Week 11

Persistent Storage: Basic File System Implementation

(Part 1)

Oana Balmau

March 14, 2023

Balmau © 2023

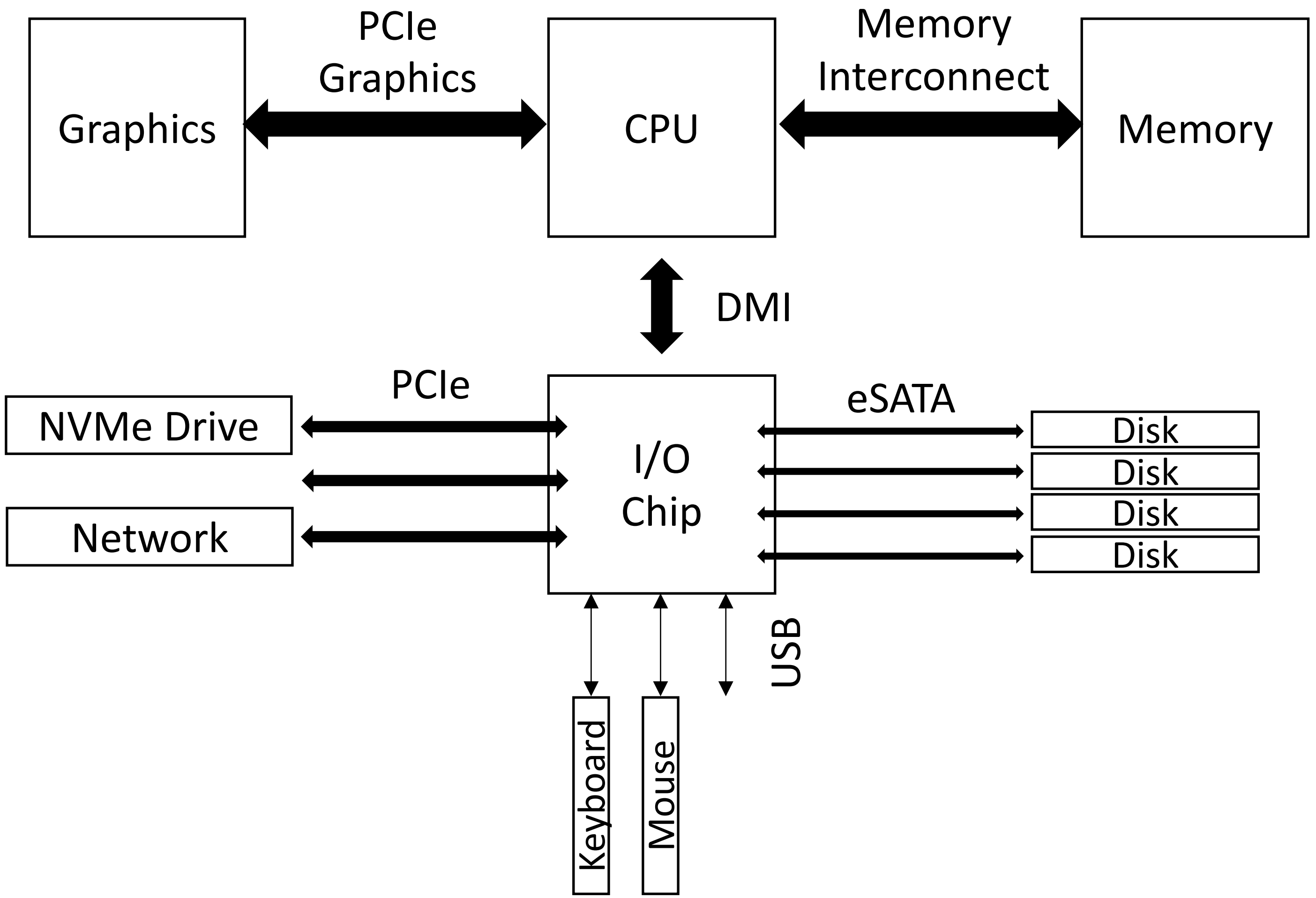
Class Admin

Week 11 **File Systems**	mar 13 **Scheduling Assignment Due** **~~Graded Exercises Due~~** C Review: Complex structs	mar 14 **Basic File System Implementation (1/2)** Optional reading: OSTEP Chapters 40, 41, 45	mar 15	mar 16 **Basic File System Implementation (2/2)** • ~~Grades released for Scheduling Assignment~~	mar 17
Week 12 **File Systems**	mar 20 **Graded Exercises Due** C Review: Pointers & Memory Allocation II	mar 21 **Advanced File System Implementation (1/2)**	mar 22	mar 23 **Advanced File System Implementation (2/2)** • Grades released for Scheduling Assignment	mar 24
Week 13 **File Systems**	mar 27 C Review: Advanced debugging	mar 28 **Handling Crashes & Performance (1/2)** Optional reading: OSTEP Chapters 38, 43	mar 29	mar 30 **Handling Crashes & Performance (2/2)** • ~~Grades released for Exercises Sheet~~ • Practice Exercises Sheet: File Systems	mar 31
Week 14 **Advanced Topics**	apr 3 No lab. Work on Assignment 3 **~~Memory Management Assignment Due~~**	apr 4 **Advanced topics: Virtualization**	apr 5	apr 6 **Advanced topics: Operating Systems Research (Invited Speaker: TBD)** Grades released for Exercises Sheet	apr 7
Week 15 **Wrap-up**	apr 10 No Lab. Prepare for end-of-semester. **Memory Management Assignment Due**	apr 11 **End-of-semester Q&A– not recorded**	apr 12	apr 13 **End-of-semester Q&A — not recorded.** Last class!	apr 14 Grades released for Memory Management Assignment

Key Concepts

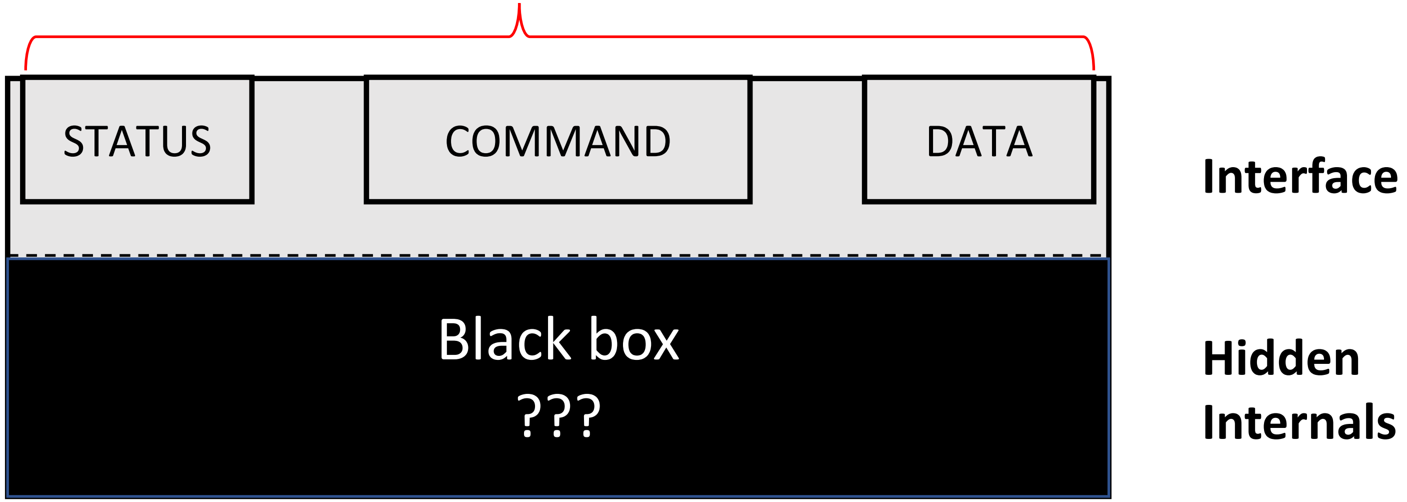
- File system "mental model"
  - Data structures : on disk, in memory
  - File data allocation methods
    - Contiguous, extent-based, linked, FAT, indexed, indirect blocks
  - File access methods
    - Create, open, write, read, close

Recap Week 10 - I/O System Architecture



Recap Week 10 - How does OS use devices?

Canonical device interface: OS reads/writes to these to control device behavior



- OS accesses device interface through: polling, interrupts,
- Direct memory access (DMA)

Recap Week 10 - File System Interface

Recap Week 10 - File

- Un-interpreted un-typed sequence of bytes
- Identified by a globally unique *uid*

Recap Week 10 - Open File

- File instance accessed by a process
- Identified by a per-process unique *tid* or *fd*

Recap Week 10 - Directory

- Set of mappings (string → uid)

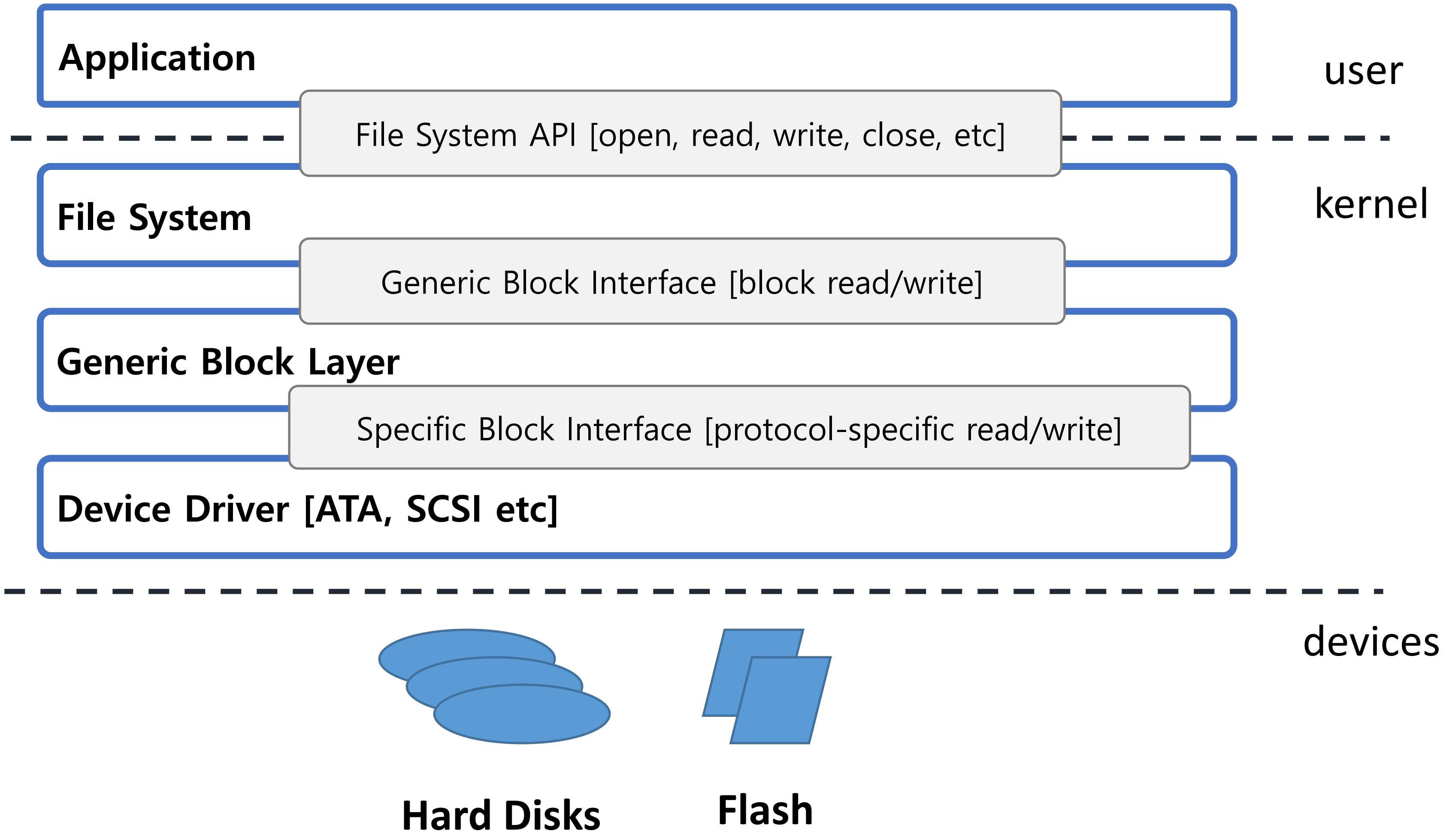
Recap Week 10 - File System Primitives

- Access: Create(), Delete(), Read(), Write()
- Random vs. Sequential and Seek()
- Concurrency: Open(), Close()
- Naming

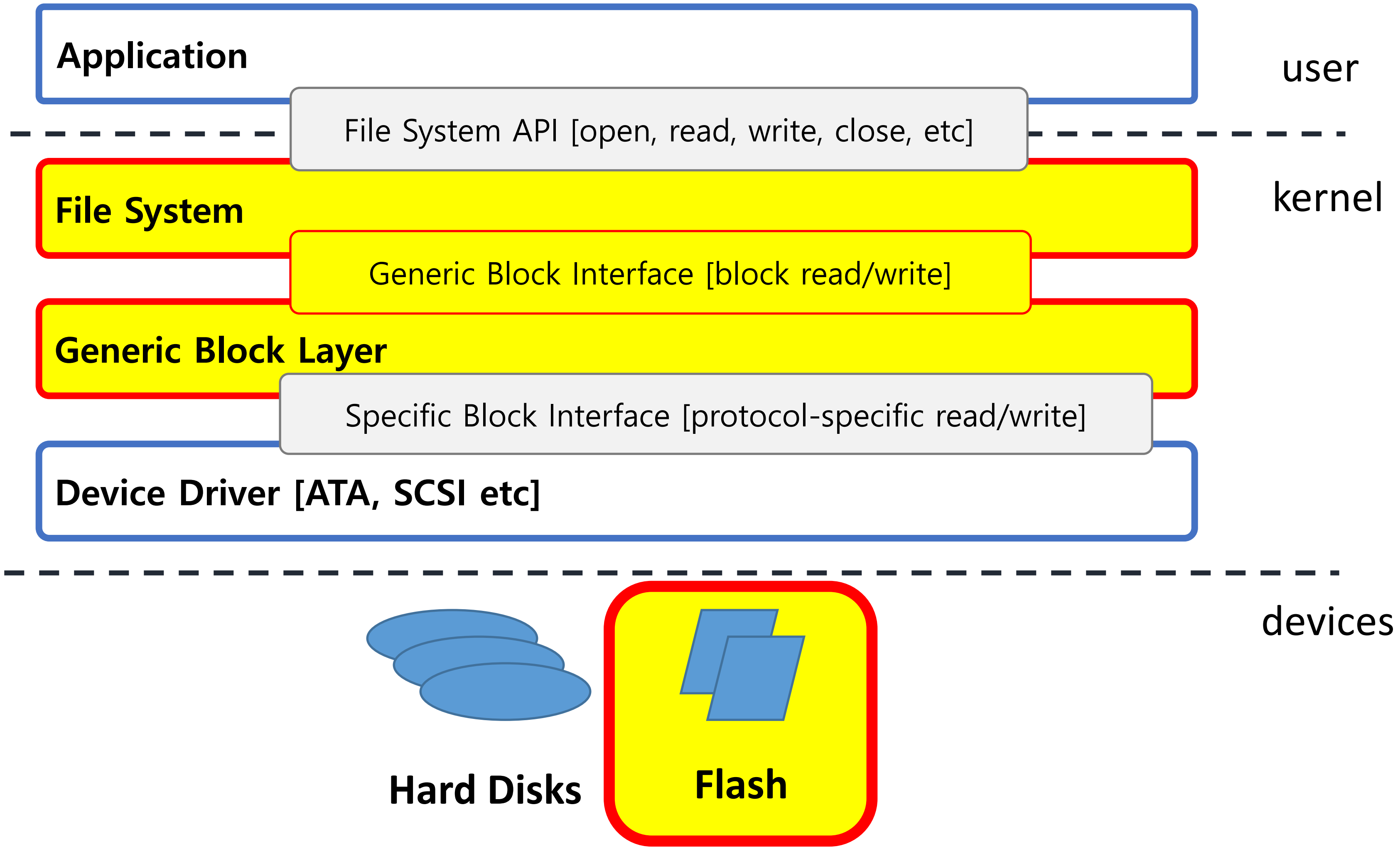
Recap Week 10 - Hierarchical Directory Structures

- Tree
- (Acyclic) Graph
  - Allows sharing of two *uid*s under different names
- Two additional primitives
  - HardLink() and SoftLink()

Overall Picture



This week and next week Lectures



File System Implementation

File System Role

The main task of the file system is to **translate**

From **user interface** functions

`Read(`**`uid`**`, buffer, `**`bytes`**`)`

To **disk interface** functions

`ReadSector(`**`logical_sector_number`**`, buffer)`

File System Implementation

Key aspects of the system:

1. **Data structures**
   - On disk
   - In memory
2. **Access methods**
   - How do we open(), read(), write() ?

Data Structures

Disk vs in-memory simple but golden rules:

- If it is **not on disk** and you crash, it is **gone**!
- **If you need it** after a crash, it **must be on disk.**

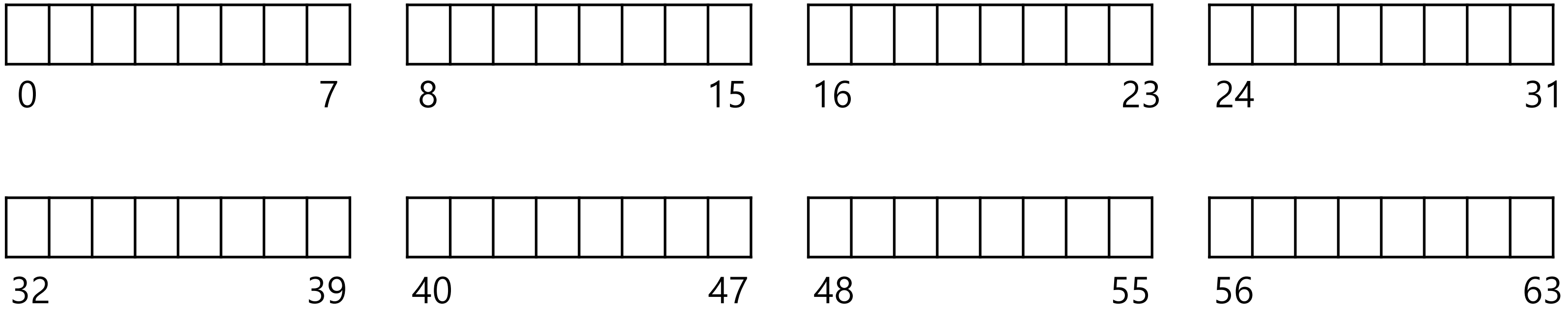
Disk Data Structures

Disk Data Structures

- **Data Region** ← occupies most space in FS
  - User data
  - Free space
- **Metadata**
  - Inodes
  - Free space management
  - Superblock

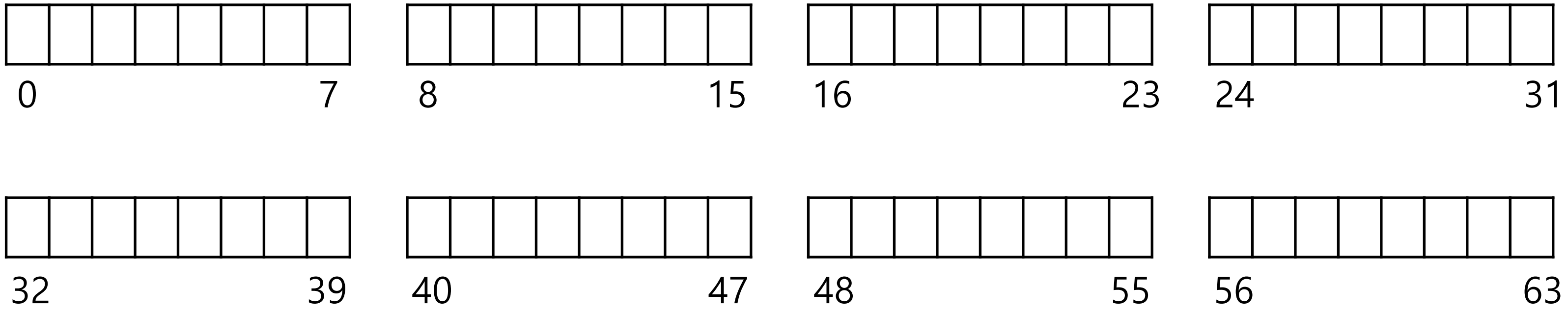
Simple Example

- Disk with 64 blocks.
- 4KB block size.



Simple Example

- Disk with 64 blocks.
- 4KB block size.

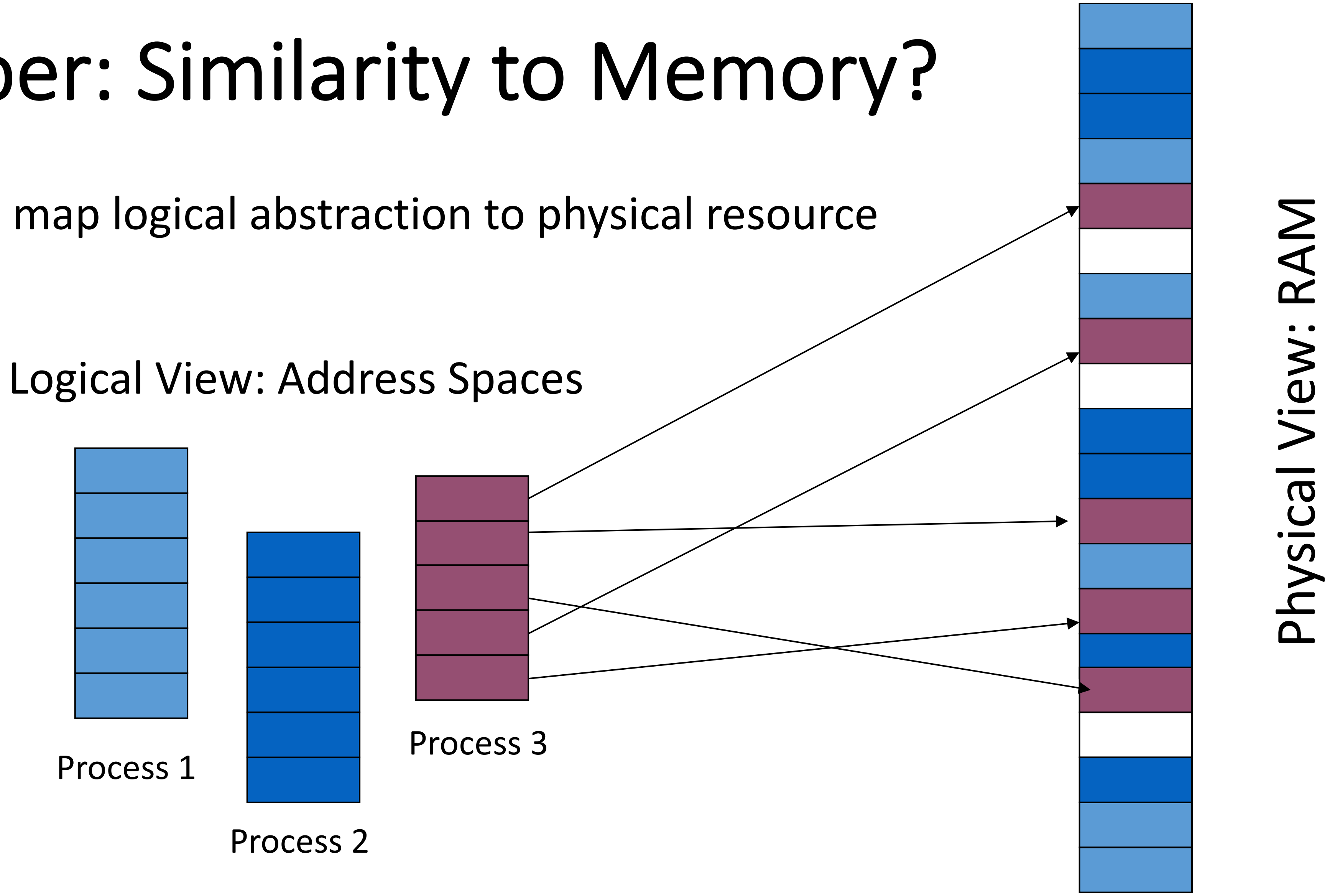


Remember: Want translation between user FS interface to disk interface

→ Need some structure to map files to disk blocks

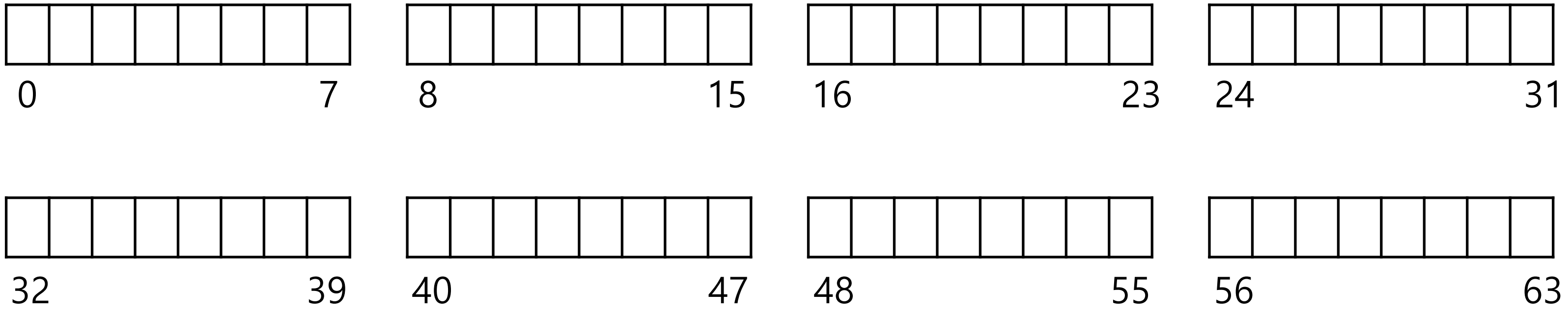
Remember: Similarity to Memory?

Same principle: map logical abstraction to physical resource



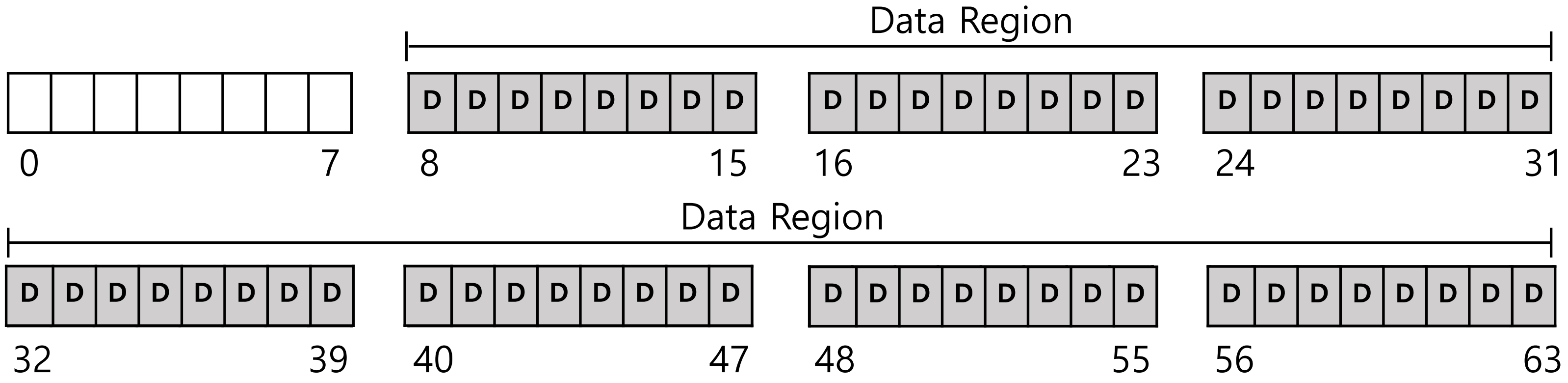
Need structure to map files to disk blocks

- Disk with 64 blocks.
- 4KB block size.



Reserve Data Region to Store User Data

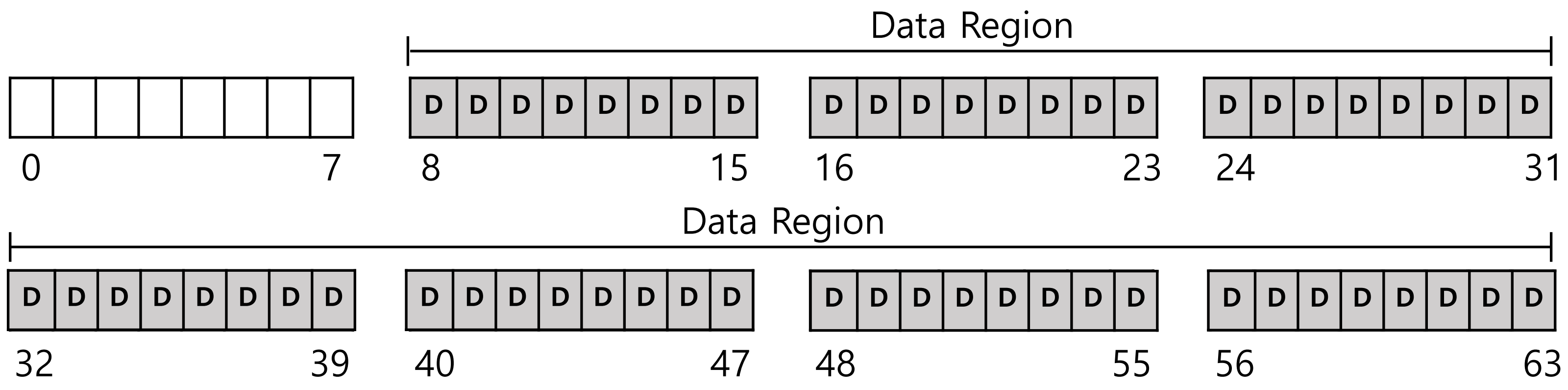
- Disk with 64 blocks.
- 4KB block size.



- Data region contains user data and free space.

Reserve Data Region to Store User Data

- Disk with 64 blocks.
- 4KB block size.



- Data region contains **user data (files)** and **free space**.

→ How do we track files ?

→ How do we track free space ?

How do we track files?

Inode

- Short from “index node”

- Tracks file metadata:
  - type (file or dir?)
  - uid (owner)
  - rwx (permissions)
  - size (in bytes)
  - blocks occupied by file
  - timing information (creation time, last access time)
  - links_count (# paths)

Reserve Inode Table to Track Files

Inode table: holds an array of on-disk inodes
- Inodes are not too large (128 or 256 bytes per inode)

How many inodes?

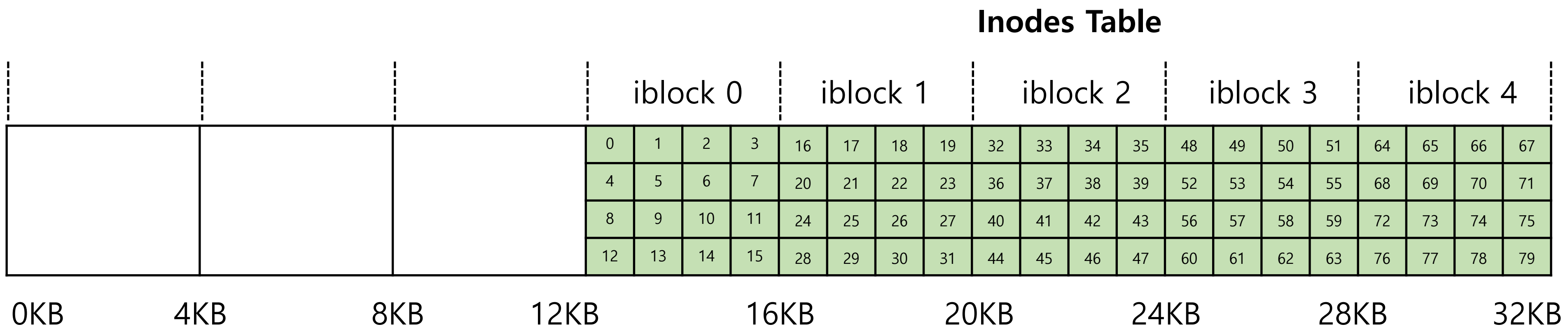
Question: Assuming 256B per inode, how many inodes in our file system with 4KB blocks?

How many inodes?

Question: Assuming 256B per inode, how many inodes in our file system with 4KB blocks?

4KB = 4096 B → 4096 / 256 = 16 inodes per block → 16 * 5 = **80 inodes in total in FS**

Close-up on Inodes Table



How do we track free space?

Allocation structures:

- For data
- For inodes

Allocation Structures Implementation

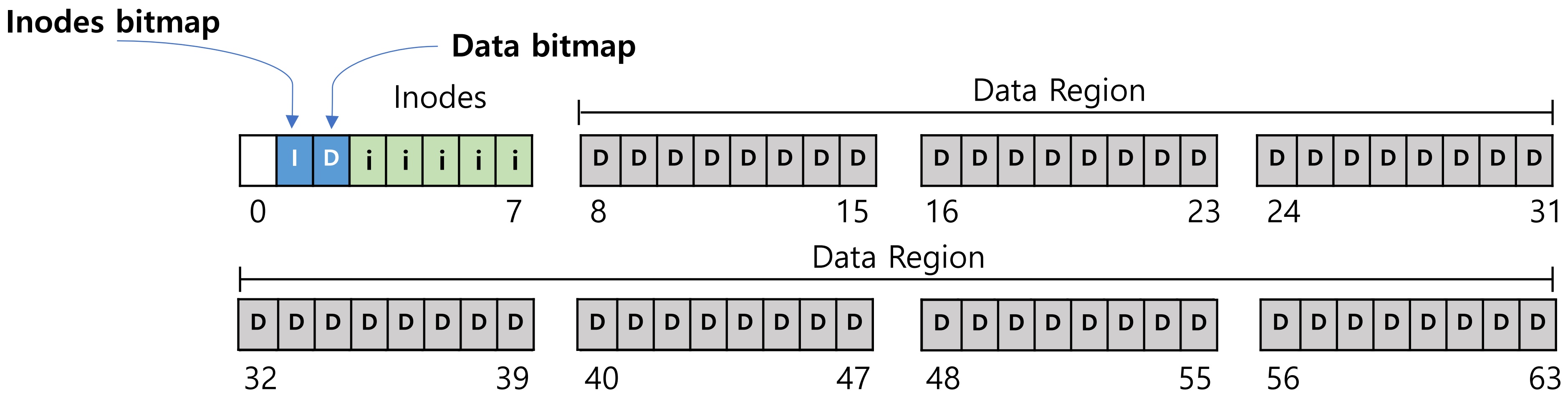
- Free-lists (Remember from memory management module)
- Bitmaps
  - Data structure where each bit indicates if corresponding object is free or in use
  - 1 = in use
  - 0 = free

Allocation Structures Implementation

We will use bitmaps:

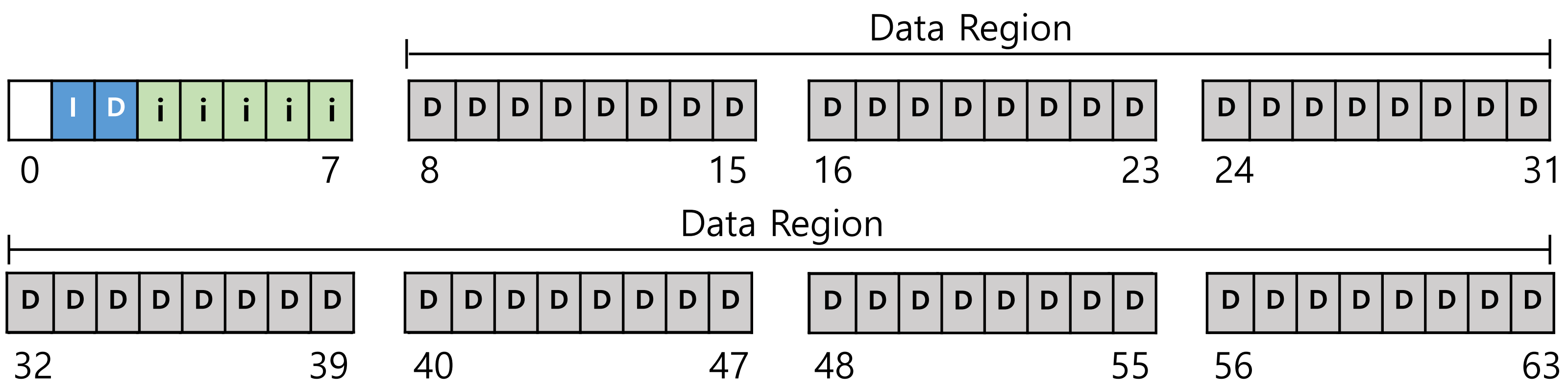
- Data bitmap
- Inodes bitmap

Reserve Blocks for Allocation Structures



Bitmap capacity?

Question: Assuming we use one 4KB block per bitmap, how many inodes and data blocks can we track?

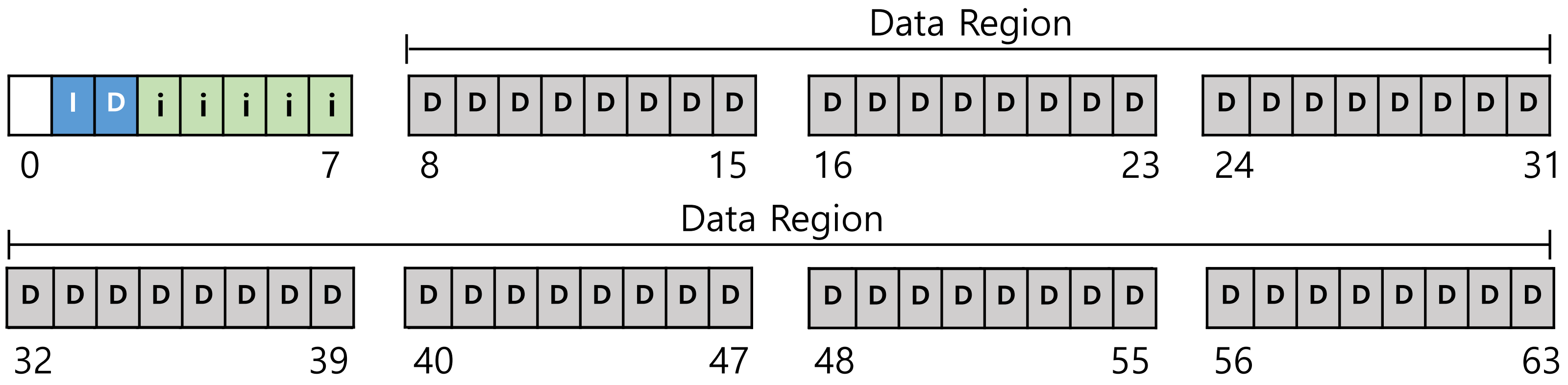


Bitmap capacity?

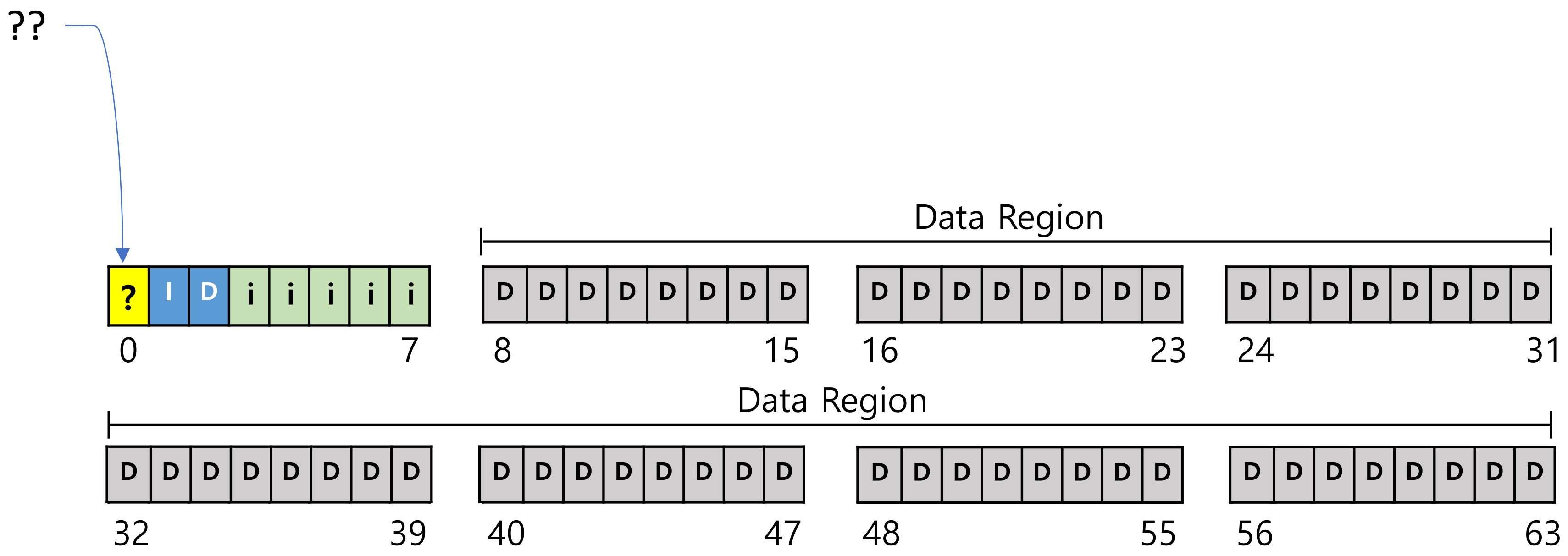
Question: Assuming we use one 4KB block per bitmap, how many inodes and data blocks can we track?

4096 B * 8 bits/B = 32768 bits in each bitmap

→ Can track ~32K inodes and ~32K blocks (a bit excessive for our system with max 80 inodes and 56 data blocks)



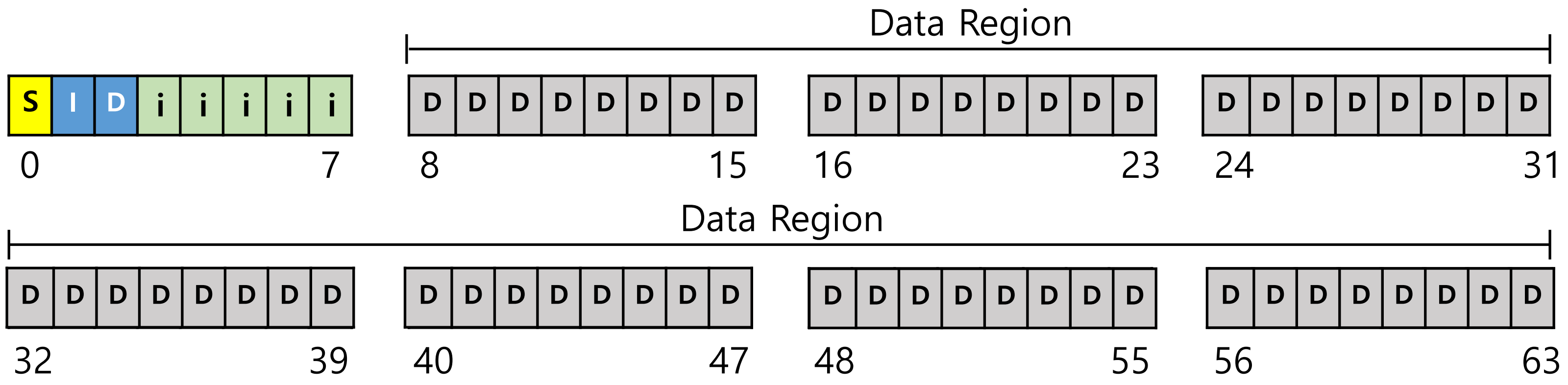
One block left



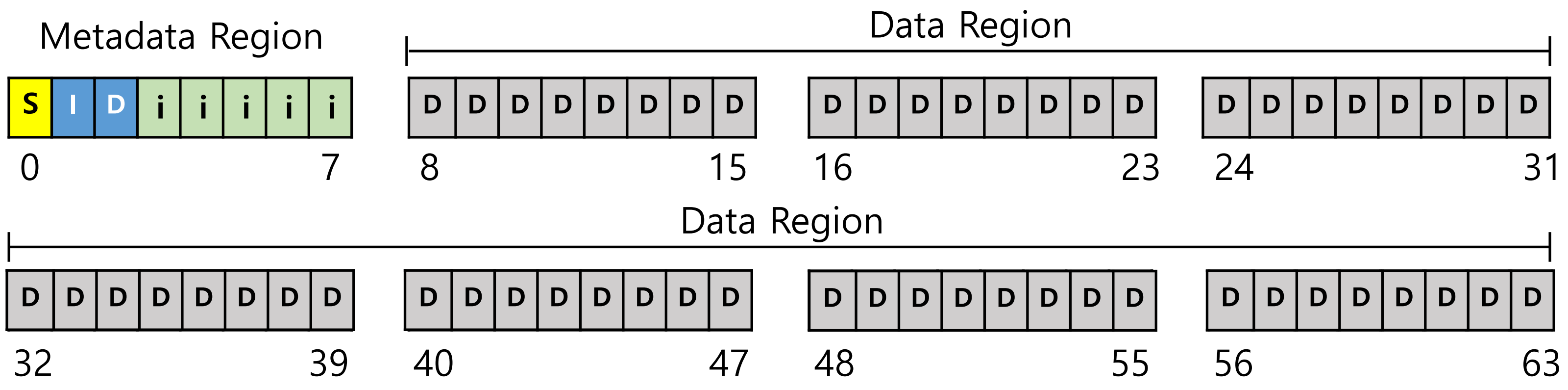
Superblock

Contains **file system metadata**

- #inodes
- #data blocks
- Start of inodes table
- …



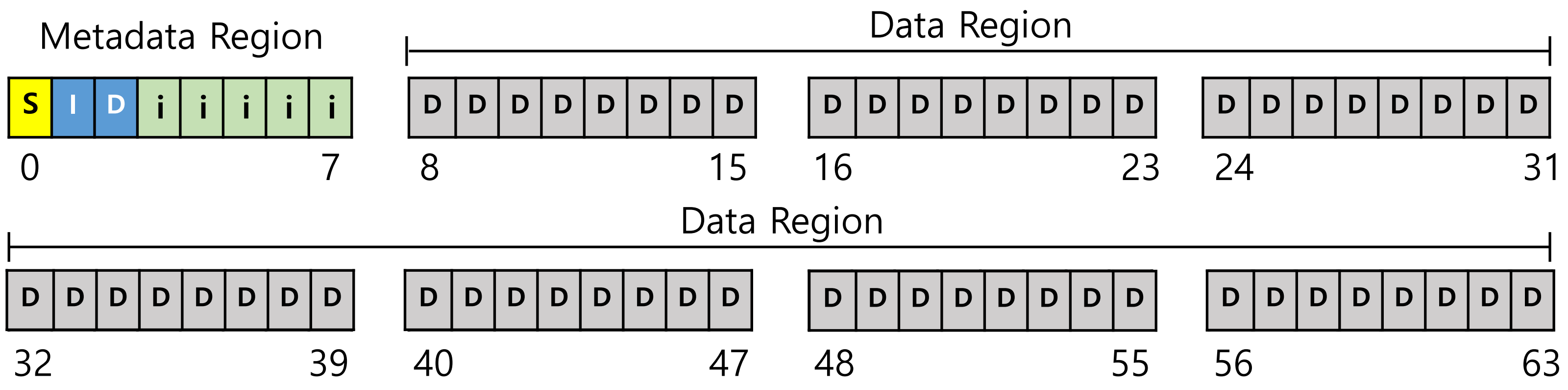
Disk Data Structures



Disk Data Structures

One important question remains:

How do we allocate files to blocks?



(and the converse: How do we handle free space?)

Remember: Inode

- Tracks file metadata:
  - type (file or dir?)
  - uid (owner)
  - rwx (permissions)
  - size (in bytes)
  - **blocks occupied by file**
  - timing information (creation time, last access time)
  - links_count (# paths)

We want to know "good" strategies to map files to blocks.

User Data Allocation

Strategies

- Contiguous
- Extent-based
- Linked list
- File-Allocation Tables (FAT)
- Indexed
- Multi-level Indexed

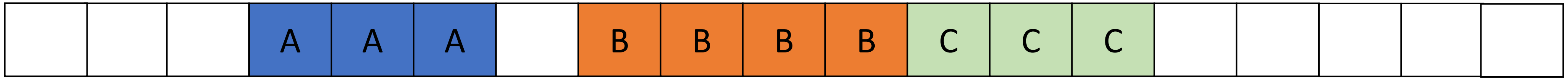
User Data Allocation

Questions

- Amount of fragmentation (external and internal)
- Ability to grow file over time
- Sequential access performance
- Random access performance
- Metadata overhead
  - i.e., “wasted space” to persistently store metadata

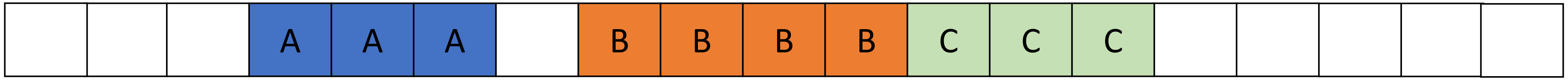
Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk
Metadata: Starting block + size of file



Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk
Metadata: Starting block + size of file



Fragmentation

Growing files

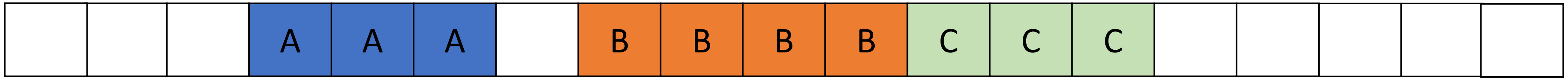
Sequential access

Random access

Metadata overhead

Contiguous Allocation

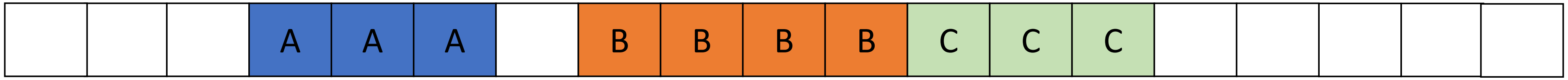
Strategy: Allocate file data blocks contiguously on disk
Metadata: Starting block + size of file



Fragmentation	High fragmentation (many unusable holes) ☹ ☹
Growing files	May not be able to grow without moving file ☹ ☹
Sequential access	Excellent ☺ ☺
Random access	Simple calculation ☺
Metadata overhead	Low ☺

Contiguous Allocation

Strategy: Allocate file data blocks contiguously on disk
Metadata: Starting block + size of file

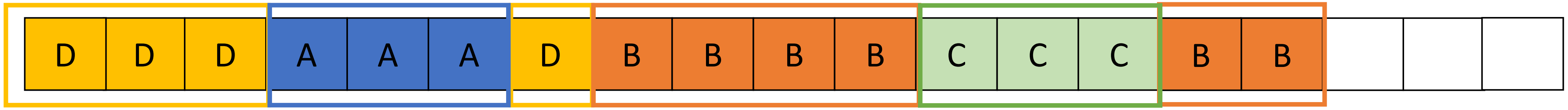


Fragmentation	High fragmentation (many unusable holes) ☹ ☹
Growing files	May not be able to grow without moving file ☹ ☹
Sequential access	Excellent ☺ ☺
Random access	Simple calculation ☺
Metadata overhead	Low ☺

Impractical

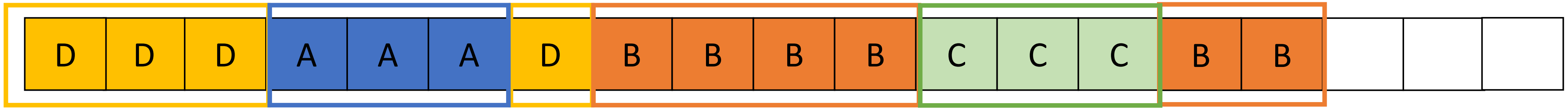
Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file
Metadata: Array of extents. Each entry contains extent starting block and size



Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file
Metadata: Array of extents. Each entry contains extent starting block and size



Fragmentation

Growing files

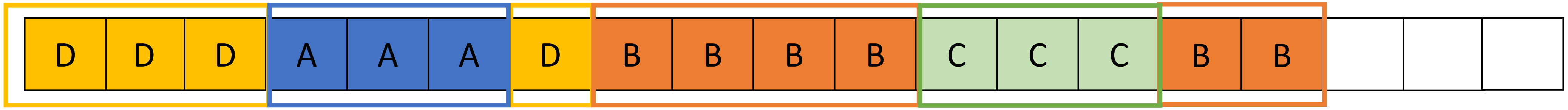
Sequential access

Random access

Metadata overhead

Extent-based Allocation

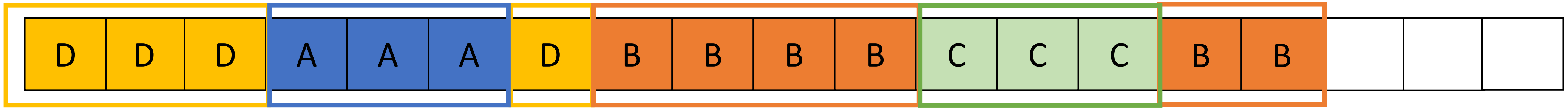
Strategy: Allocate multiple contiguous regions (called **extents**) per file
Metadata: Array of extents. Each entry contains extent starting block and size



Fragmentation	Helps external fragmentation
Growing files	Can grow ☺
Sequential access	Good performance ☺
Random access	Simple calculation ☺
Metadata overhead	Can get large if many extents. ☹

Extent-based Allocation

Strategy: Allocate multiple contiguous regions (called **extents**) per file
Metadata: Array of extents. Each entry contains extent starting block and size

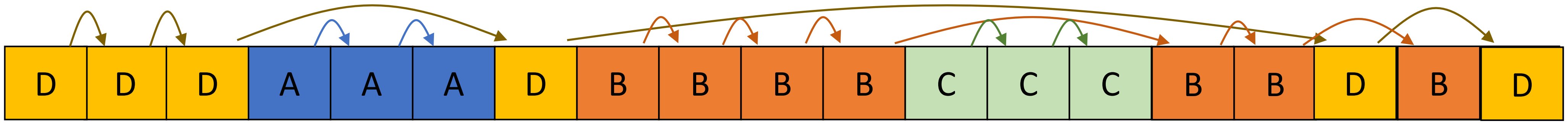


Fragmentation	Helps external fragmentation
Growing files	Can grow ☺
Sequential access	Good performance ☺
Random access	Simple calculation ☺
Metadata overhead	Can get large if many extents. ☹

Common practice in Linux

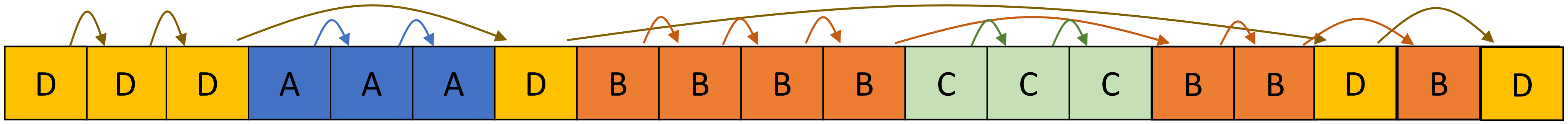
Linked-List Allocation

Strategy: Allocate linked-list of blocks
Metadata: Location of first block of file, plus each block contains pointer to next block.



Linked-List Allocation

Strategy: Allocate linked-list of blocks
Metadata: Location of first block of file, plus each block contains pointer to next block.



Fragmentation

Growing files

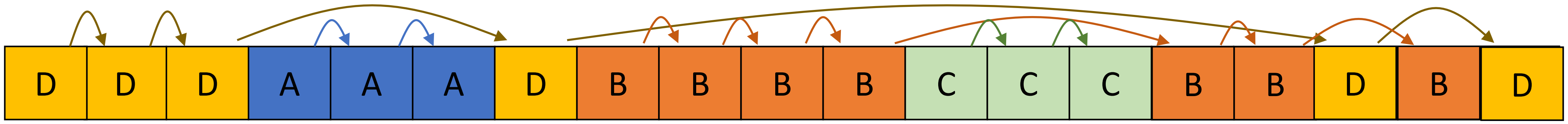
Sequential access

Random access

Metadata overhead

Linked-List Allocation

Strategy: Allocate linked-list of blocks
Metadata: Location of first block of file, plus each block contains pointer to next block.

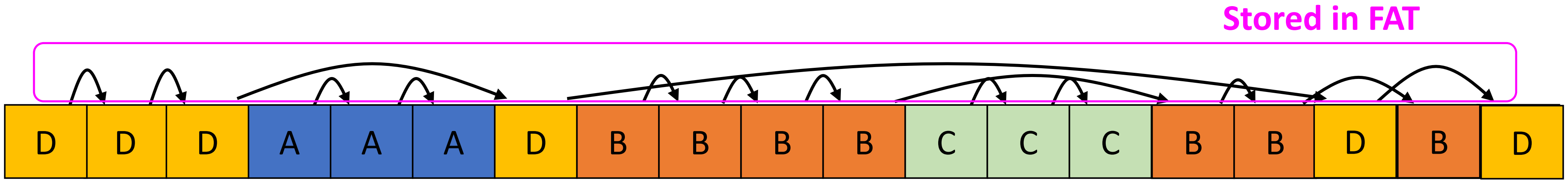


Fragmentation	No external fragmentation, low internal fragmentation ☺
Growing files	Can grow easily ☺
Sequential access	Depends on data layout
Random access	Slow ☹☹
Metadata overhead	Waste space on pointers in data blocks. ☹

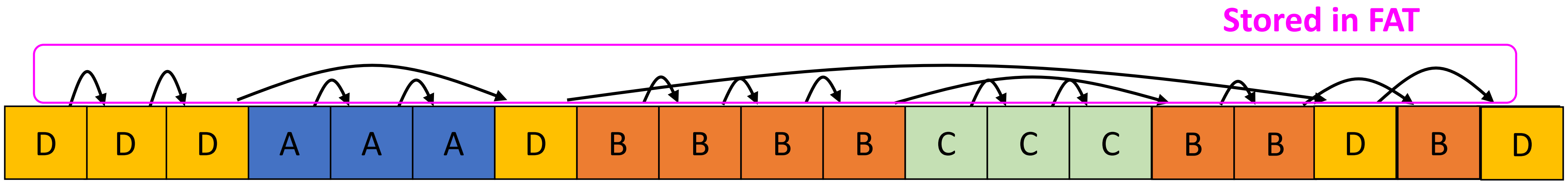
File-Allocation Tables (FAT)

Strategy: Keep links information for all files in a data structure on disk, called the **file allocation table (FAT)**. *Optimization: the FAT can be cached in memory.*

Metadata: Location of first block of file, plus FAT table itself.

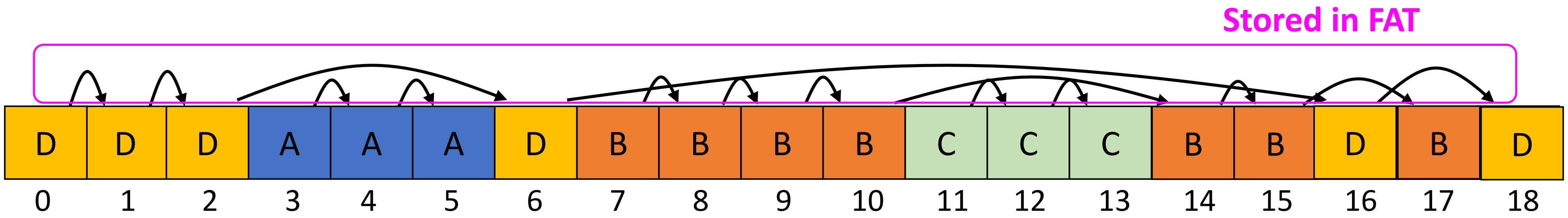


File-Allocation Tables (FAT)



Draw the FAT, assuming a linked-list structure.

File-Allocation Tables (FAT)



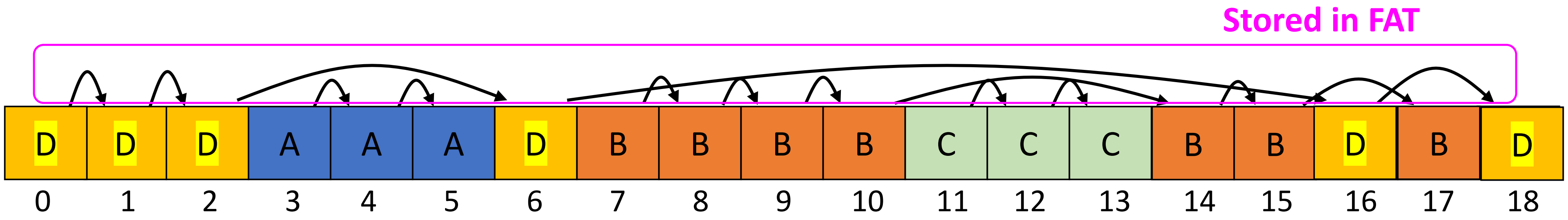
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	
1	
2	
3	
4	
5	
6	

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	
17	
18	

File-Allocation Tables (FAT)



Draw the FAT, assuming a linked-list structure.

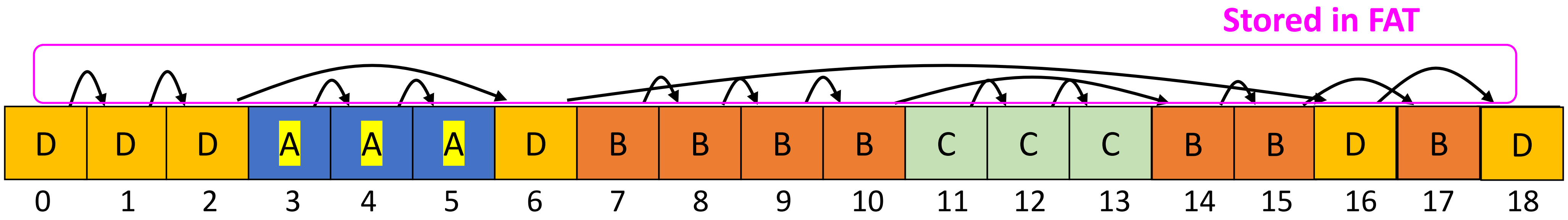
Block index	Next block
0	**1**
1	**2**
2	**6**
3	
4	
5	
6	**16**

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	**18**
17	
18	**-1**

convention to signal end of file

File-Allocation Tables (FAT)



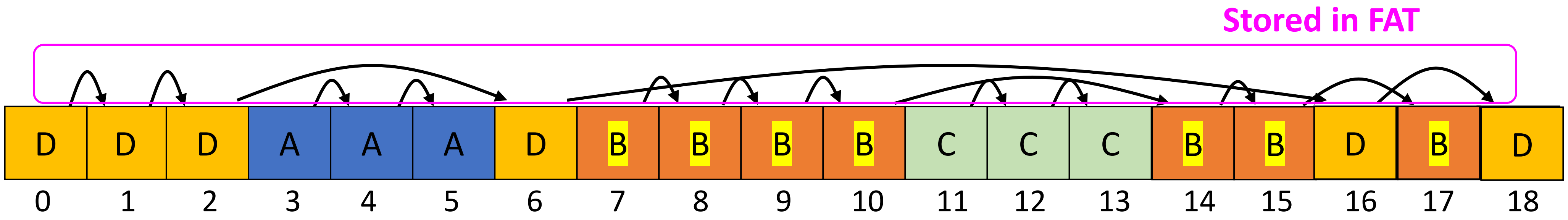
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	**4**
4	**5**
5	**-1**
6	16

Block index	Next block
7	
8	
9	
10	
11	
12	

Block index	Next block
13	
14	
15	
16	18
17	
18	-1

File-Allocation Tables (FAT)



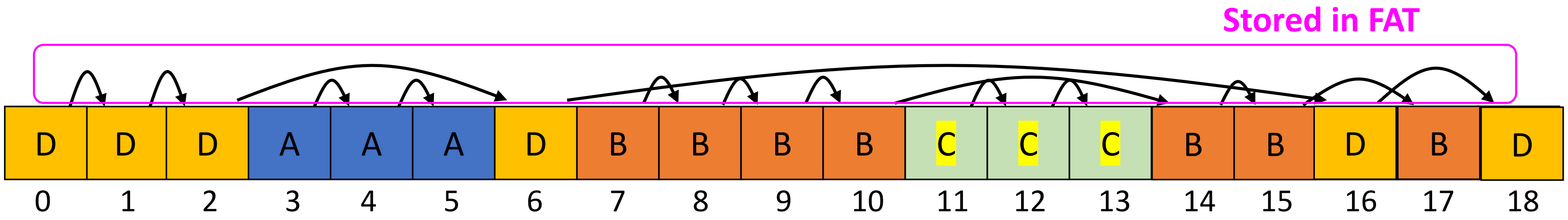
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Block index	Next block
7	**8**
8	**9**
9	**10**
10	**14**
11	
12	

Block index	Next block
13	
14	**15**
15	**17**
16	18
17	**-1**
18	-1

File-Allocation Tables (FAT)



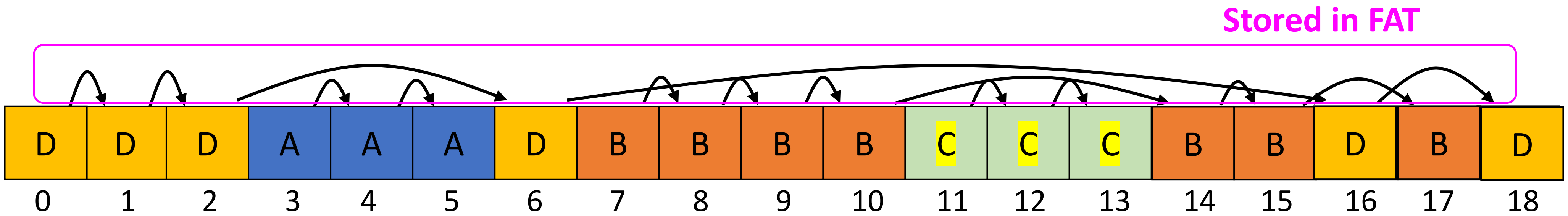
Draw the FAT, assuming a linked-list structure.

Block index	Next block
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Block index	Next block
7	8
8	9
9	10
10	14
11	**12**
12	**13**

Block index	Next block
13	**-1**
14	15
15	17
16	18
17	-1
18	-1

File-Allocation Tables (FAT)



Draw the FAT, assum

Block index	Next
0	1
1	2
2	6
3	4
4	5
5	-1
6	16

Note 1: **Metadata**

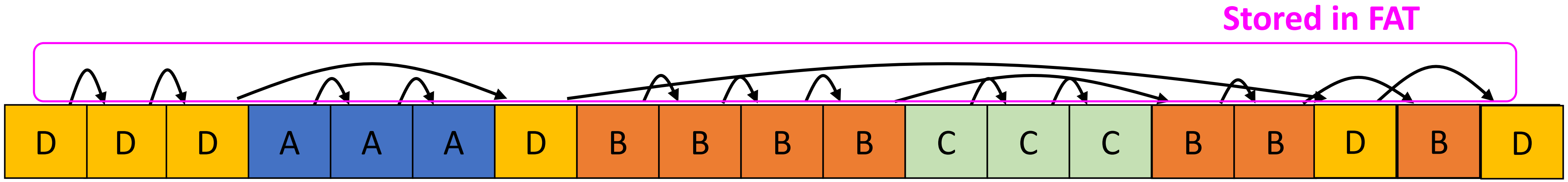
- -1 tells us where a file ends.
- How do we know where files start?
  - **Need to remember position of first block for each file**

Note 2: **FAT Data Structure**

- This example has a linked list FAT
- FAT implementation can use other data structures too

File-Allocation Tables (FAT)

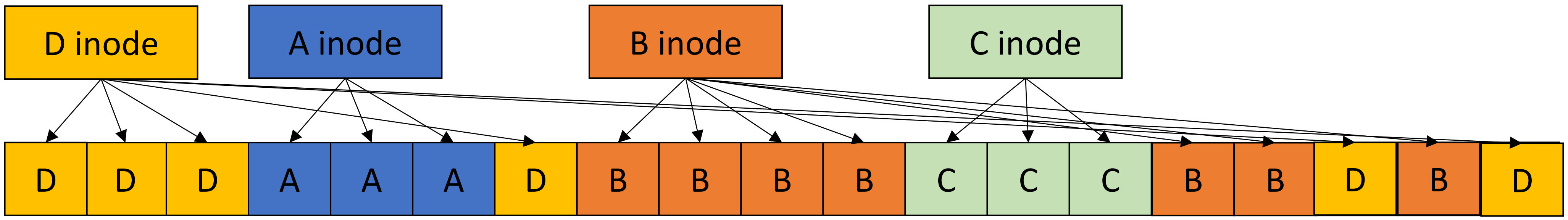
Strategy: Keep links information for all files in a data structure on disk, called the **file allocation table (FAT)**. *Optimization: the FAT can be cached in memory.*
Metadata: Location of first block of file, plus FAT table itself.



Fragmentation	Same as linked list ☺
Growing files	Same as linked list ☺
Sequential access	Same as linked list
Random access	Significantly better than linked list if FAT cached in memory ☺
Metadata overhead	Low for small files. What about large files? ☹

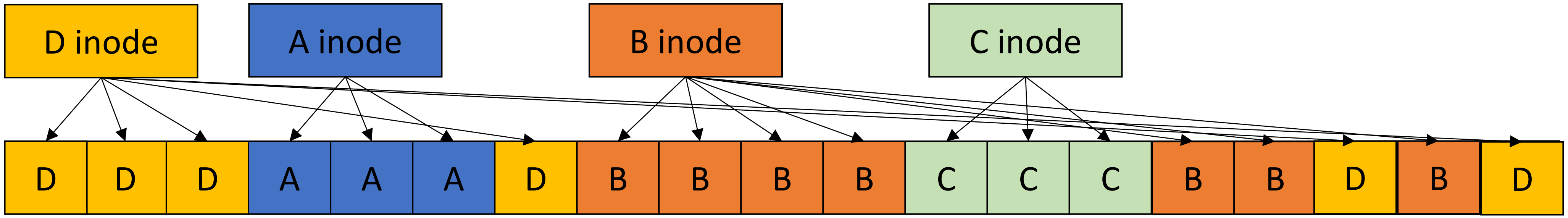
Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode.** Cap at maximum N pointers.
Metadata: Index for each file.



Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode.** Cap at maximum N pointers.
Metadata: Index for each file.



Fragmentation

Growing files

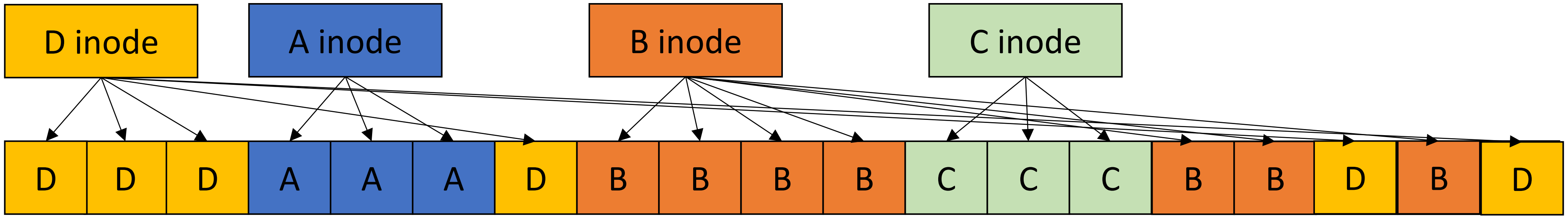
Sequential access

Random access

Metadata overhead

Indexed Allocation

Strategy: Keep pointers to blocks of file in **an index in the file's inode.** Cap at maximum N pointers.
Metadata: Index for each file.



Fragmentation	No external fragmentation, low internal fragmentation ☺
Growing files	Can grow easily ☺
Sequential access	Efficient ☺
Random access	Efficient ☺
Metadata overhead	Low for small files. What if file size > N * size of data block?

Can be combined with extent allocation

Indexed Allocation with Indirect Blocks

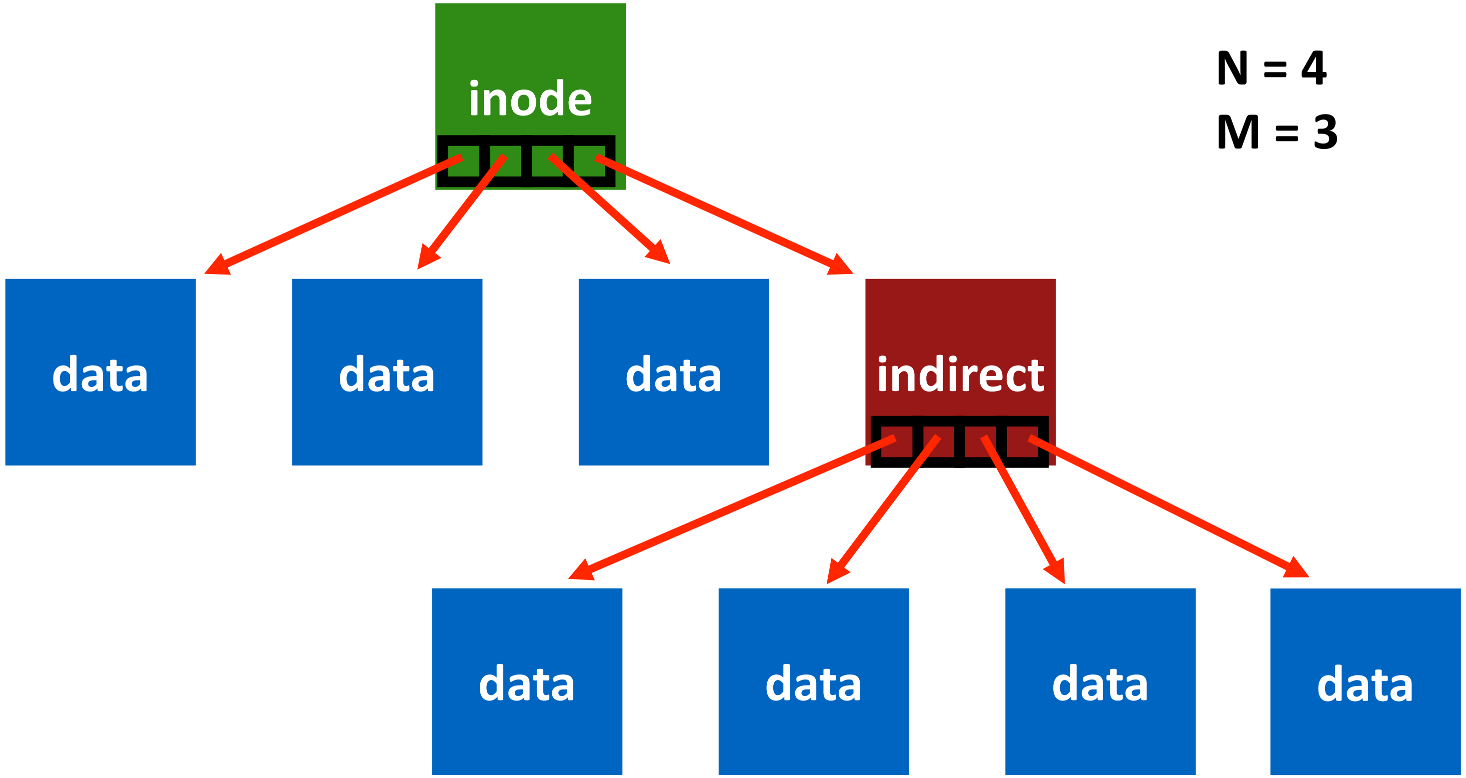
- N pointers in inode block
- First M (< N) point to first M **data blocks**
- Blocks M+1 to N point to ***indirect blocks***

Indirect blocks

  - Do not contain data
  - But pointers to subsequent data blocks

- Double-indirect blocks also possible

Indexed Allocation with Indirect Blocks



N = 4
M = 3

Indexed Allocation with Indirect Blocks

- Same advantages as indexed allocation

plus

- Possible to extend to very large files

What About Directories?

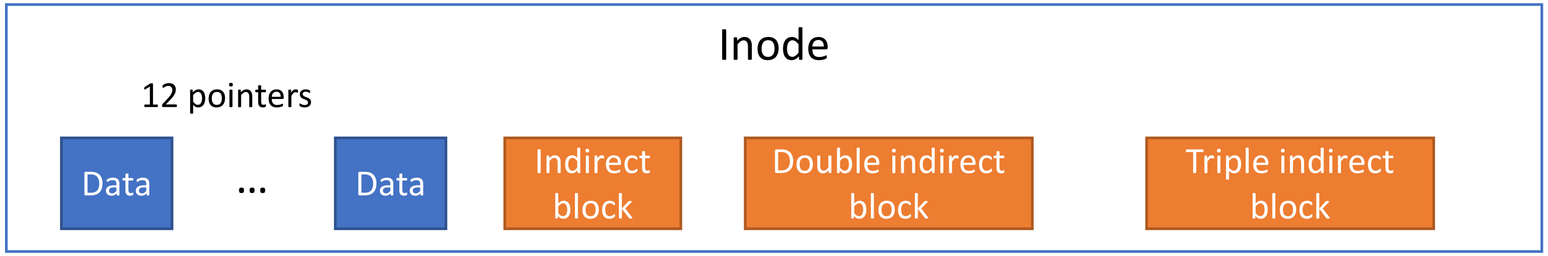
- Directories stored as files

Let’s practice!

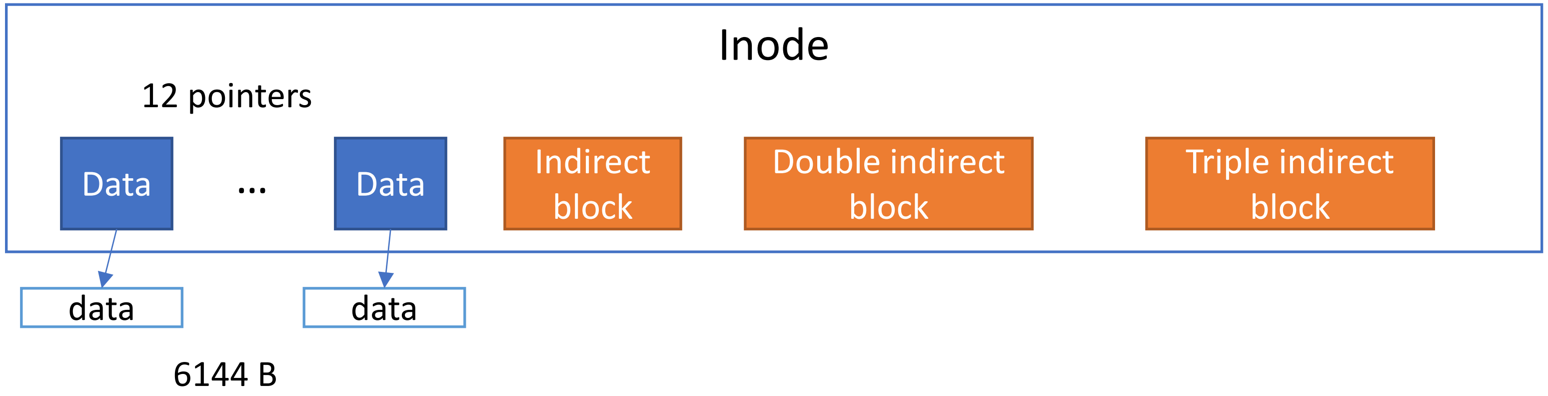
Assume we have a disk with 512-byte disk sectors and 4-byte disk addresses. The file system uses inodes with **12 direct block pointers, 1 indirect block pointer, and 1 double-indirect block pointer, and 1 triple-indirect block pointer**. The block size is identical to the sector size of the disk.

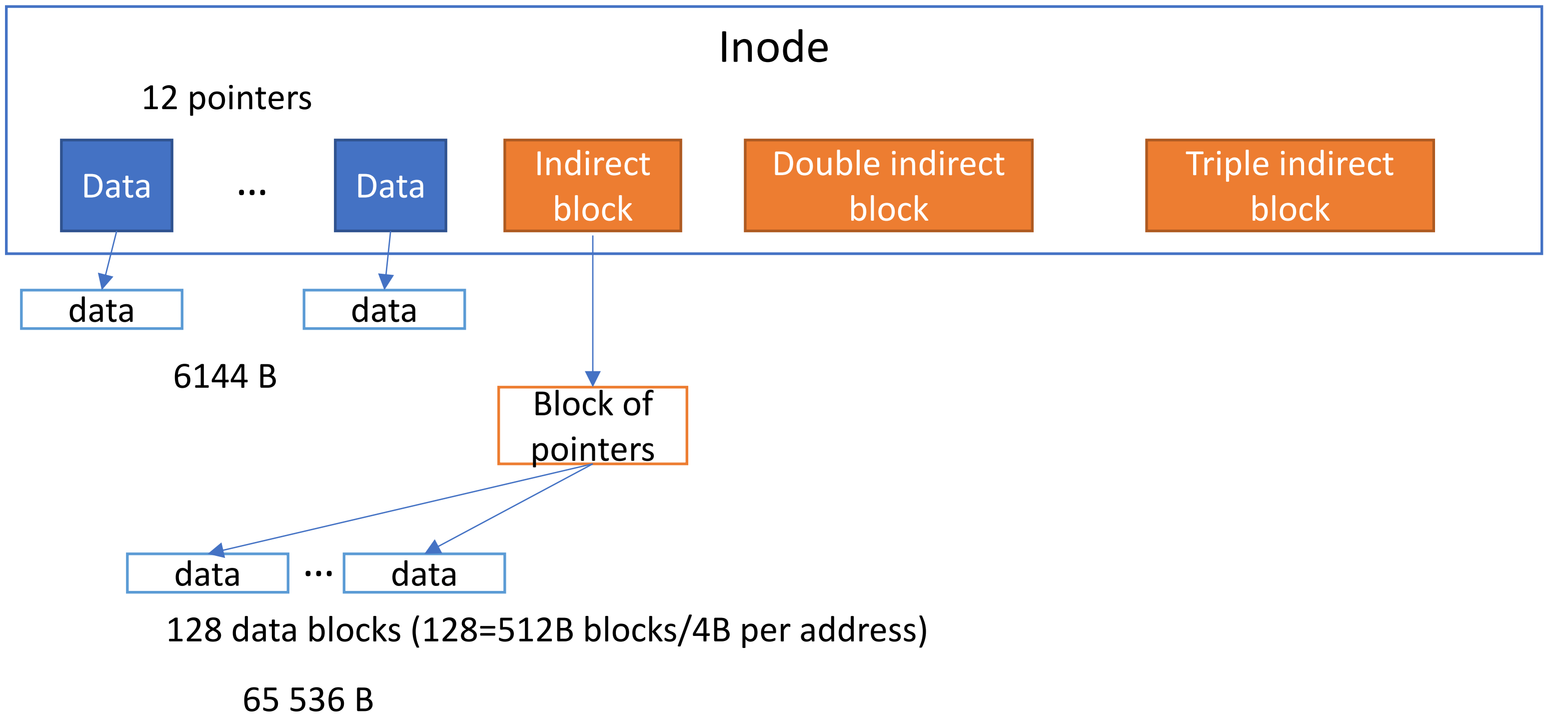
A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses? Explain your answer by **drawing a diagram illustrating which disk accesses occur.**

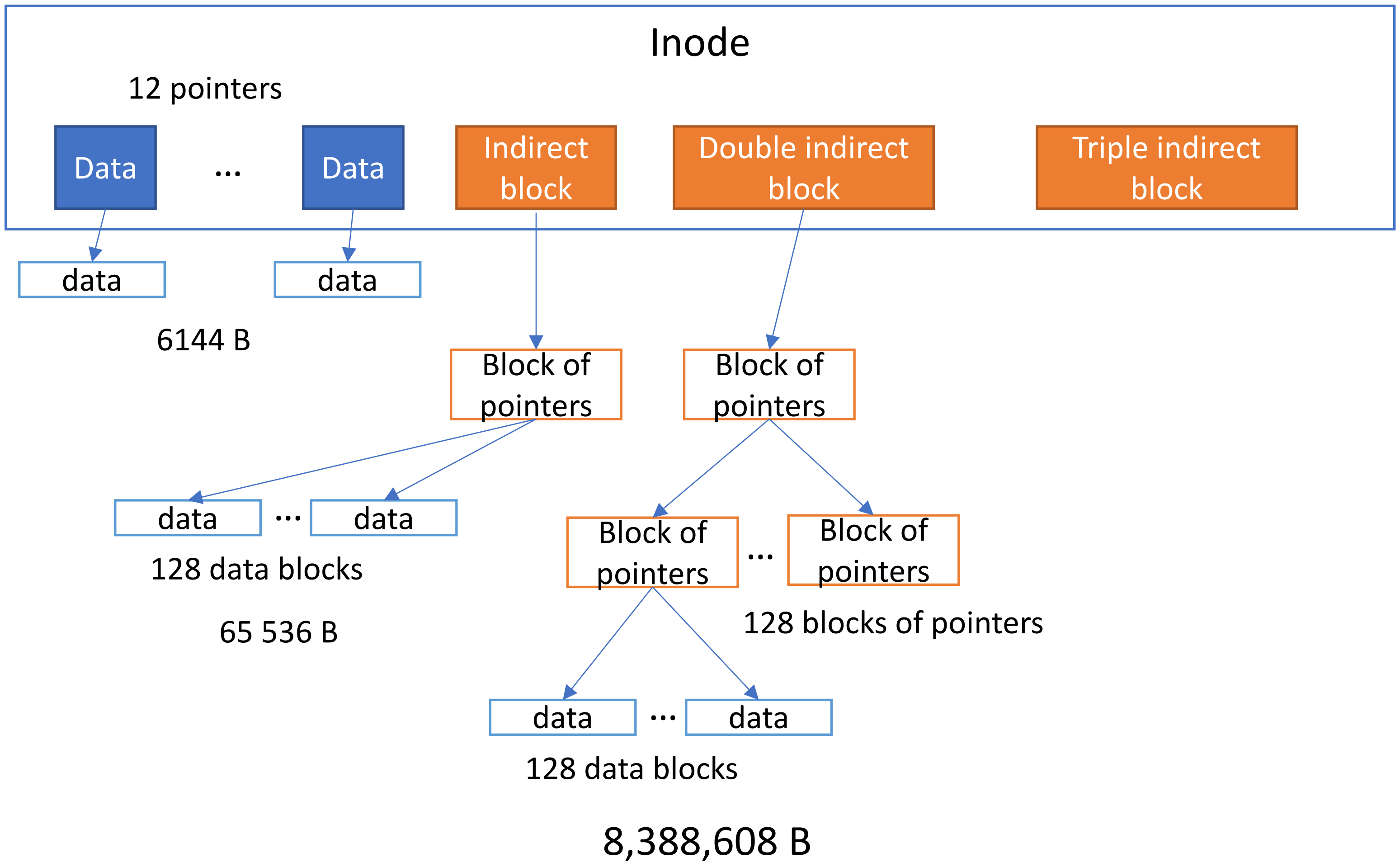
Assume that prior to the open, the inode of the file and the blocks belonging to the file are *not* in the cache, but that there is space in the cache to store all of them. You may also assume that there is no file system activity going on, other than the accesses to this particular file.

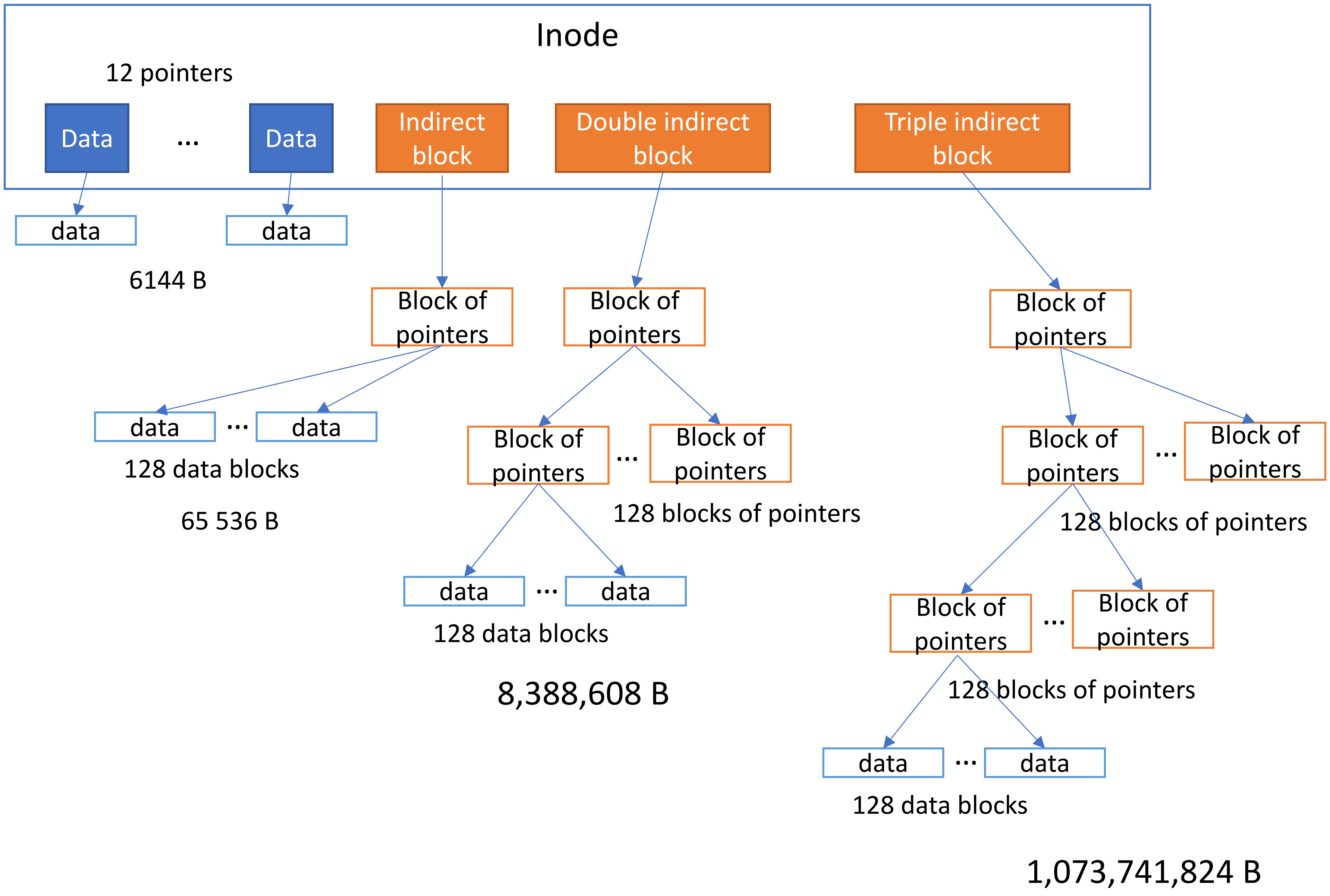


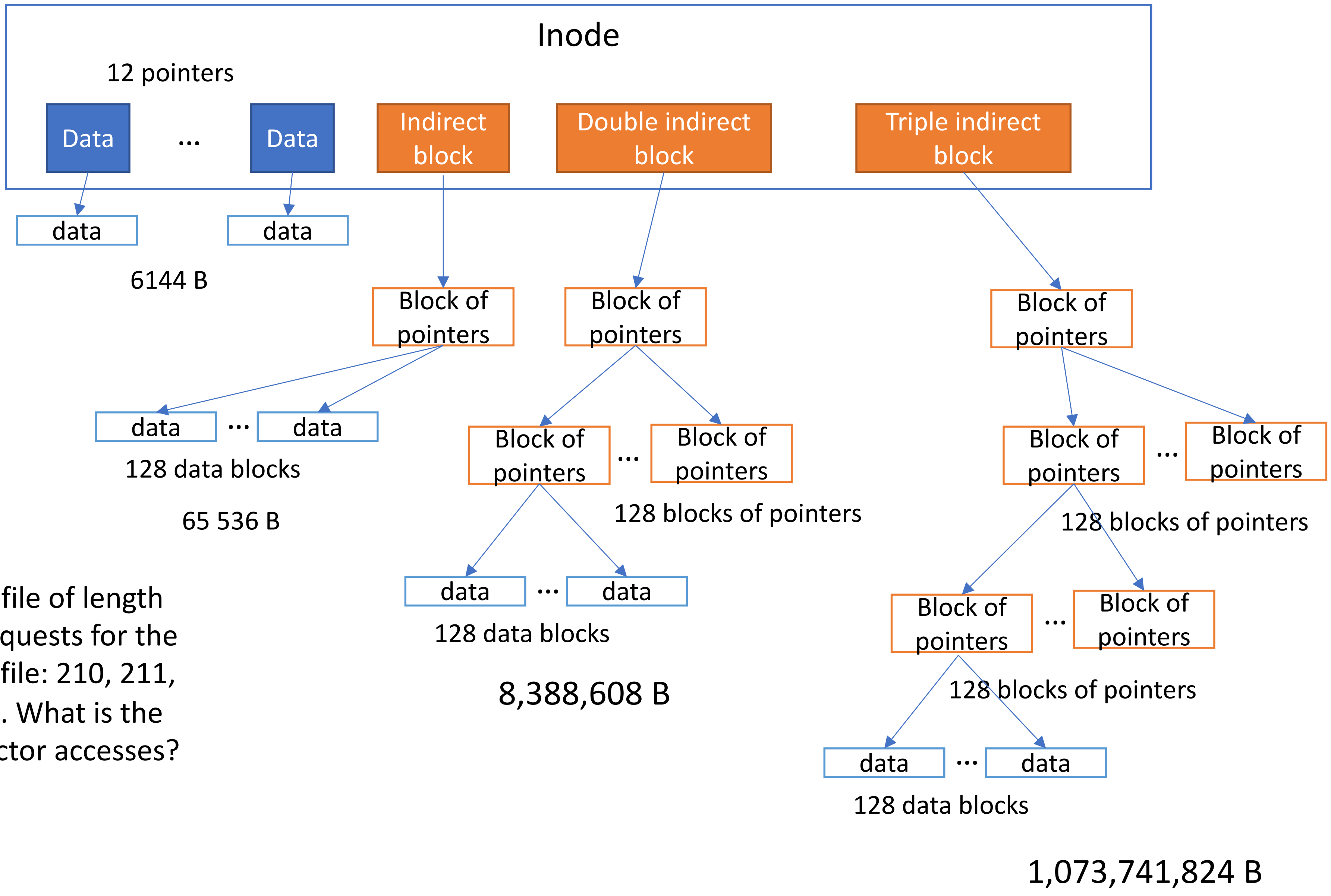
Inode
12 pointers
Data
...
Data
Indirect block
Double indirect block
Triple indirect block


Inode
12 pointers
Data
...
Data
Indirect block
Double indirect block
Triple indirect block
data
data
6144 B

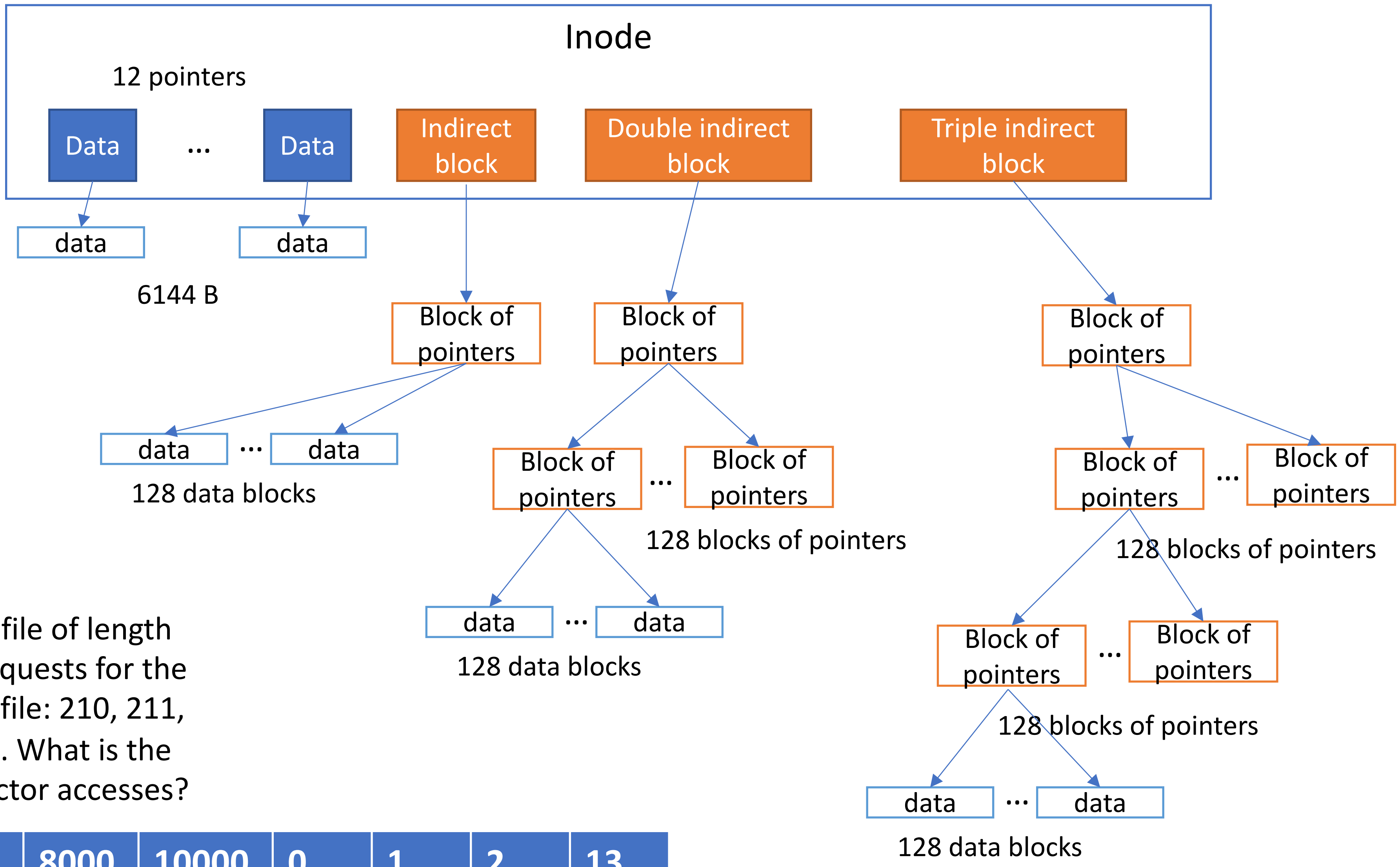

Inode
12 pointers
Data
...
Data
Indirect block
Double indirect block
Triple indirect block
data
data
6144 B
Block of pointers
data
...
data
128 data blocks (128=512B blocks/4B per address)
65 536 B


Inode
12 pointers
Data
...
Data
Indirect block
Double indirect block
Triple indirect block
data
data
6144 B
Block of pointers
Block of pointers
data
...
data
128 data blocks
65 536 B
Block of pointers
...
Block of pointers
128 blocks of pointers
data
...
data
128 data blocks
8,388,608 B


Inode
12 pointers
Data
...
Data
Indirect block
Double indirect block
Triple indirect block
data
data
6144 B
Block of pointers
Block of pointers
Block of pointers
data
...
data
128 data blocks
65 536 B
Block of pointers
...
Block of pointers
128 blocks of pointers
Block of pointers
...
Block of pointers
128 blocks of pointers
data
...
data
128 data blocks
8,388,608 B
Block of pointers
...
Block of pointers
128 blocks of pointers
data
...
data
128 data blocks
1,073,741,824 B

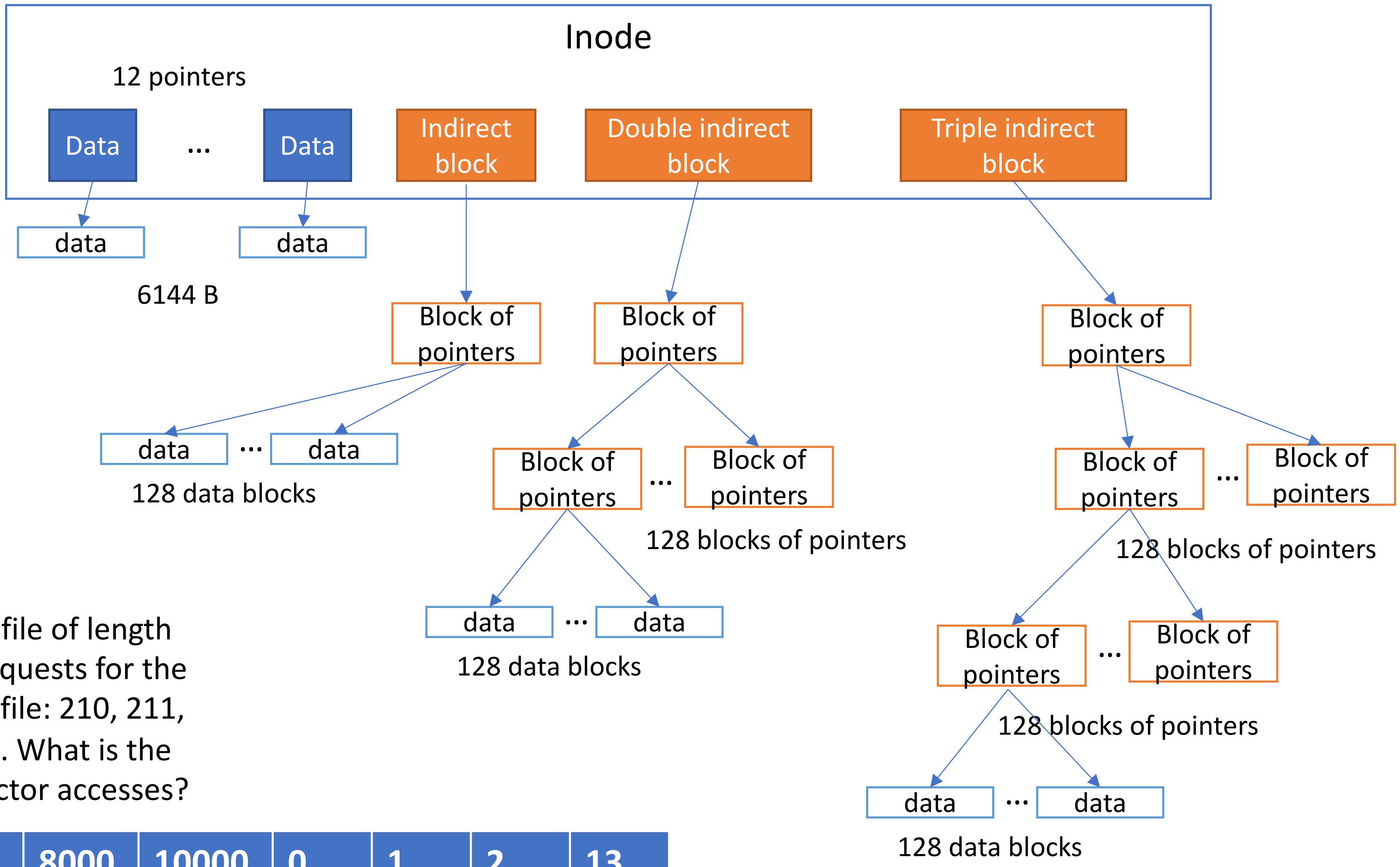


A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?



A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?

inode	210	211	8000	10000	0	1	2	13



A user process opens a file of length 1GB, and issues read requests for the following blocks of this file: 210, 211, 8000, 10000, 0, 1, 2, 13. What is the total number of disk sector accesses?

inode	210	211	8000	10000	0	1	2	13
1	3	1	2	2	1	1	1	2

14 disk accesses

inode	210	211	8000	10000	0	1	2	13
1	3	1	2	2	1	1	1	2

14 disk accesses

Explanations:

- Inode: 1 access because the problem statement assumption is that nothing is cached.
- 210: lives in the double indirect block. Nothing is cached apart from the inode, so we have 3 accesses: 2 for the pointer blocks and 1 for the data block
- 211: lives in the double indirect block, neighboring 210. So, the 2 pointer blocks (and the inode) are cached from the previous accesses. We only need 1 disk access for the data block.
- 8000: lives in the double indirect block. The first pointer block (and inode) are cached from reading 210, and 211. But, the second pointer block and data block are not cached, so we have 2 disk accesses
- 10000: same explanation as for 8000 (note that 8000 and 10000 are more than 128 blocks apart, so the second pointer block is not cached from when we read 8000, as it was the case for reading 211 after 210).
- 0, 1, 2: 1 access because they live in the direct blocks.
- 13: lives in the indirect block. We thus have 2 accesses: one for the pointer block and one for the data block.

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 40, 41, 45.

https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).