Operating Systems: File System Interface, Implementation, and System Internals – Part II

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapter 13, 14 and 15)

Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated to files.
- Three allocation methods are in practice:
  - **Contiguous allocation**
  - **Linked Allocation**
  - **Indexed Allocation**

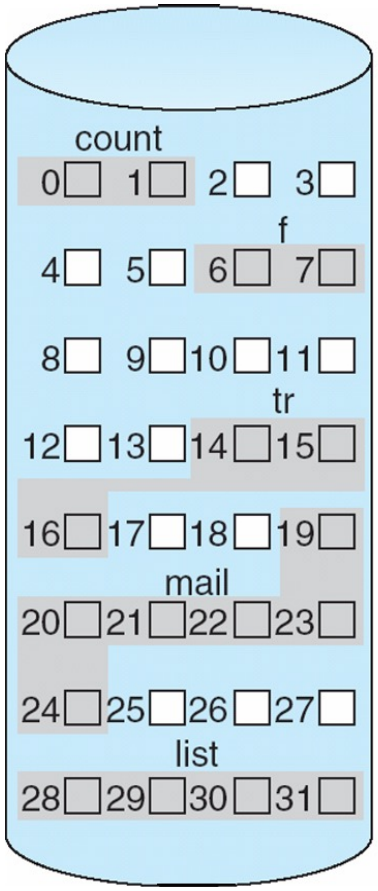
Contiguous Allocation

Each file occupies set of contiguous blocks

- Directory entry contains only starting location (block #) and length (number of blocks).
- Supports random access.

Problems:

- Knowing how the file size would grow.
- Finding contiguous space for file,
- External Fragmentation

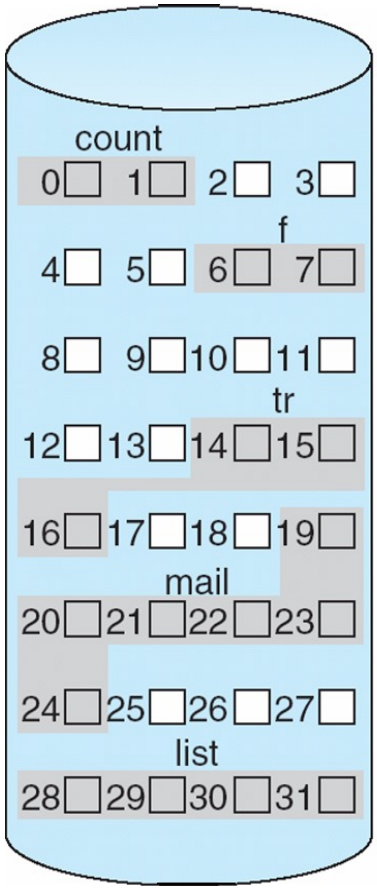


directory

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

Contiguous Allocation Question -1

Problem: How many disk I/O operations are required to add a block at the end of the file `list`? You may assume FCB, and other required file system data structures are in memory.



directory

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

Contiguous Allocation

Problem: How many disk I/O operations are required to add a block at the end of the file `list`? You may assume FCB and other required file system data structures are in memory.

Answer:

- 4 block read operations to read the file.
- 4 block write operations to write the file
- 1 block write operation to add to the end of the file.
- Total of 9 disk I/O operations.

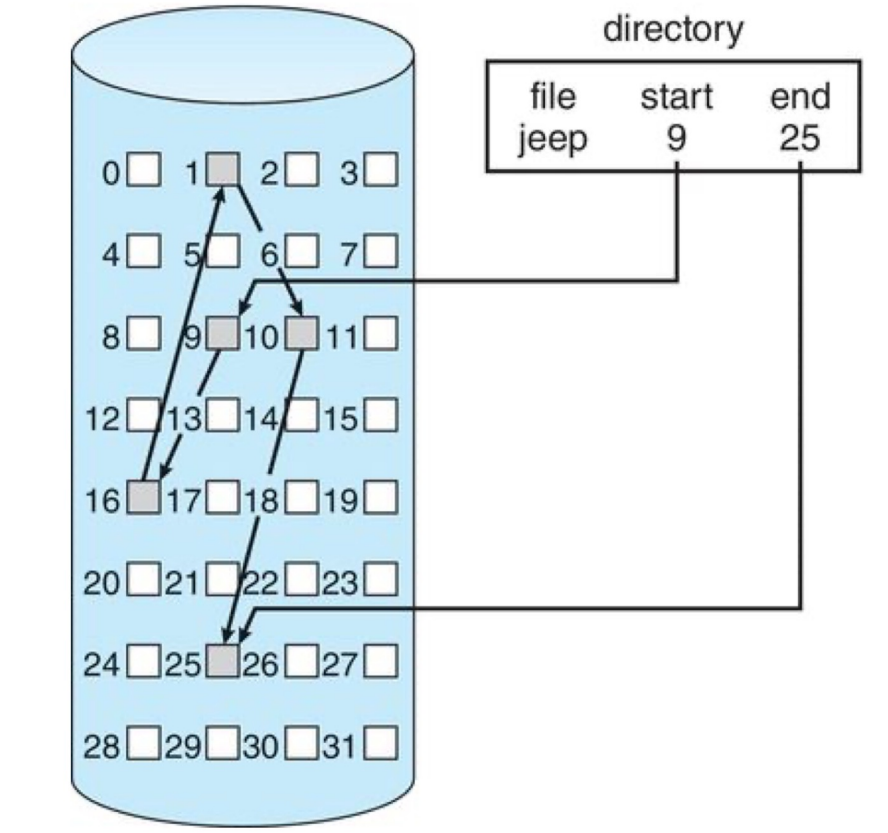
Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme know as:
- **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

Allocation Methods - Linked

Each file is a linked list of disk blocks, and blocks may be scattered anywhere on the disk

- Each block contains pointer to next block
- Directory contains the starting location (block#) and the ending location (block#)

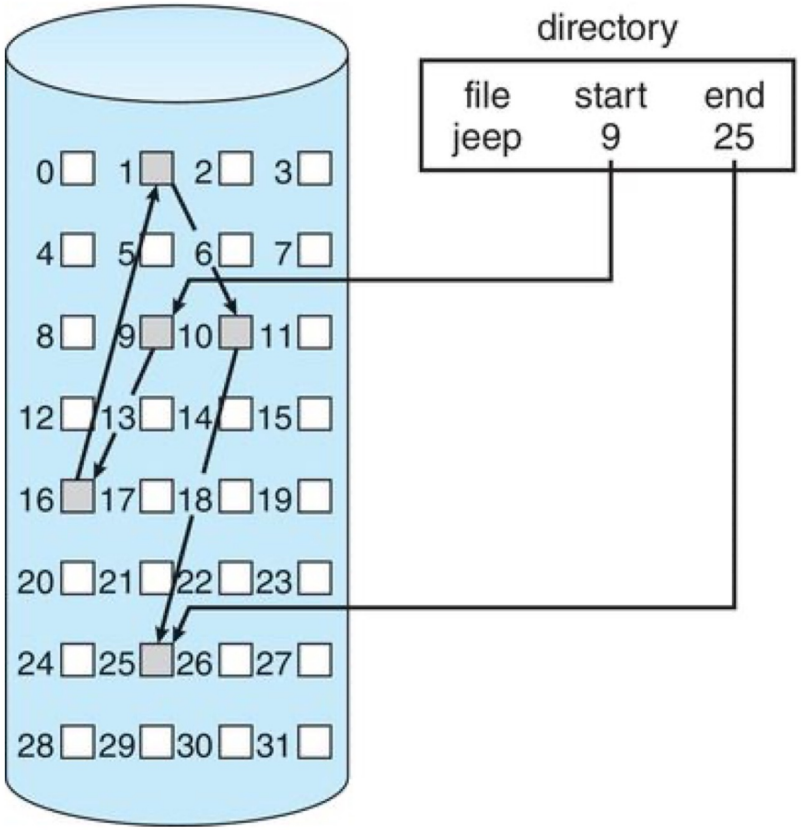


Allocation Methods – Linked Cont…

- No external fragmentation
- However, space is wasted in storing pointers
  - Alleviate this problem by clustering blocks into groups
  - However, this increases internal fragmentation
- Reliability can be a problem (if pointer is lost or damaged)
  - Use doubly linked lists! – however this increases overhead
- Locating a block can take many I/Os
- Accessing a block is slow as you need to traverse all the previous blocks.

Linked Allocation Question -1

Problem: How many disk I/O operations are required to add a block at the end of the file `jeep`? You may assume FCB and other required data structures are in memory.



Linked Allocation Question -1

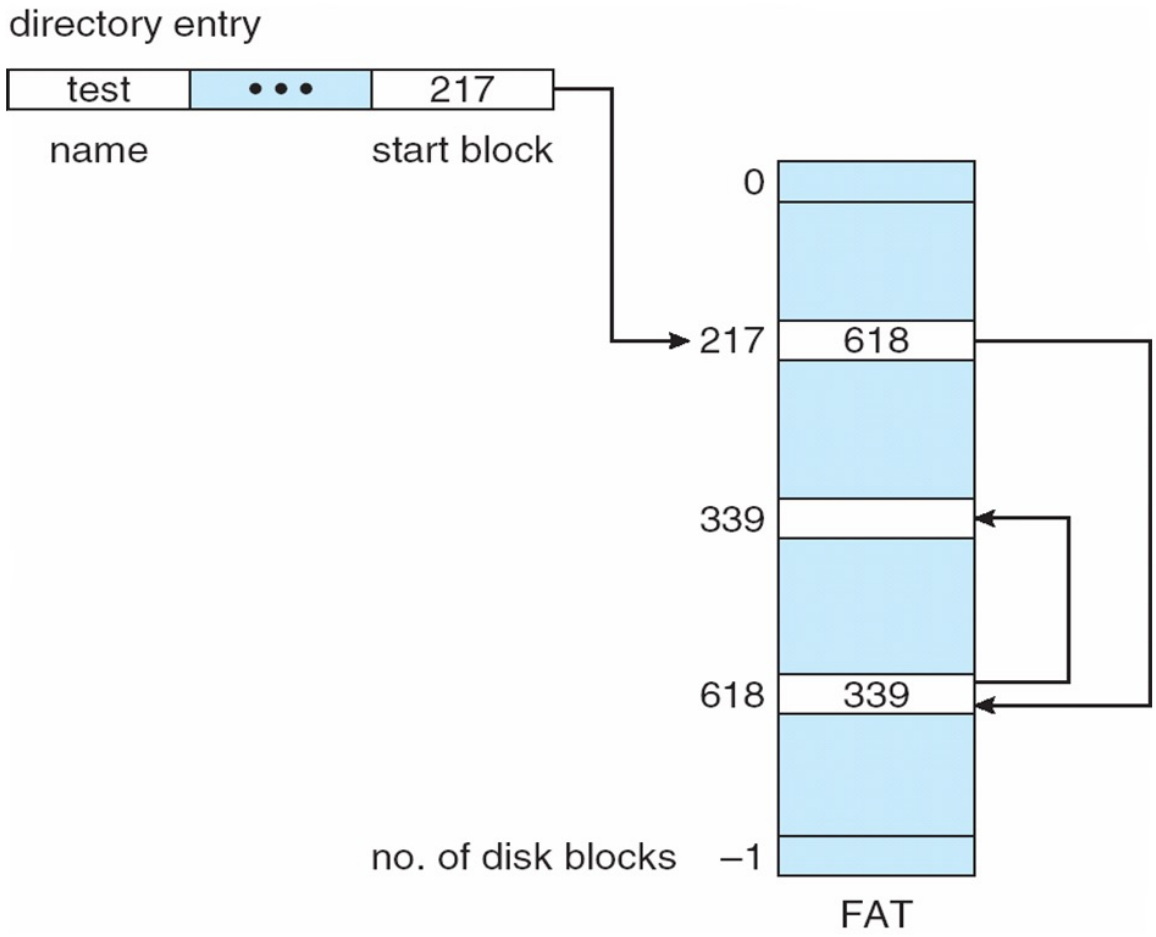
Problem: How many disk I/O operations are required to add a block at the end of the file `jeep`? You may assume FCB and other required data structures are in memory.

- 1 block read operation to read the file's last block (last block# given in the directory structure).
- 1 block write to update the pointer in the (previous) last block (block #25) to point to the newly added block at the end.
- 1 block write operation to write the new block.
- Total of 3 disk I/O operations.

Variation of Linked allocation – FAT

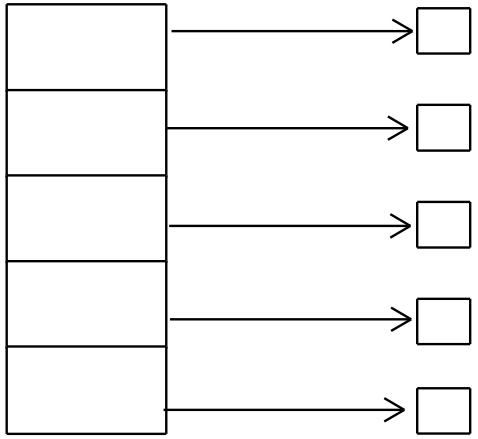
- **FAT (File Allocation Table)** variation of Linked Allocation
  - Beginning of volume has this table, indexed by block numbers of the file system.
  - Directory entry contains **the block# of the first block of the file**
  - The table entry indexed by that block# contains the block# of the next block in the file
  - Chain continues until it reaches the last block
    - Its table entry has a special end-of-file value
  - An unused block is indicated by a table value of 0
- **FAT table must be cached.**

File-Allocation Table



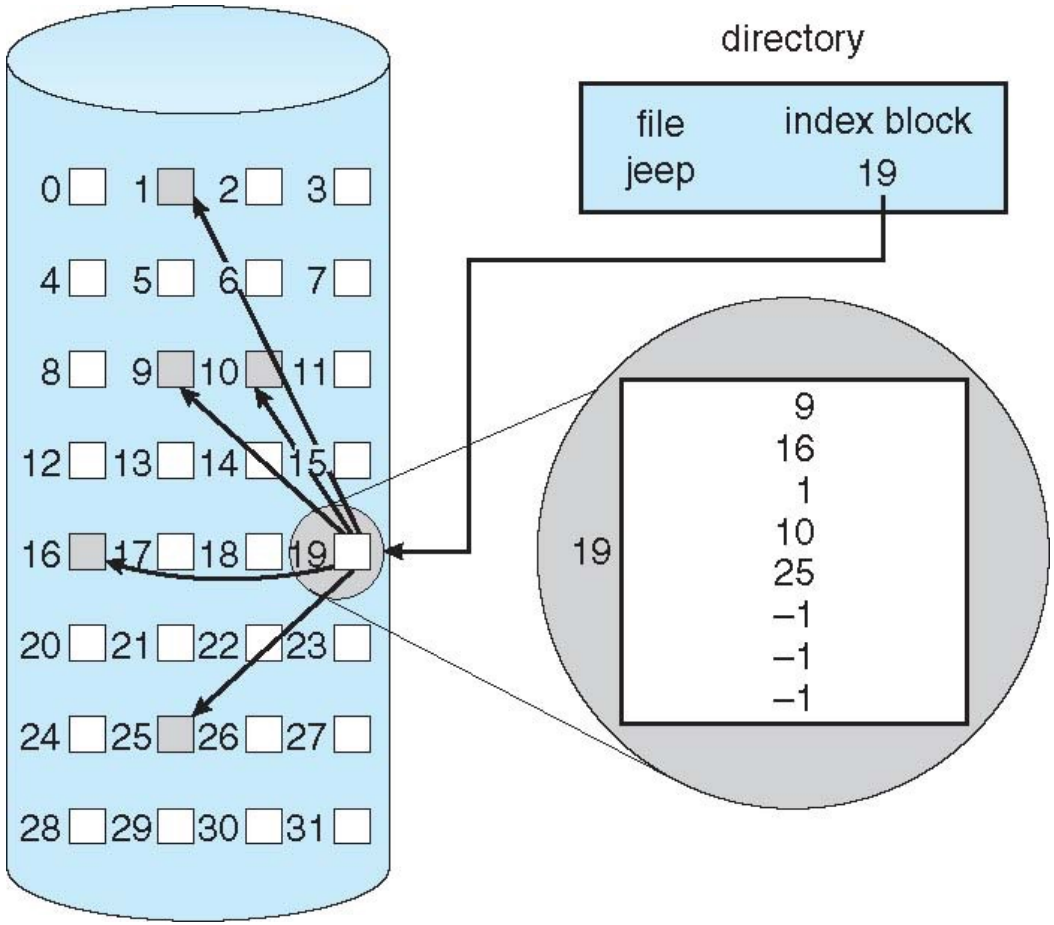
Allocation Methods – Indexed Allocation

- **Each file has its own index table consisting of of pointers to its data blocks**
- i^{th} entry in the index table points to the i^{th} block of the file.
- Directory contains the address of the index block
- Similar to page table and paging!



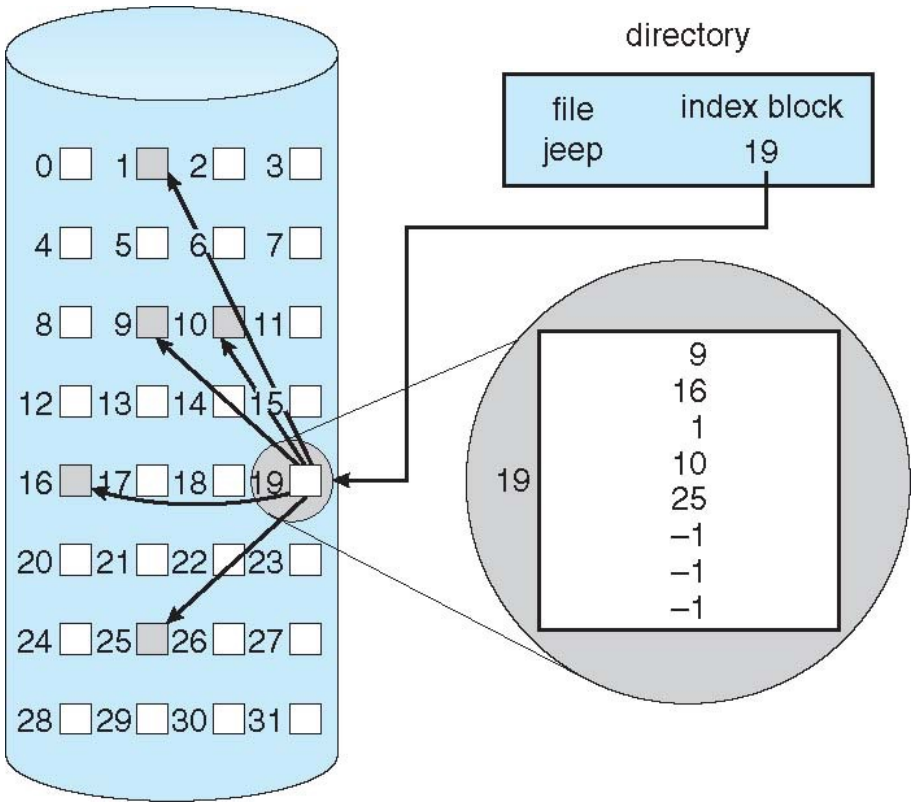
index table

Example of Indexed Allocation



Linked Allocation Question -1

Problem: How many disk I/O operations are required to add a block at the end of the file `jeep`? You may assume FCB and other required data structures such as index block are are in memory.



Indexed Allocation (Cont.)

- Supports random access, but overhead of index block
- No external fragmentation
- Number of entries in the index block depends on the size of the block and size of the pointer holding block addresses.
  - If block size = 512 bytes and pointer size = 4 bytes:
    - Total number of entries in the index block = 512/4 = 128
  - If the index table requires just 1 block, what is the max. file size in this case?

Indexed Allocation Cont…

- What happens if file is large such that the index block runs out of space to hold data block addresses?
- Various scheme to deal with this:
  - **Linked Scheme**
  - **Multilevel Scheme - Multilevel index**
  - **Combined Scheme**

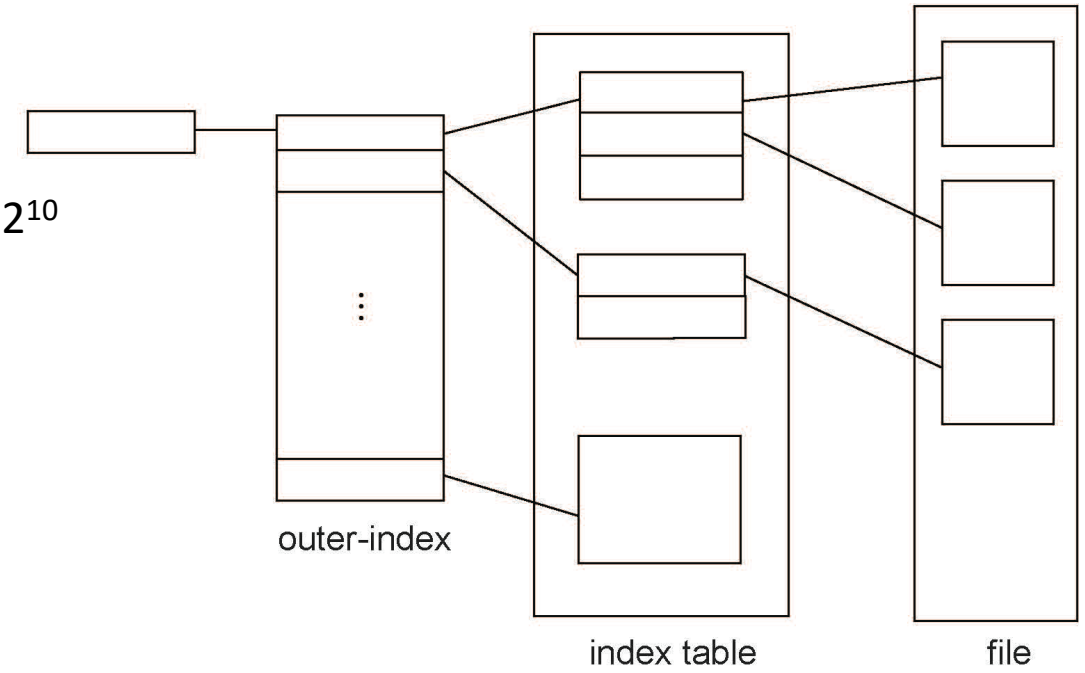
Multilevel Scheme - Multilevel index

- **Two level Scheme** – First-level index block points to a set of second-level index blocks, which in turn point to the file's data blocks.
  - This approach could be continued to a third, fourth or n-th level, depending on the desired maximum file size.
- Similar to Hierarchical paging!

Indexed Allocation – Multi-Level Scheme

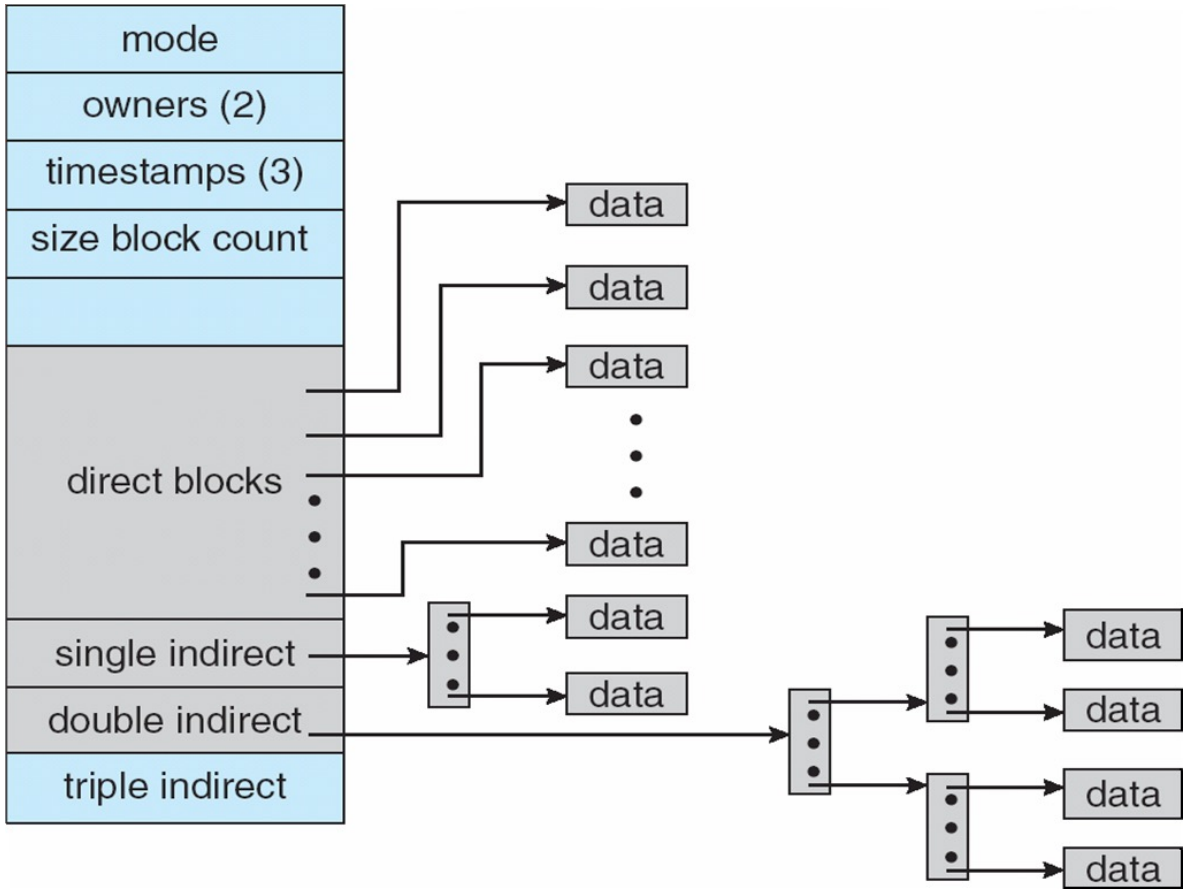
Example: Consider a two-level index scheme. Given block size = 4KB = 2^{12} bytes and pointer size = 4 bytes. What is the maximum file size in the system?

- No. of pointers in one block = $2^{12}/4 = 2^{12}/2^2 = 2^{10}$
- First level index block points to 2^{10} second level index blocks.
- Each second level index blocks point to 2^{10} data blocks
- The maximum file size in the system = $2^{10} * 2^{10} * 2^{12} = 2^{32} = 4GB$



Combined Scheme: UNIX UFS

Uses direct blocks (addresses of the data blocks) and indirect blocks (addresses of the index blocks).



Performance

- **Contiguous allocation** is **great** for sequential and random access. Works well for small files. However, suffers from external fragmentation.

- **Linked allocation** is **good** for sequential access for large files, but inefficient for random access

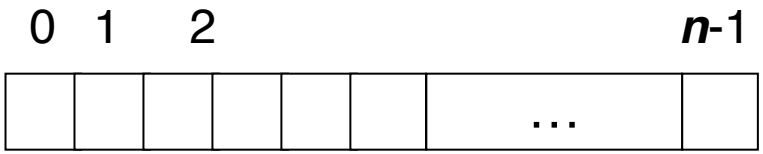
- **Indexed allocation** is more complex, but is good for both sequential and random access, for large files.

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
- Many implementations of free-space lists exists
  - **Bit vector** – maintains a bit vector of size n to store information of n blocks.
  - **Linked List** – maintains a linked list of free blocks.
    - Space is frequently contiguously used and freed,
    - With contiguous-allocation linked list of extents, or clusters maintained.
      - Each entry in the free-space list then consists of a disk address and a count.

Free space list Implementation - Bit vector

- **Bit vector** or **bit map** (***n*** blocks)



bit[***i***] = 1 ⇒ block[***i***] free
0 ⇒ block[***i***] occupied

- Simple and easy to find first free block or ***n*** consecutive blocks
- Need to be kept in memory for efficiency.
- Also, to be written to disk occasionally – **Why?**

Bit vector example

- Inefficient as it requires lot of space.
  - Example:

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

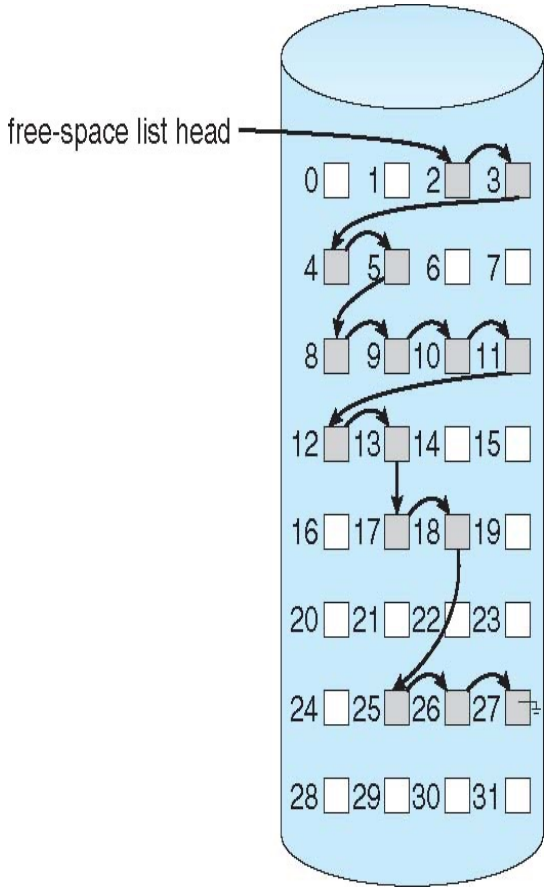
    disk size = 2^{40} bytes (1 terabyte)

    no of blocks = ***n*** = $2^{40}/2^{12} = 2^{28}$

    32 MB needed to store the bit map!

Free space list Implementation - Linked List

- Linked list - keep track of all free blocks.
- Generally, the system just adds and removes single blocks from the beginning of the list.
- Traversing the list and/or finding a contiguous block of a given size are not easy sometimes.



Free space list Implementation - Linked List

- **Grouping**
  - 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.