File Allocation

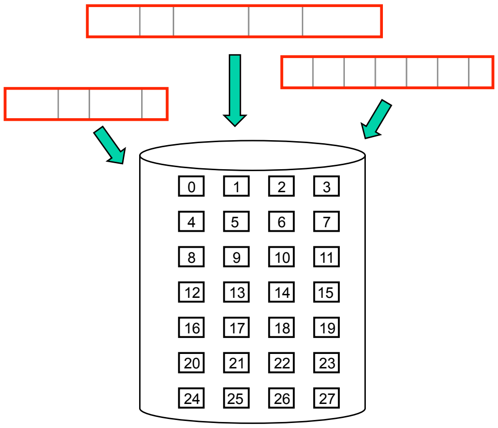
- The File Allocation Problem; Requirements
- Contiguous Allocation
- Linked Allocation and its Variations
  - Example: File Allocation Table (FAT)
- Indexed-Allocation
  - Example: UNIX

The File Allocation Problem

Observation: Files are sequences of (potentially variable-sized) records.

Observation: The storage device offers a large array of fixed-sized blocks.

Question: How do we store our file data in these blocks?

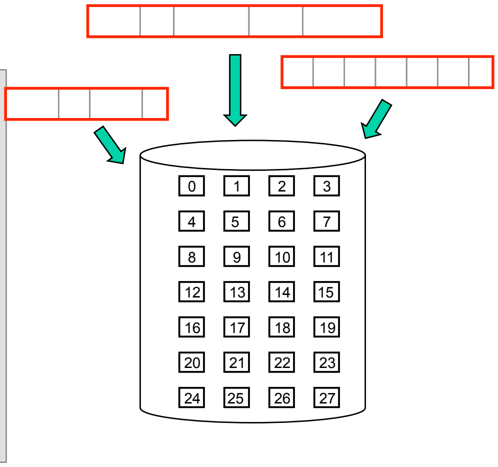


The File Allocation Problem

Question: How do we store our file data in these blocks?

Requirements:

- Space on storage device must be utilized effectively
- File operations must be supported efficiently
  - Direct access to data in file (random access)
  - Traversal of file (sequential access)
  - append/insert/delete file data
  - create/delete files

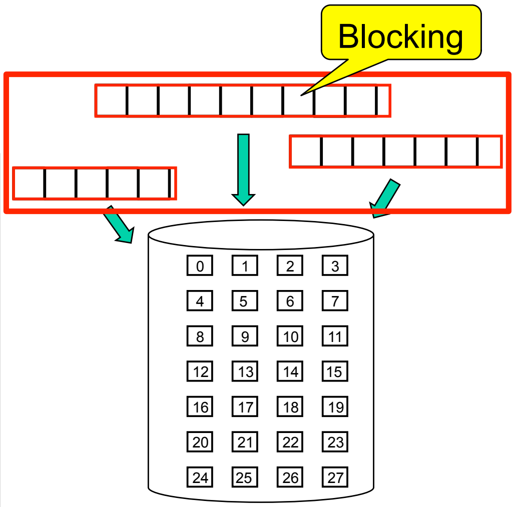


Blocking

Observation: Files are sequences of (potentially variable-sized) records.

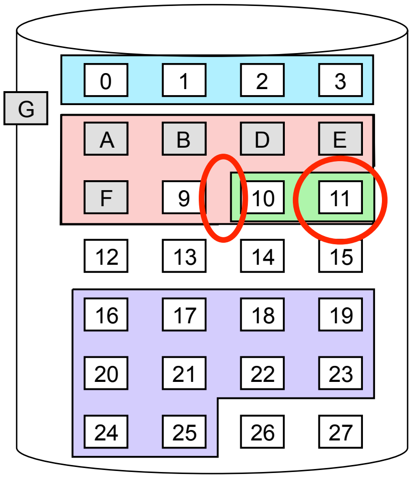
Requirements:

- Space on storage device must be utilized effectively
- File operations must be supported efficiently
  - Direct access to data in file (random access)
  - Traversal of file (sequential access)
  - append/insert/delete file data
  - create/delete files



Contiguous Allocation

file	start	length
file1	4	7/8
file2	26	2
file3	16	10



File mapped onto a sequence of adjacent physical blocks.

Pros:

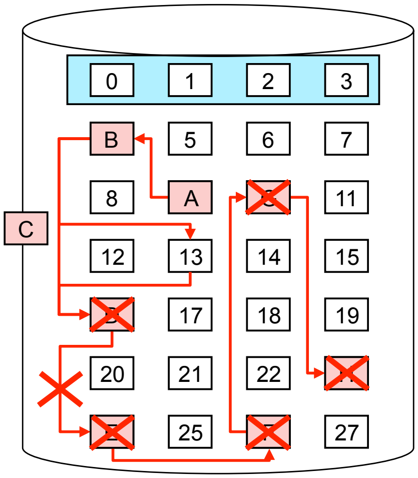
- minimizes head movements
- simplicity of both sequential and direct access.

Cons:

- Inserting/Deleting records, or changing length of records difficult.
- Size of file must be known *a priori*. (Solution: copy file to larger hole if exceeds allocated size.)
- External fragmentation
- Pre-allocation causes internal fragmentation

Linked Allocation

file	start	end
file1	9	23
...	...	...
...	...	...



- Scatter logical blocks throughout secondary storage.
- Link each block to next one by forward pointer.

Pros:

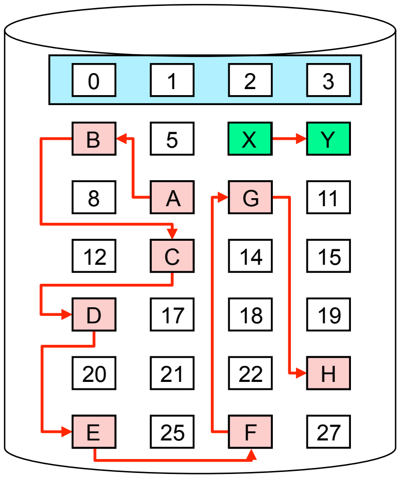
- blocks can be easily inserted or deleted
- no upper limit on file size necessary *a priori*

Cons:

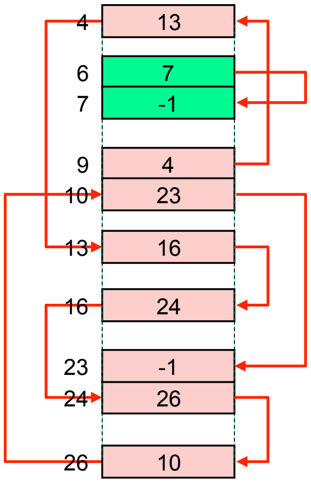
- random access difficult and expensive
- sequential access expensive
- overhead required for pointers in blocks
- reliability

Variations Linked Allocation

file	start	end
file1	9	23
…	…	…
…	…	…

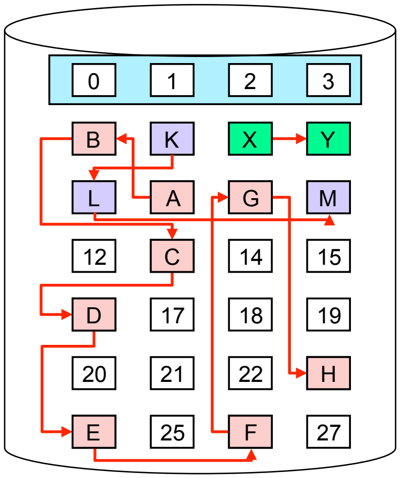


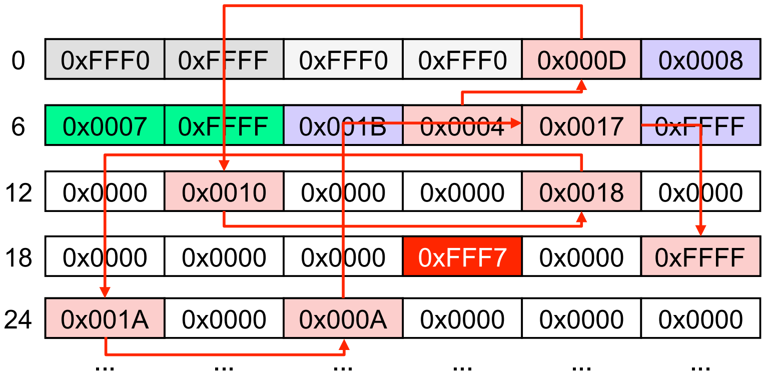
- Maintain all pointers as separate list.
- Preferably load the list into main memory.



Example: FAT (File Allocation Table)

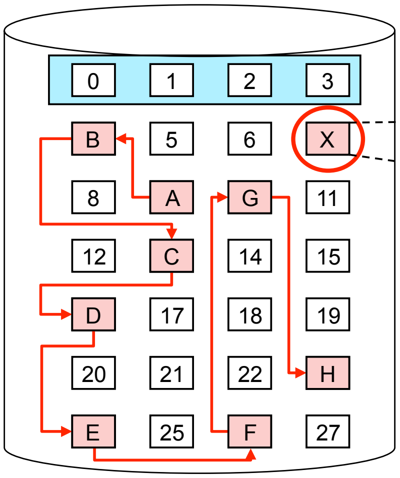
- Example: FAT-16
- File allocation table is array of 16-bit entries
- One entry per “cluster”





0	0xFFF0	0xFFFF	0xFFF0	0xFFF0	0x000D	0x0008
6	0x0007	0xFFFF	0x001B	0x0004	0x0017	0xFFFF
12	0x0000	0x0010	0x0000	0x0000	0x0018	0x0000
18	0x0000	0x0000	0x0000	0xFFF7	0x0000	0xFFFF
24	0x001A	0x0000	0x000A	0x0000	0x0000	0x0000
	…	…	…	…	…	…

Indexed Allocation



file	index block
file1	7
…	…
…	…

Keep all pointers to blocks in one location:
index block (one index block per file)

9	4	13	16	24	26	10	23	-1	-1	-1

Pros:

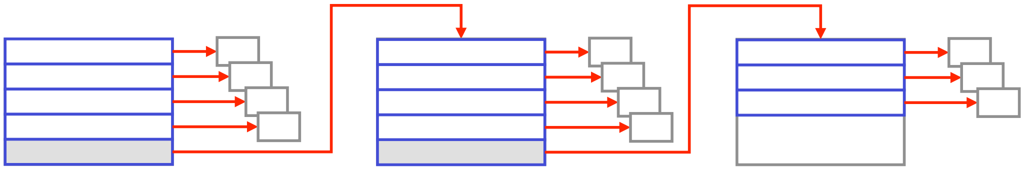
- supports random access
- no external fragmentation

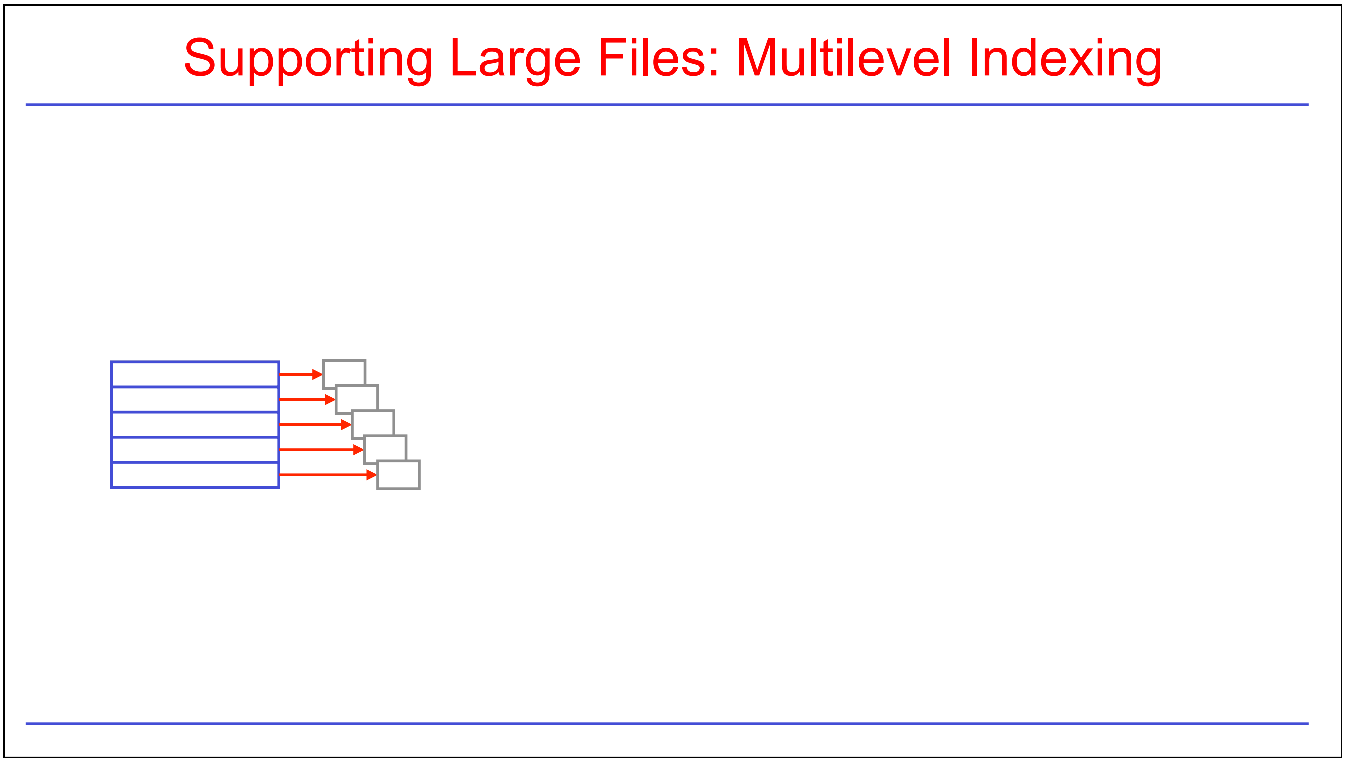
Cons:

- internal fragmentation in index blocks

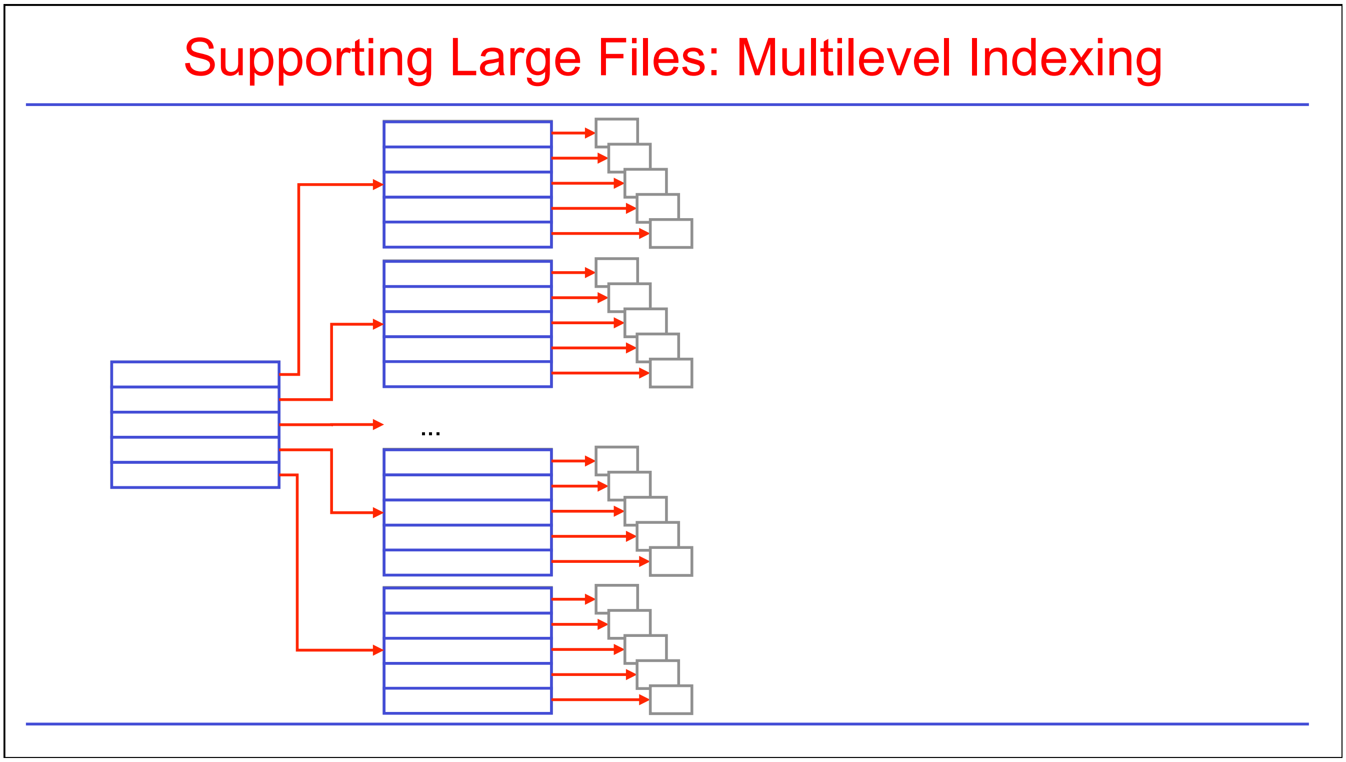
Question: What is a good size for index block?
Trade-off: fragmentation *vs.* file length

Supporting Large Files: Linked Index Blocks



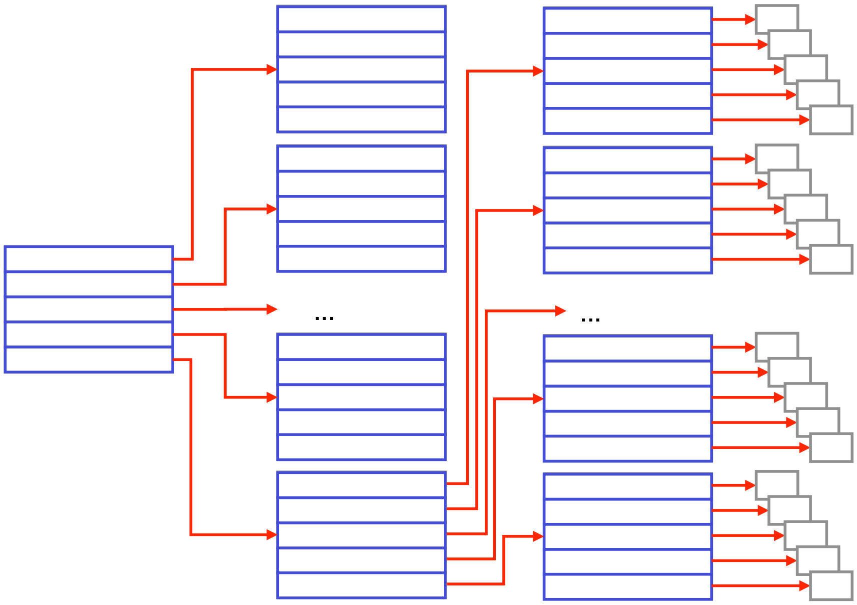


Supporting Large Files: Multilevel Indexing

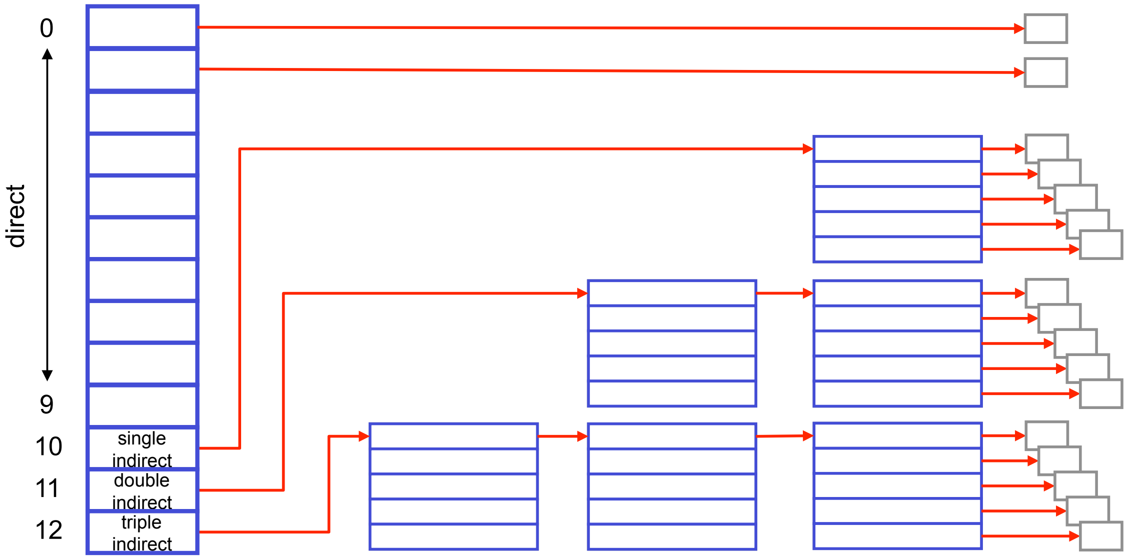


Supporting Large Files: Multilevel Indexing

Supporting Large Files: Multilevel Indexing

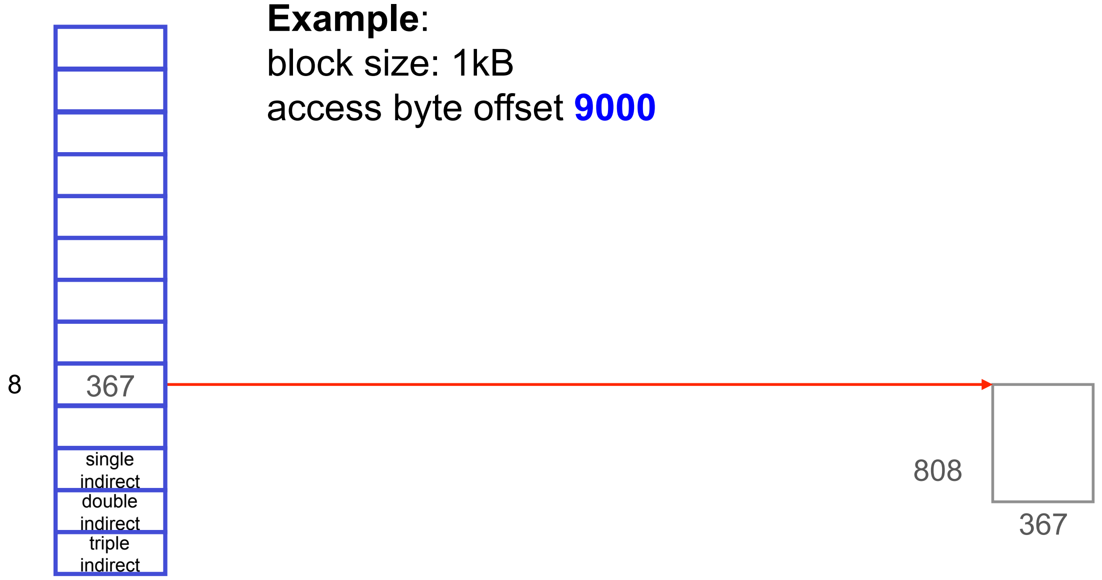


Index Block Scheme in UNIX



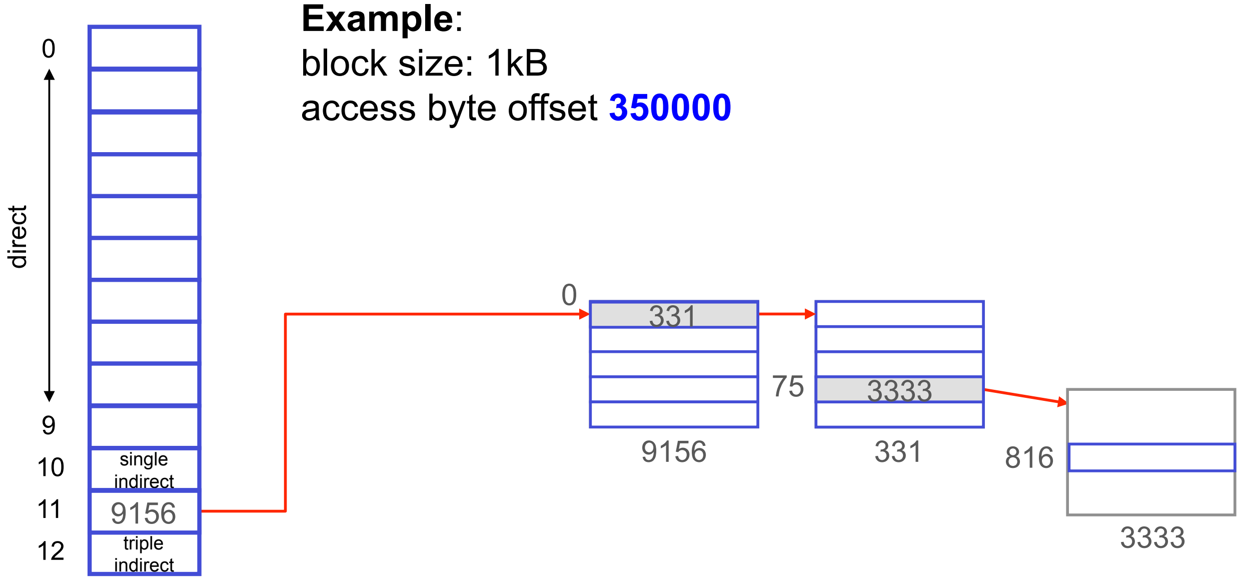
Index Block Scheme in UNIX

Example:
block size: 1kB
access byte offset **9000**



Index Block Scheme in UNIX

Example:
block size: 1kB
access byte offset **350000**



File Allocation

- The File Allocation Problem; Requirements
- Contiguous Allocation
- Linked Allocation and its Variations
  - Example: File Allocation Table (FAT)
- Indexed-Allocation
  - Example: UNIX