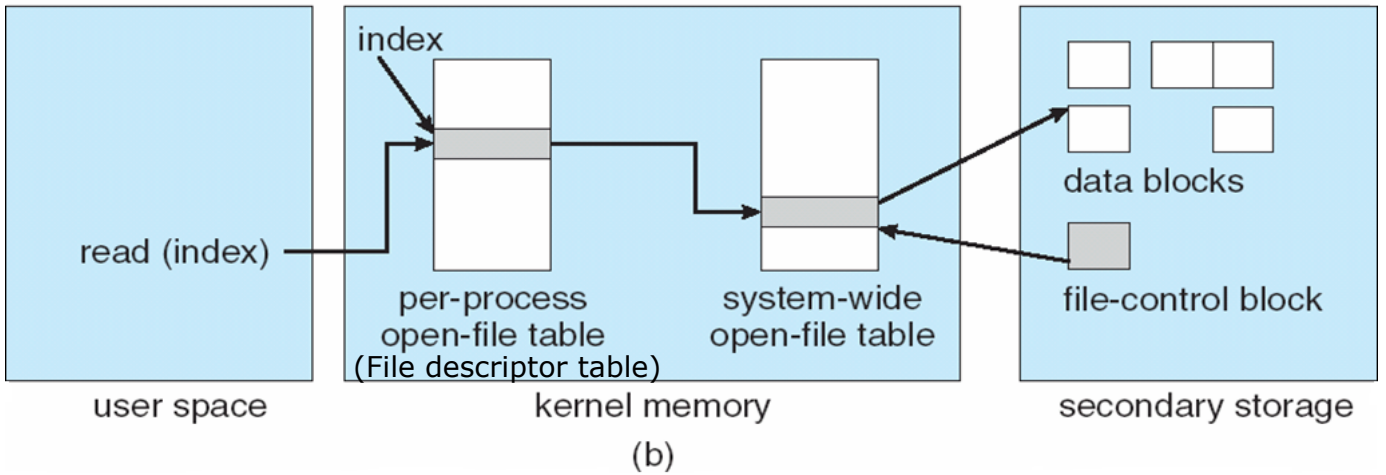
- **Boot control block** contains information needed by system to boot OS from that volume (“bootable” volume)
- **Volume control block** contains volume details
- Directory structure organizes the files
- Per-file **file control block** (**FCB**) stores attributes of each file

File control block of file attributes

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

File system data structures





Allocation of disk blocks to files

- Allocation methods
  - *Contiguous* allocation
  - *Linked* allocation
  - *Indexed* allocation
- Considerations
  - File size & need for dynamic allocation of file storage
  - File access method
  - Accessing sequential disk blocks is fast; jumping to a random disk block is slow
    - Involves *mechanical* movement (rotate the disk, then seek to disk sector)

System parameters

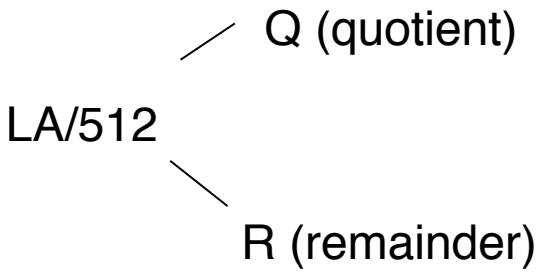
- Disk block size = 512 words (e.g., word size = 4 bytes)
- Each disk block is identified by a *block number* or *index*, whose size is one word
- Logical address (LA): offset of word in file to access (starting offset = 0)

Contiguous allocation

- Each file occupies a contiguous set of blocks on the disk
- Simple directory information – starting block number + length of file (in number of blocks)
- Random access is efficient
- Practical issues
  - Need to estimate file size at the beginning
  - Disk must have large enough “hole” (i.e., set of contiguous unallocated blocks) to accommodate the file
  - Can be hard to grow the file dynamically
  - Even if file has holes (i.e., no data) in the middle, disk blocks must be allocated for the holes
    - Problem of *internal fragmentation* – non-utilization of storage whose allocation status is *not* free

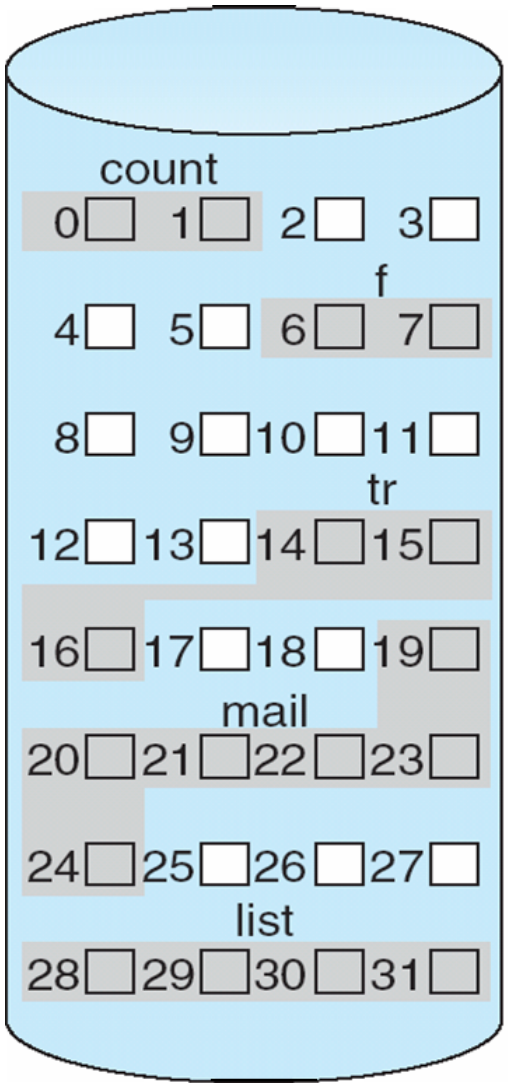
Contiguous Allocation

- Mapping from file word's logical address to physical disk location



- Disk block that contains word LA = Q-*th* block from the file's first block (Q = 0 for the first block)
- Offset of word LA in the disk block = R

Contiguous allocation of disk blocks



directory

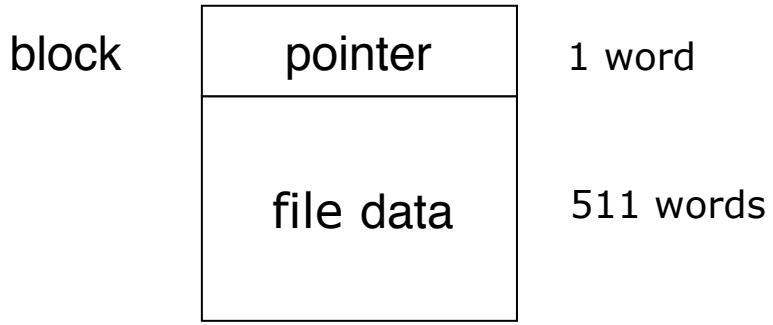
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Note: can't store a file that's 7 blocks long, although there are enough empty blocks

Problem is called *external fragmentation*: non-utilization of storage whose allocation status *is* free

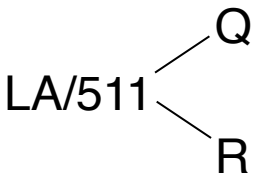
Linked allocation

- Each disk block has a pointer to the next disk block (null pointer for the last block)



NB: Pointer = index of next disk block

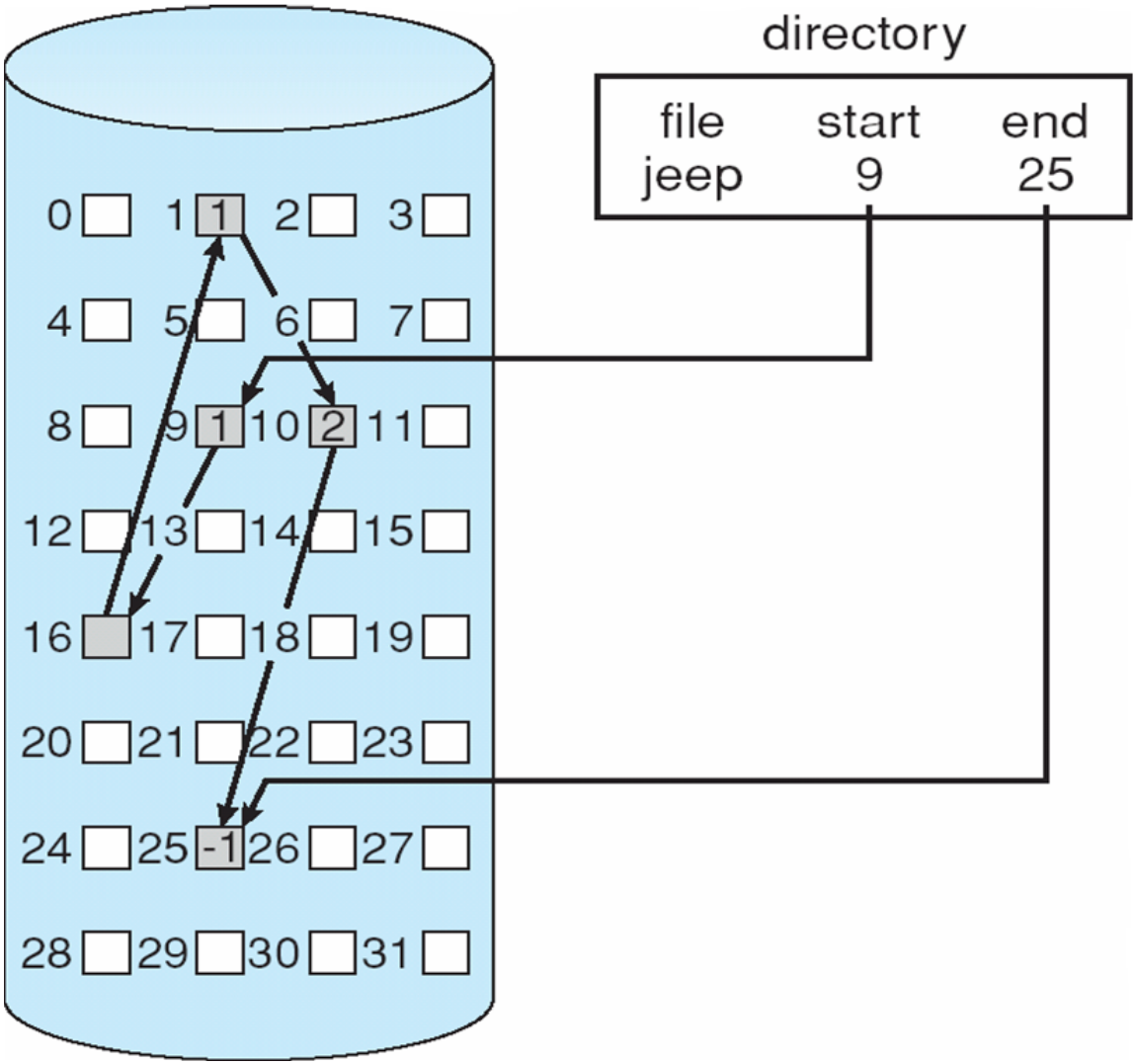
Linked allocation (cont'd)



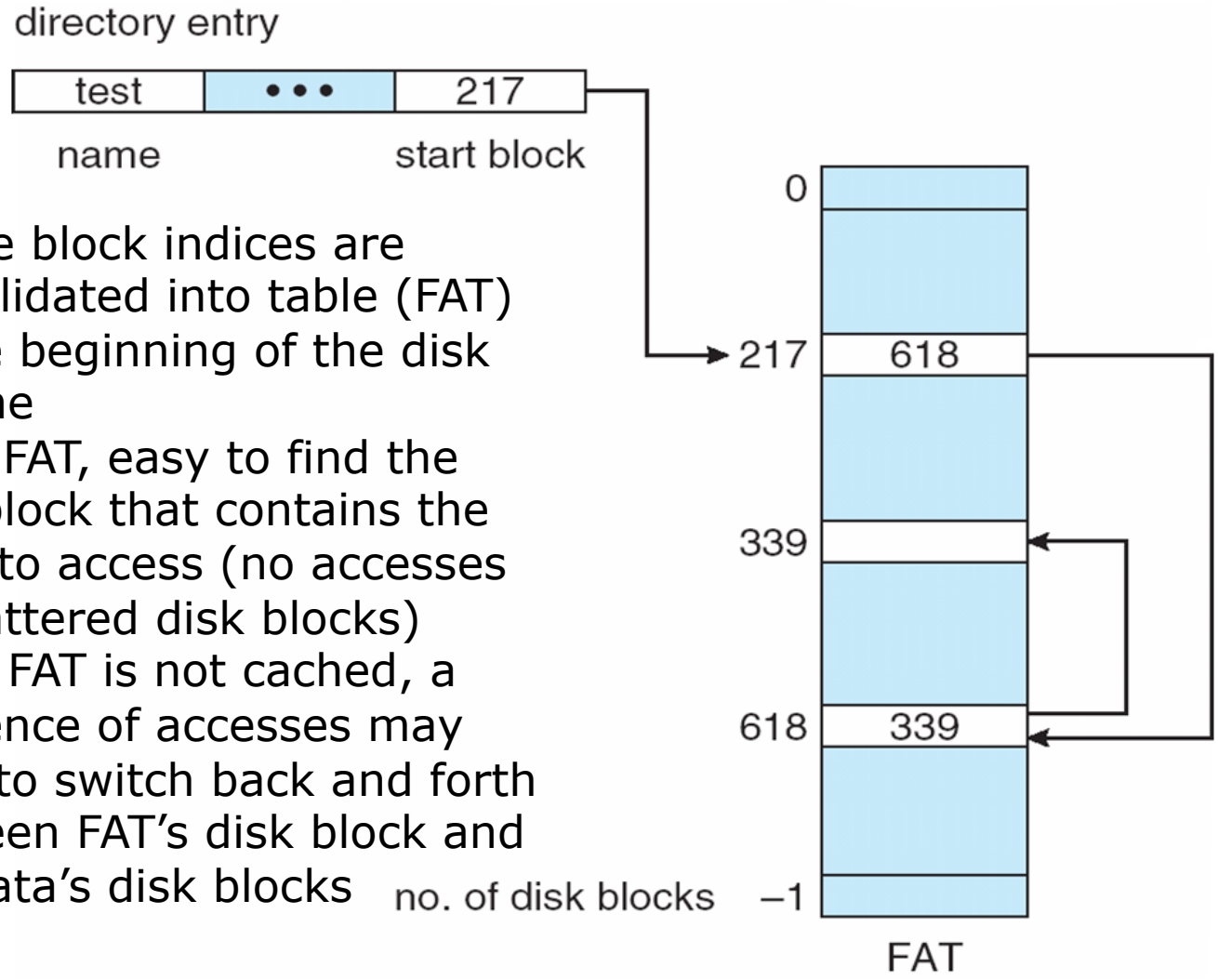
- Disk block that contains word LA = Q-*th* block in the file's linked list of blocks
- Offset of word LA in the disk block = R

- Simple directory information – starting disk block number only
- No added restrictions on file size
- Efficient use of free disk storage – no external fragmentation
- Random access is expensive – need to access many scattered disk block indices to find the required word
- Dynamic allocation of disk block generally requires traversal of linked list of (scattered) disk blocks as well
- Important variant
  - File-allocation table (FAT) in MS-DOS and OS/2

Linked allocation



File Allocation Table (FAT) in MS-DOS

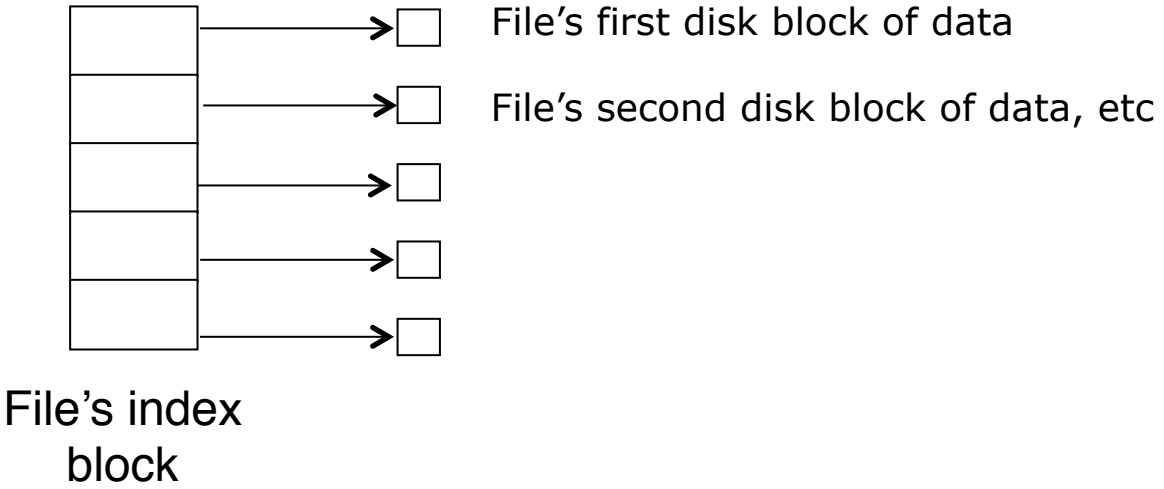


- All the block indices are consolidated into table (FAT) at the beginning of the disk volume
- From FAT, easy to find the disk block that contains the word to access (no accesses to scattered disk blocks)
- But if FAT is not cached, a sequence of accesses may have to switch back and forth between FAT's disk block and the data's disk blocks

System-wide table for the whole file system

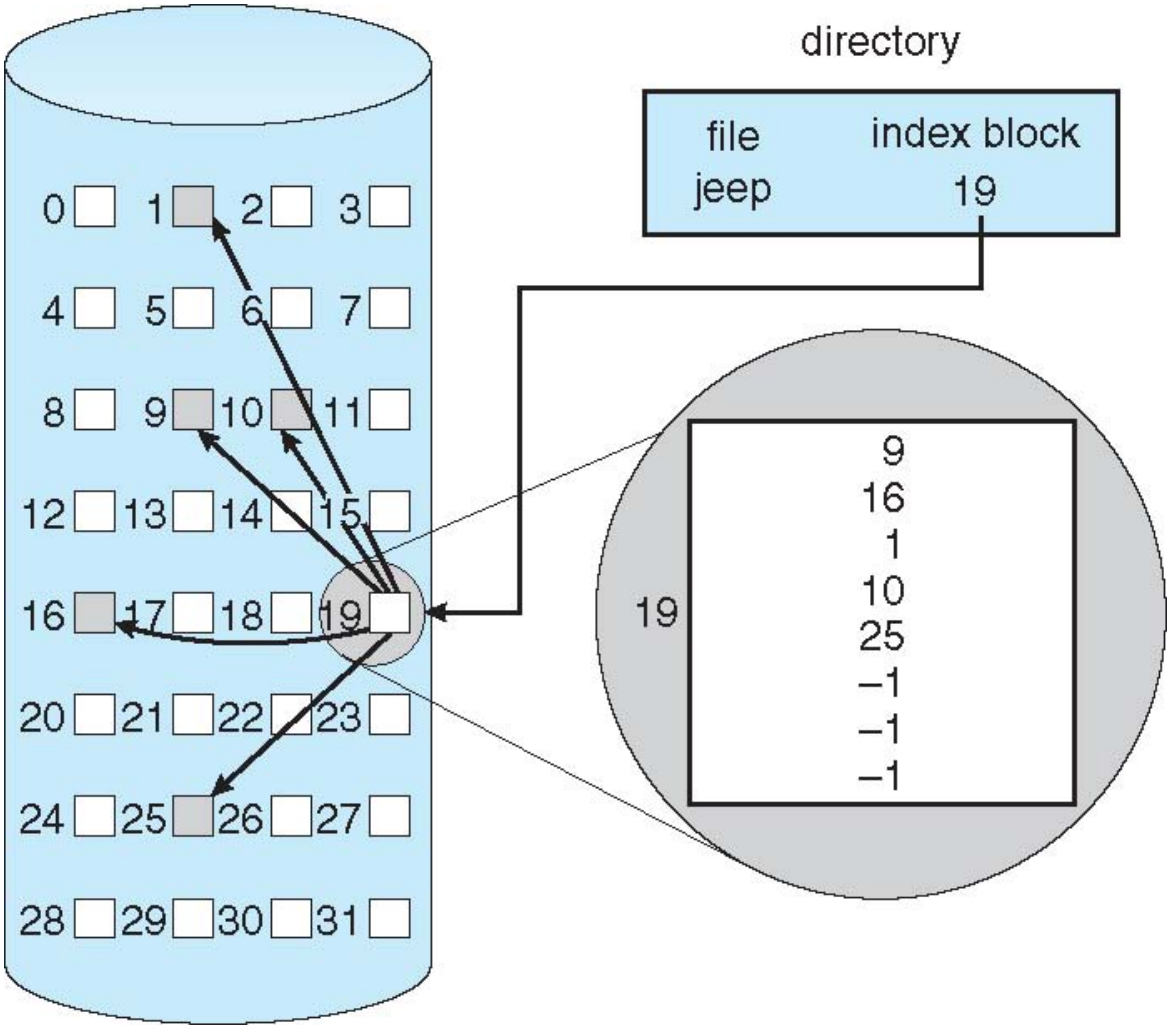
Indexed allocation

- Consolidate all pointers to a file's disk blocks into an **index block** that's *specific to the file*

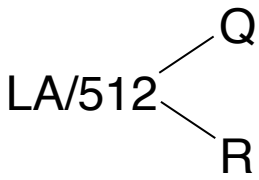


- Null pointers are okay anywhere – "holes" in files don't take up disk space

Example of indexed allocation



Indexed allocation (cont'd)



- Index of disk block that contains word LA = Q-*th* entry in index block
- Offset of word LA in the disk block = R

- Simple directory structure
- Pretty efficient direct access (assuming cached directory information, two disk block accesses at most)
- Easy to add/remove disk block of data (holes take up no disk space)
- No external fragmentation of disk storage
- Overhead of one full index block (even for small files)
- Limited maximum file size – how large?

Activity 7.1: Linked index blocks

- One index block limits the file size to 512 x 512 words.
- One simple extension to support much larger files is to allow a linked list of *index blocks*, similar to a linked list of *data blocks* in linked allocation (Slide 11.21).
- Based on the above allocation approach …
  - Show the calculation for you to determine the disk block that contains the word at logical address LA.
  - Assume that the only information initially cached in main memory is the file's directory, which contains the index of the file's first index block. How many different disk blocks will you have to access in total in order to read the word at LA?
  - Do we have external fragmentation of disk storage? Explain.
  - Do we have internal fragmentation of disk storage? Explain.

Two-level indexed allocation

- We can generalize indexed allocation to two levels of indices, i.e., each file has one (outer index) block of (inner) *index blocks*
  - Previously, each file has one index block of *data blocks*
- Maximum file size supported = 512^3 words

$$LA / (512 \times 512) \begin{cases} Q_1 \\ R_1 \end{cases}$$

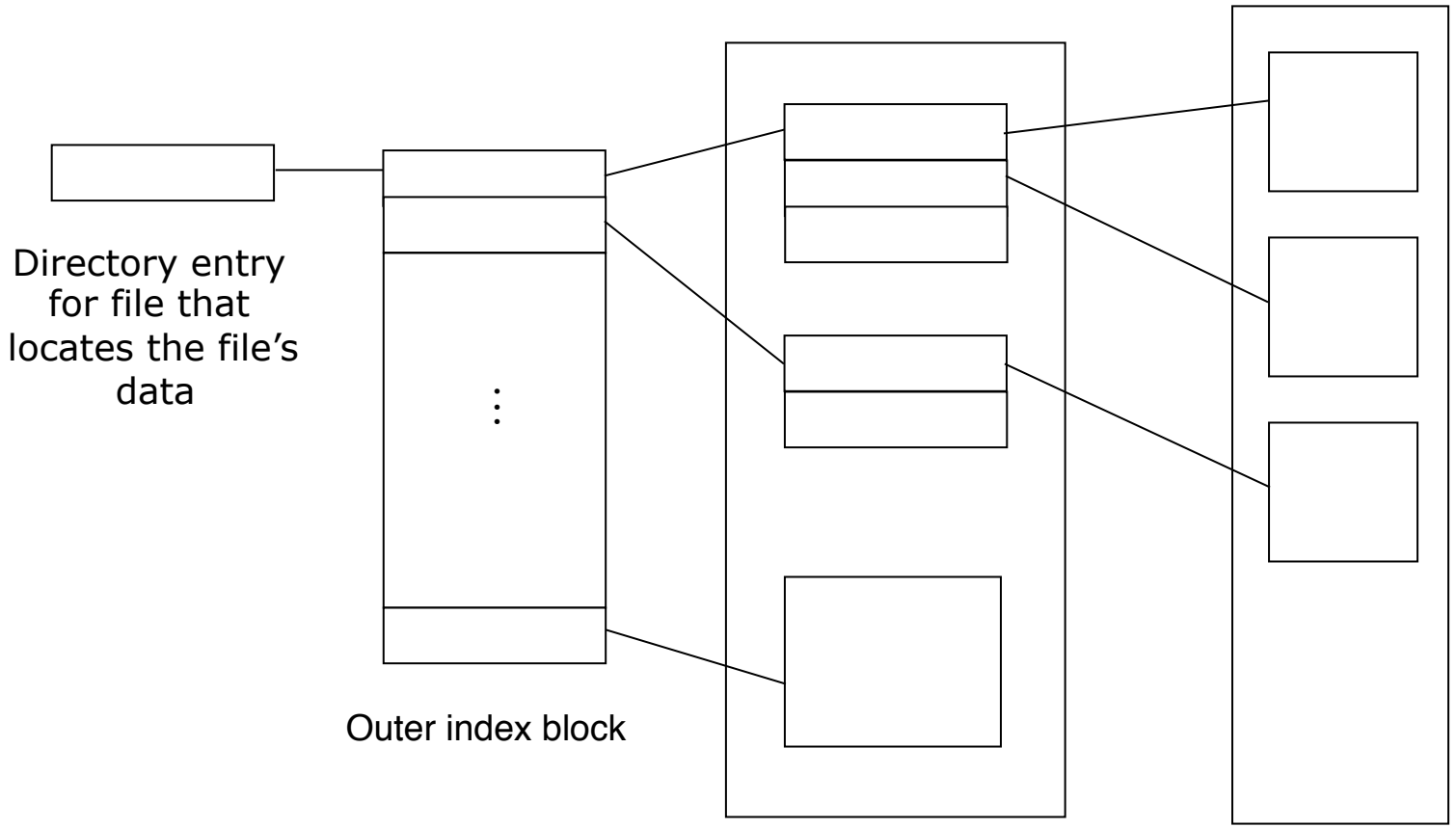
Index of inner index block that locates word LA = Q_1-*th* entry in outer index block

$$R_1 / 512 \begin{cases} Q_2 \\ R_2 \end{cases}$$

Index of data block that contains word LA = Q_2-*th* entry in the inner index block found above

Offset of word LA in the data block = R_2

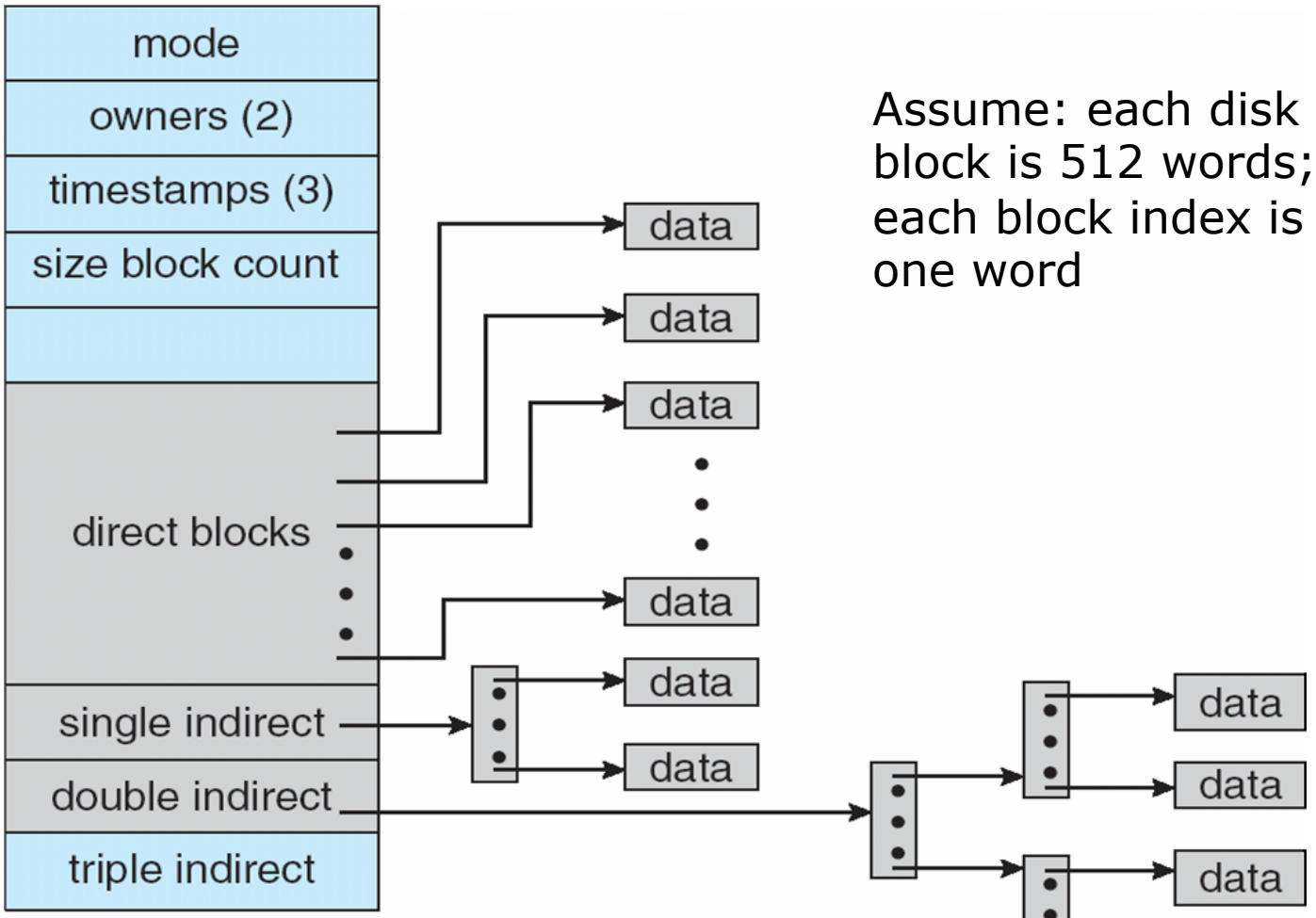
Illustration of two-level indexed allocation



Activity 7.2: Multiple-level indexing and UFS

- Two-level indexing can be naturally extended to multiple levels.
- If you have *n* levels of indexing
  - What is the maximum file size that can be supported as a function of *n*?
  - Assume that in the beginning, only the file's directory information is cached, which contains the index of the file's outermost index block. How many different disk blocks will you have to access in total in order to access a required word in the file?
- The Unix file system (UFS) uses a hybrid indexing scheme shown on the next slide. It has *m* direct blocks (each direct block is a disk block that holds the file's data) for the beginning part of a file. After that (if required), it has, in sequence, one single-indirect block that points to 512 direct blocks, one double-indirect block that points to 512 single-indirect blocks, and a triple-indirect block that points to 512 double-indirect blocks.
  - Discuss the rationale behind this hybrid design.

Hybrid indexed allocation in UFS



Assume: each disk block is 512 words; each block index is one word