Operating System: File System Implementation

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

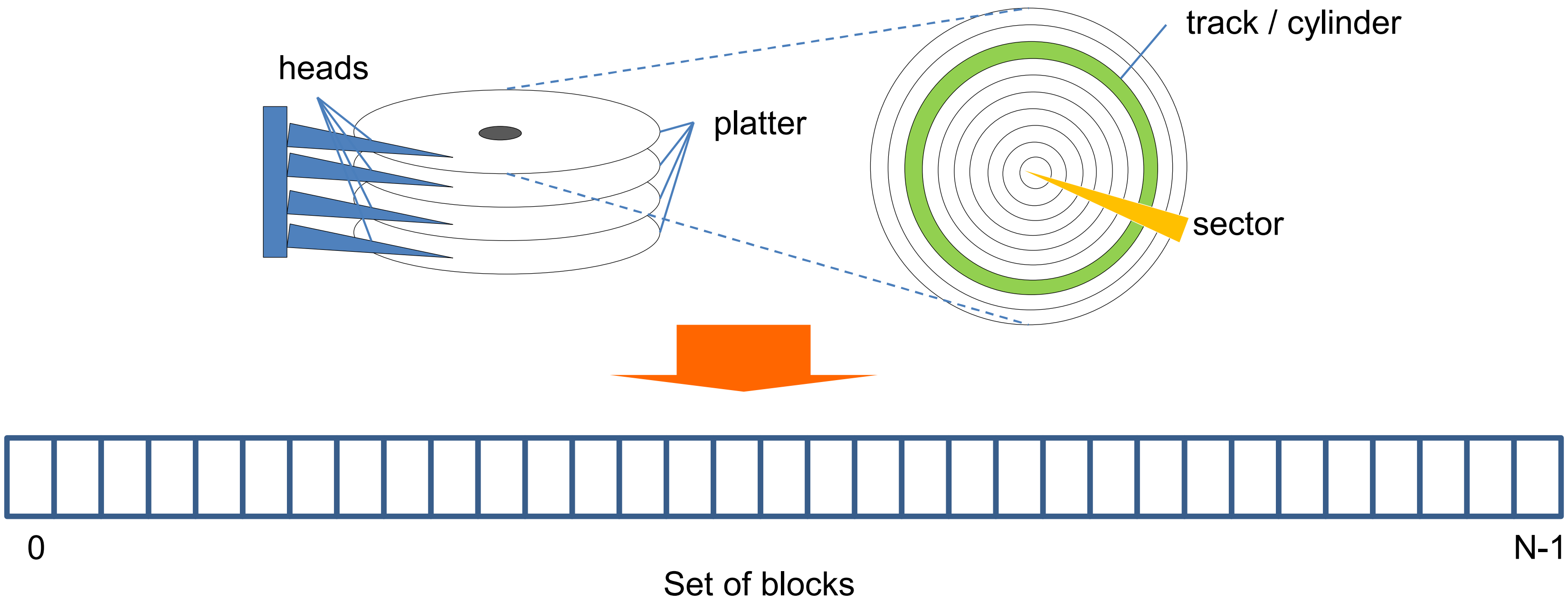
What is File System Implementation?

- File system of the user's viewpoint
  - How to show the file system to user?
  - File system interface
  - File, directory, attribute, and operation
  - E.g. tree structure

- File system of the storage management viewpoint
  - How to map the logical file system to the storage device?
  - File system implementation
  - Layout, data structures, and algorithms
  - It is required to understand the storage internals

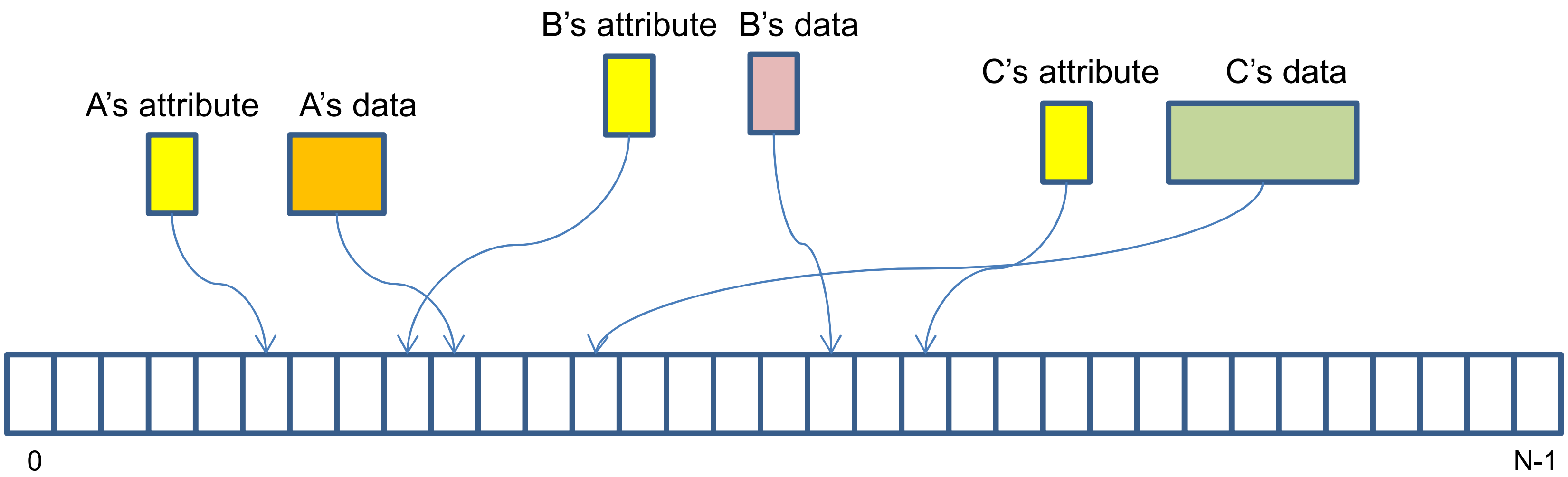
What is File System Implementation? (Cont.)

- File system regards a storage device as a sequence of blocks
  - Data is transferred between disk and memory in units of blocks
  - Each block has one or more sectors, which is typically 512 bytes



What is File System Implementation? (Cont.)

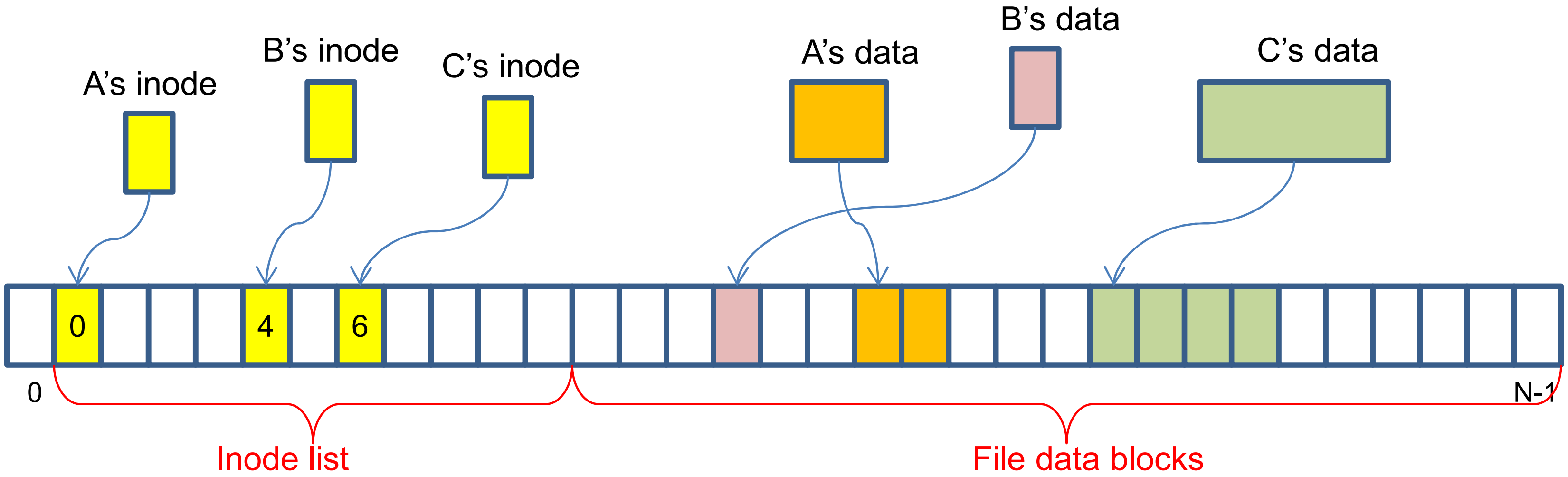
- File system implementation
  - Both attribute and file data should be stored for each file
    - Attributes – size, permission, owner, time, etc.



  - How to allocate the storage for both attributes and file data?

What is File System Implementation? (Cont.)

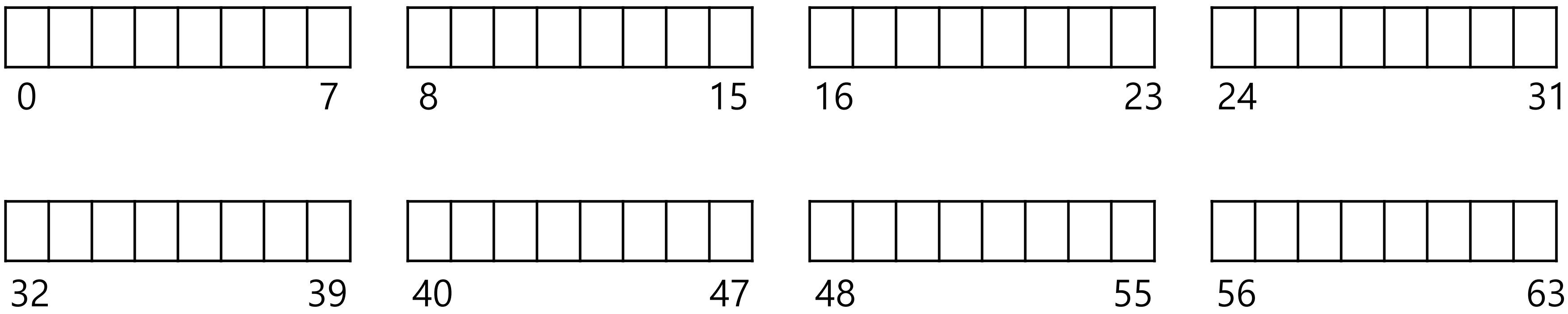
- Traditional file system
  - Inode
    - Attribute + indexes for file data
  - Inode region and file data region are separated



  - The sizes of inodes are same, but those of file data are different
    - Inodes can be accessed via i-number
  - How to allocate the storage for file data?
    - contiguous / linked / indexed allocation

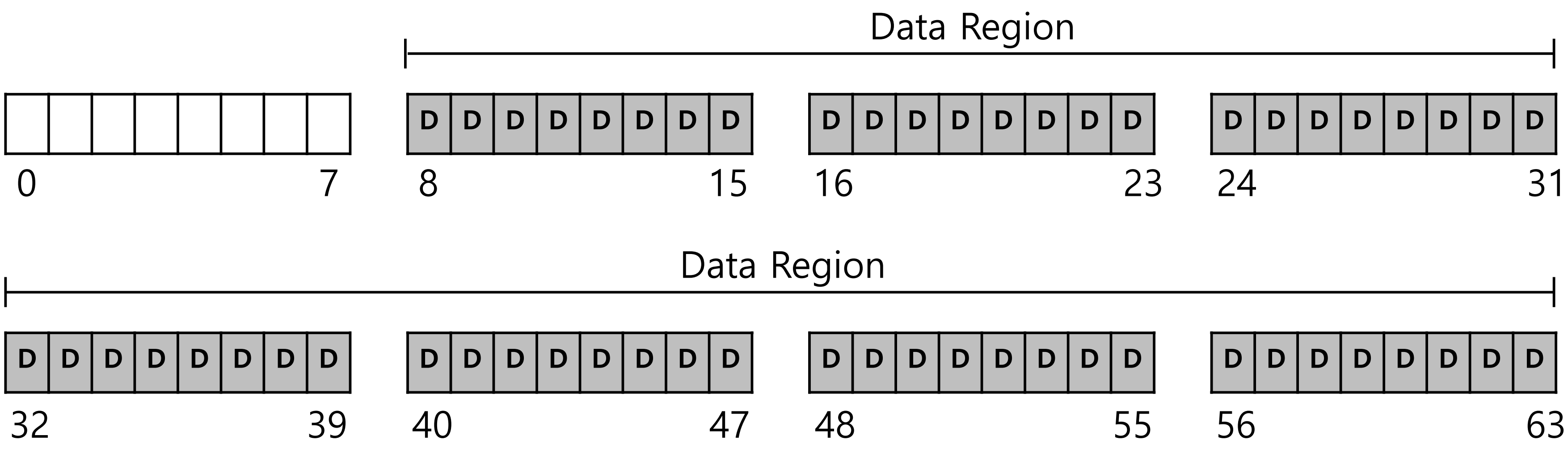
VSFS: Overall Organization

- Very Simple File System
  - Divide the disk into blocks (Block size is 4 KB)
  - Block size is a multiple of sector size
  - The blocks are addressed from `0 to N -1`



VSFS: Data region in file system

- Reserve data region to store user data



  – File system has to track which data block comprise a file, the size of the file, its owner, etc.

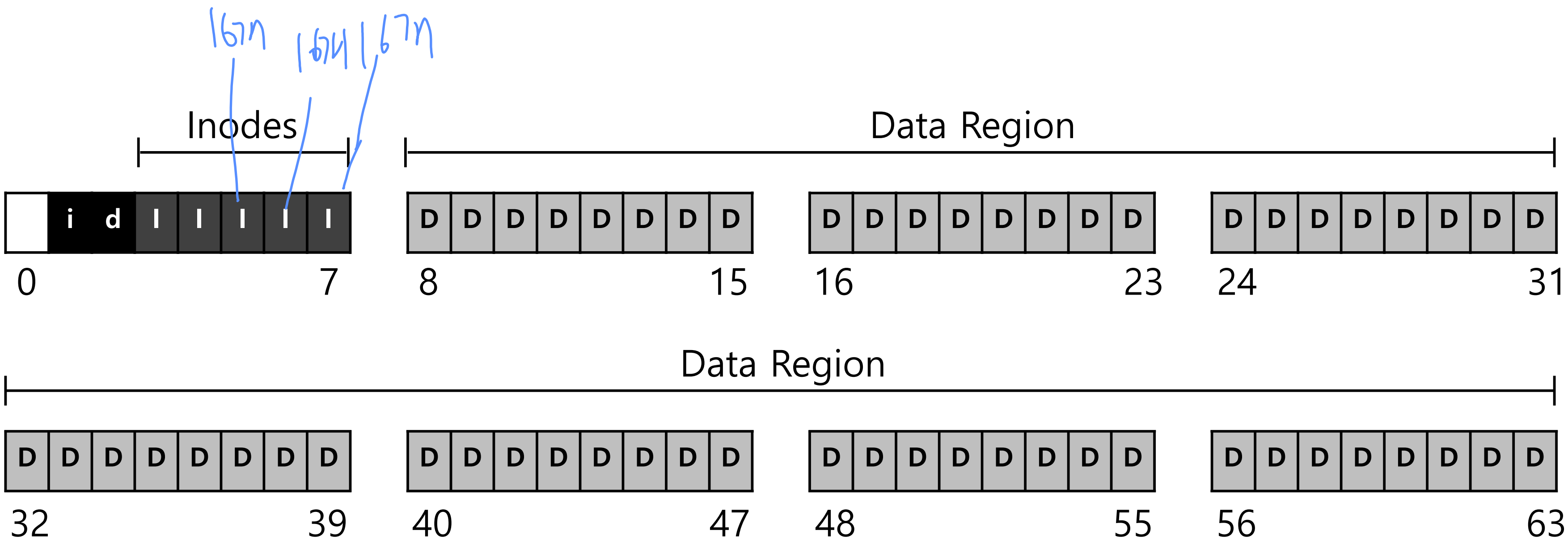
How we store these inodes in file system?

VSFS: Inode table in file system

블록하나가 4KB니까

4KB/256B = 16개

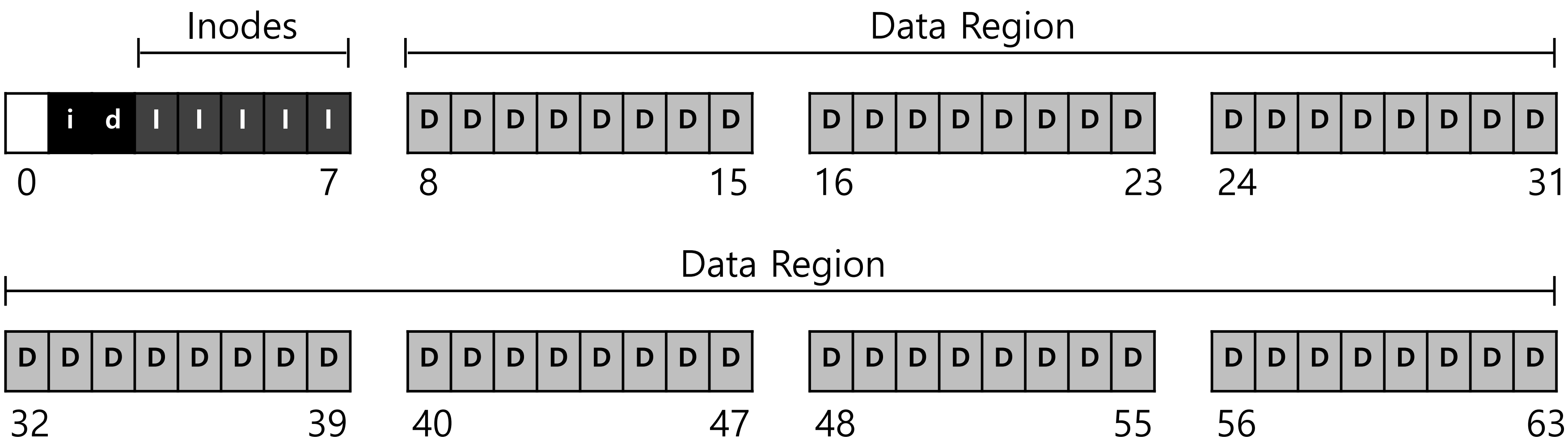
- Reserve some space for inode table
  - This holds an array of on-disk inodes
  - Ex) inode tables : 3 ~ 7, inode size : 256 bytes
    - ➢ 4-KB block can hold 16 inodes
    - ➢ The filesystem contains 80 inodes (maximum number of files)



VSFS: allocation structures

- This is to track whether inodes or data blocks are free or allocated
- Use bitmap, each bit indicates free(0) or in-use(1)
  - **data bitmap**: for data region
  - **inode bitmap**: for inode table

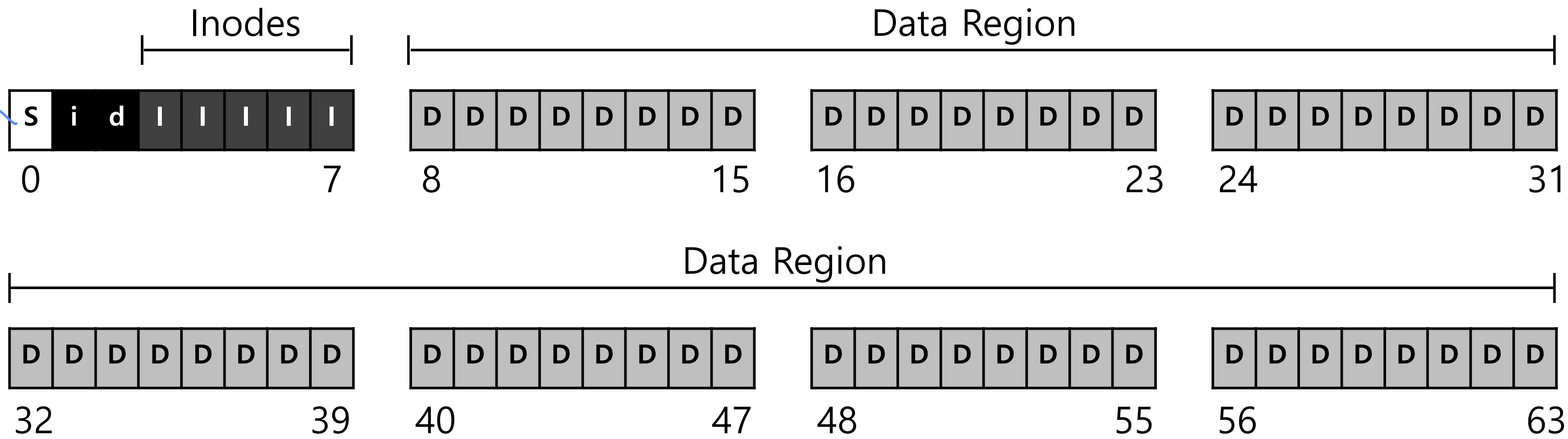
데이터가 블록에 존재하는지 확인하기 위해



VSFS: Superblock

- Super block contains this information for particular file system
  - Ex) The number of inodes, begin location of inode table. etc

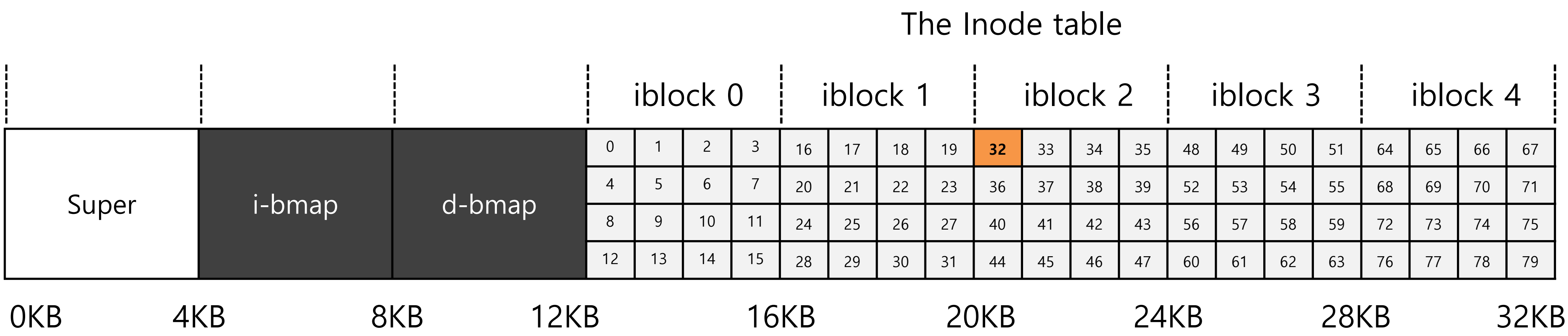
가장먼저 여기를 읽고 초기화같은 작업진행



  - Thus, when mounting a file system, OS will read the superblock first, to initialize various information

Inode allocation

- Each inode is referred to by inode number
  - by inode number, file system calculate where the inode is on the disk
  - Ex) inode number: 32
    - ➢ Calculate the offset into the inode region (32 x sizeof(inode)) = 8192
    - ➢ Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



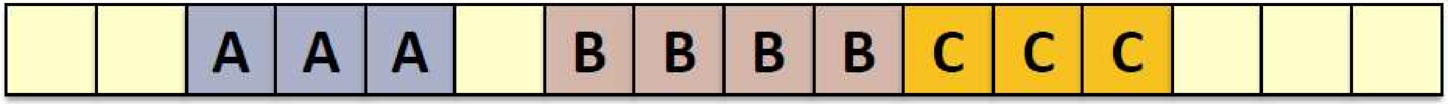
Data allocation

- How to map files to disk blocks?
  - Similar to mapping variable-sized address spaces to physical memory
  - Same principle: map logical abstraction to physical resources

- Issues
  - The amount of fragmentation (mostly external)
  - Ability to grow file over time
  - Performance of sequential accesses
  - Speed to find data blocks for random accesses
  - Metadata overhead to track data blocks

Contiguous Allocation

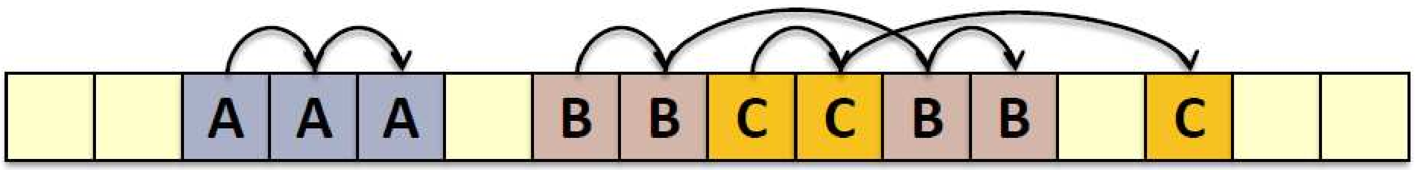
- Allocate each file to contiguous blocks on disk
  - Metadata: <starting block #, length>
  - Feasible and widely used for CD-ROMs
  - Example: IBM OS/360



  - Horrible external fragmentation (needs periodic compaction)
  - May not be able to grow file without moving
  - Excellent performance for sequential accesses
  - Simple calculation to perform random accesses
  - Little overhead for metadata

Linked Allocation

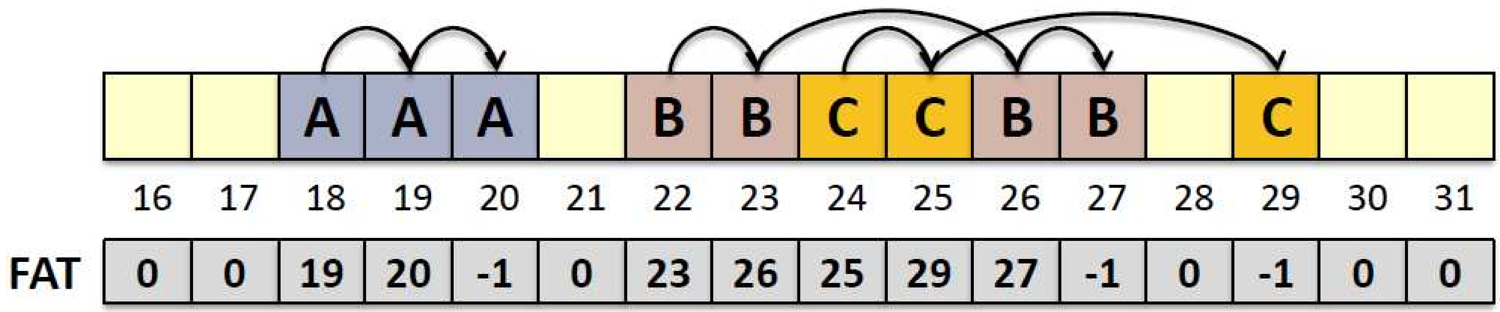
- Allocate linked-list of fixed-sized blocks
  - Metadata: <starting block #>
  - Each block contains pointer to next block
  - Example: TOPS-10, Alto



  - No external fragmentation
  - File can grow easily
  - Sequential access performance depends on data layout
  - Poor random access performance
  - Waste pointer per block (fragile -- it can be lost or damaged)

File Allocation Table (FAT)

- Variation of linked allocation
  - Keep linked-list information for all files in on-disk FAT
  - FAT is cached in main memory to avoid disk seeks
  - Metadata: <starting block #> + FAT
  - Example: MS-DOS, Windows (FAT12, FAT16, FAT32)

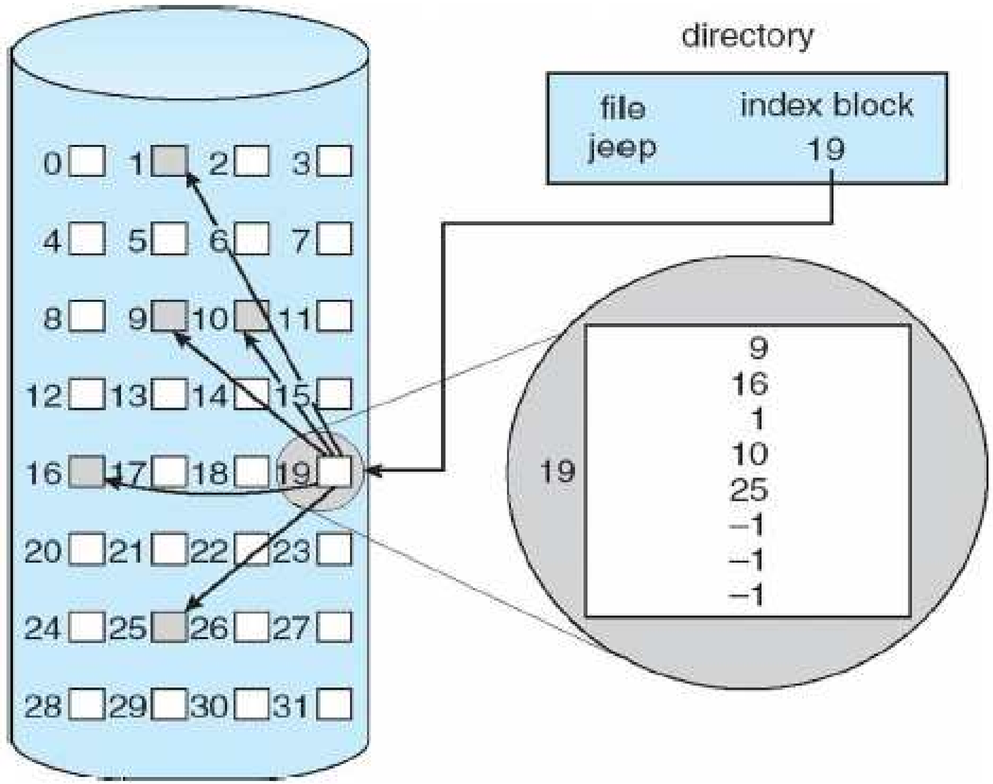


  - Improved random access performance
  - Scalability with larger file systems?

Indexed Allocation

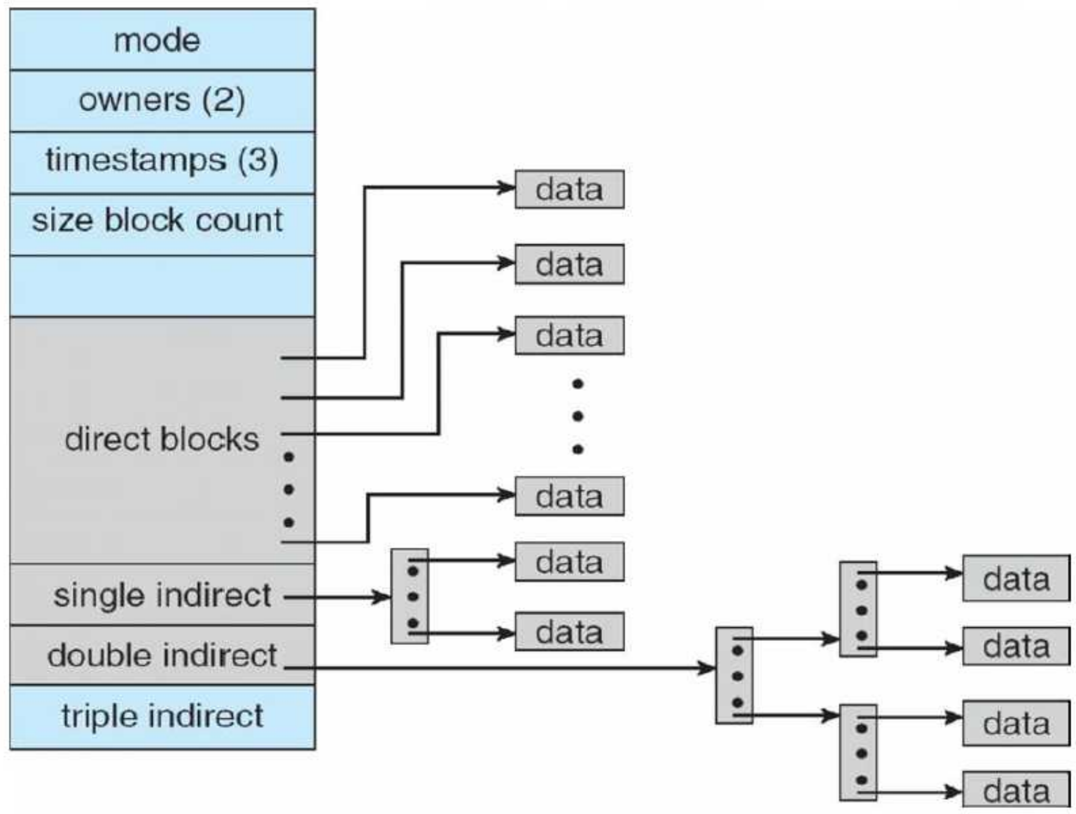
- Allocate fixed-sized blocks for each file
  - Metadata: An array of block pointers
  - Each block pointer points to the corresponding data block

- No external fragmentation
- File can grow easily up to max file size
- Sequential access performance depends on data layout
- Random accesses supported
- Large overhead for metadata: wasted space for unneeded pointers (most files are small)



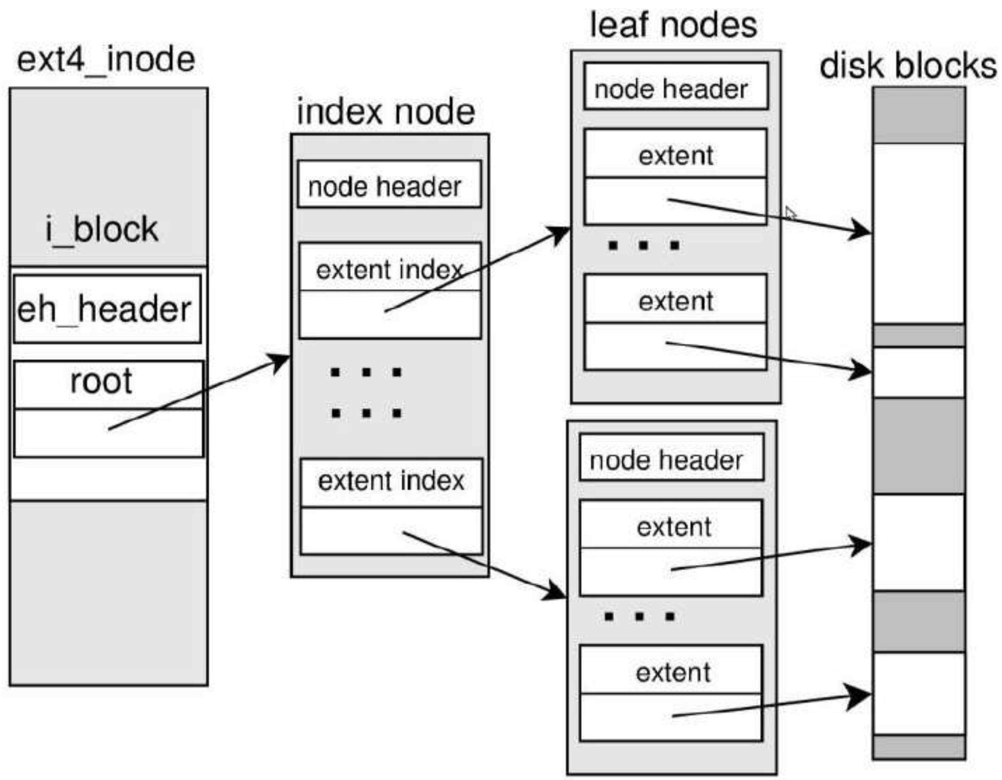
Multi-level Indexing

- Variation of indexed allocation
  - Dynamically allocate hierarchy of pointers to data blocks
  - Metadata: small number of direct pointers + indirect pointers
  - Example: Unix FFS, Linux Ext2/3
  - Does not waste space for unneeded pointers
  - Need to read indirect blocks of pointers to calculate addresses (extra disk read)
    - ➢ Keep indirect blocks cached in main memory



Extent-based Allocation

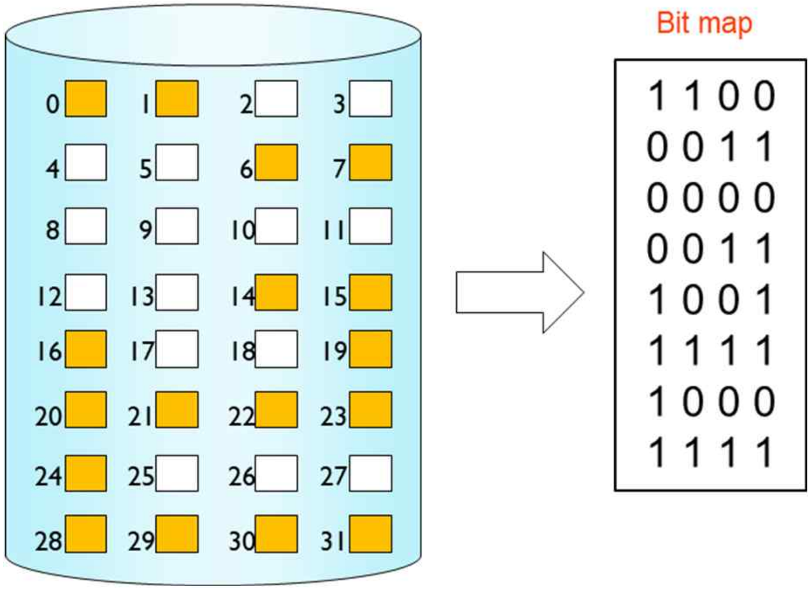
- Allocate multiple contiguous regions (extents) per file
  - Organize extents into multi-level tree structure (e.g. B+tree)
  - Each leaf node: <starting block #, extent size>
  - Example: Linux Ext4



- Reasonable amount of external fragmentation
- Still good sequential performance
- Some calculations needed for random accesses
- Relatively small metadata overhead

Free Space Management

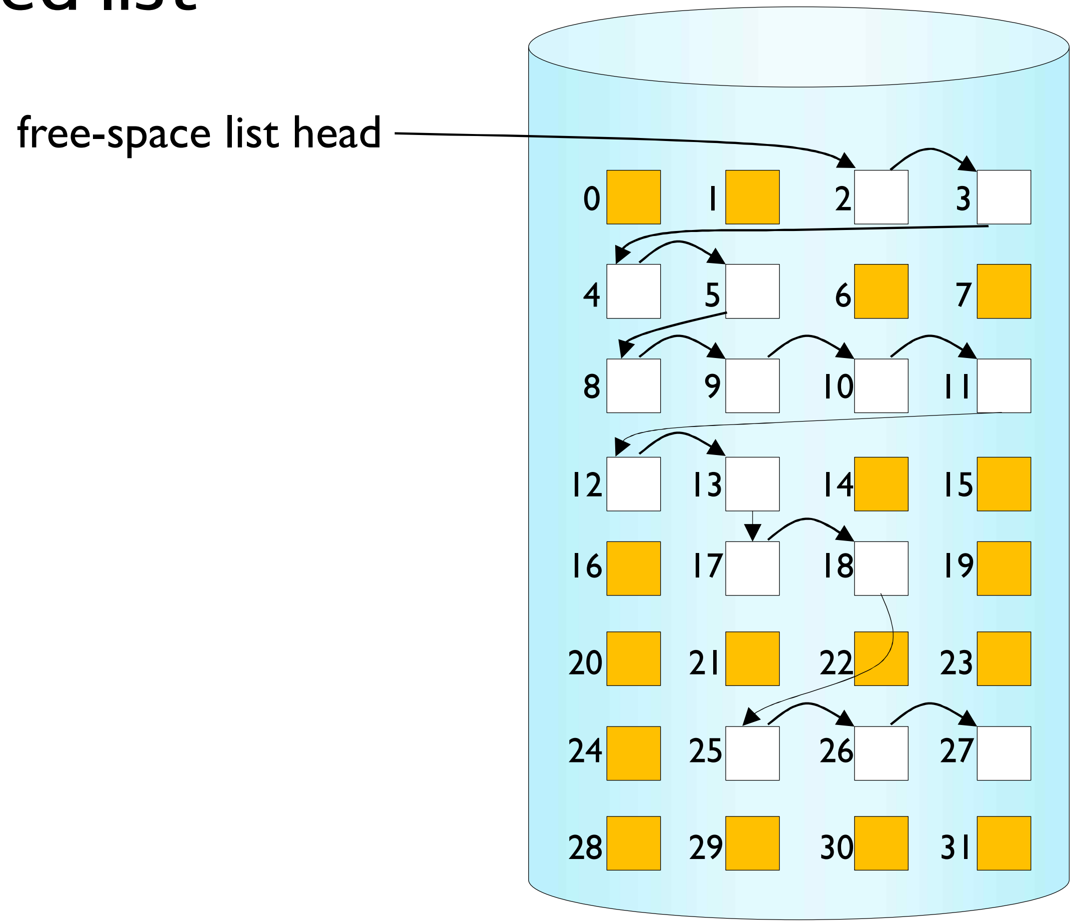
- Bit map (also called bit vector)
  - If block[i] is allocated, bit is 1; else, bit is 0
  - E.g.



  - Easy to get contiguous free blocks
  - Bit map requires extra space
    - e.g. block size = 2^{12} bytes (= 4KB)
    - disk size = 2^{40} bytes (= 1TB)
    - $n = 2^{40}/2^{12} = 2^{28}$ bits (= 32MB)

Free Space Management

- Linked list



  - No space overhead
  - Cannot get contiguous free blocks easily

Directory Implementation

- Linear list
  - (file name, pointer to the file)
  - Simple to implement
  - Time-consuming to search a file
  - B-tree can be used

- Hash Table
  - Linear list with hash data structure
  - Hash function converts "file name" to "pointer to the file's linear list"
  - Reduces directory search time
  - Collisions should be solved

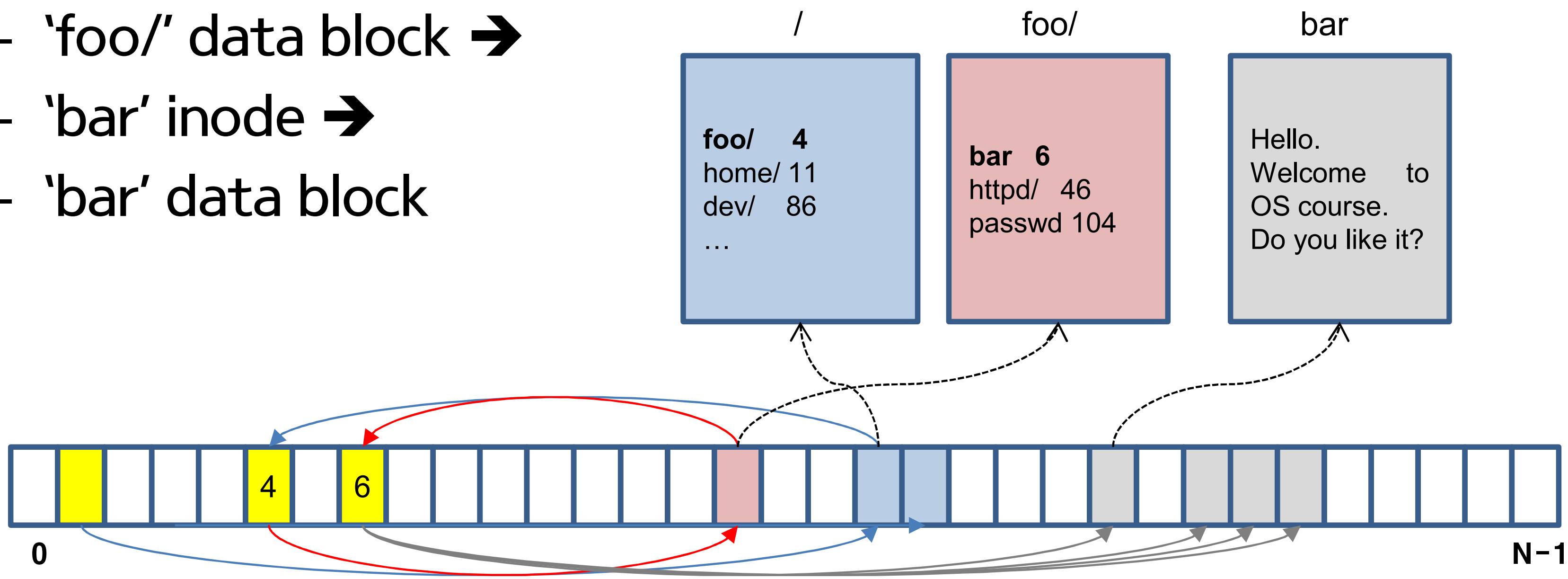
VSFS: Directory

- linear list of <file name, inode number>
  - Similar to Linux Ext2 directory
  - Supports variable-sized names
  - Example: `/dir`
    - ➢ Inode number for `/dir`?
    - ➢ Inode number for the root directory?

inode number	record length	name length	name								
5	4	2	.	\0	\0	\0					
2	4	3	.	.	\0	\0					
12	4	4	f	o	o	\0					
0	4	4	b	a	r	\0	*<deleted entry>*				
24	8	7	f	o	o	b	a	r	\0	\0	

Access Paths: Reading a File From Disk

- Issue an `open("/foo/bar", O_RDONLY)`,
  - '/' inode →
  - '/' data block →
  - 'foo/'inode →
  - 'foo/' data block →
  - 'bar' inode →
  - 'bar' data block



  - How can find the inode number of '/'?
    - ➢ In most Unix file systems, the root inode number is fixed

Access Paths: Reading a File From Disk

- Issue an `open("/foo/bar", O_RDONLY)`,
  - Traverse the pathname and thus locate the desired inode
  - Begin at the root of the file system `(/)`
    - ➢ In most Unix file systems, the root inode number is 2
  - Filesystem reads in the block that contains inode number 2
  - Look inside of it to find pointer to data blocks (contents of the root)
  - By reading in one or more directory data blocks, It will find "foo" directory
  - Traverse recursively the path name until the desired inode ("bar")
  - Check permissions, allocate a file descriptor for this process and returns file descriptor to user

Access Paths: Reading a File From Disk (Cont.)

- Issue `read()` to read from the file
  - Read in the first block of the file, consulting the inode to find the location of such a block
    - Update the inode with a new last accessed time
    - Update in-memory open file table for file descriptor, the file offset

- When file is closed:
  - File descriptor should be deallocated, but for now, that is all the file system really needs to do. No disk I/Os take place

Access Paths: Reading a File From Disk (Cont.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
								read		
					write					
read()					read					
									read	
					write					
read()					read					
										read
					write					

File Read Timeline (Time Increasing Downward)

Access Paths: Writing to Disk

- Issue `write()` to update the file with new contents
- File may allocate a block (unless the block is being overwritten)
  - Need to update data block, data bitmap
  - It generates five I/Os:
    - ➢ one to read the data bitmap
    - ➢ one to write the bitmap (to reflect its new state to disk)
    - ➢ two more to read and then write the inode
    - ➢ one to write the actual block itself
  - To create file, it also allocate space for directory, causing high I/O traffic

Access Paths: Writing to Disk (Cont.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write							write		
					write					
write()					read					
	read									
	write								write	
					write					
write()					read					
	read									write
	write									
					write					

File Creation Timeline (Time Increasing Downward)

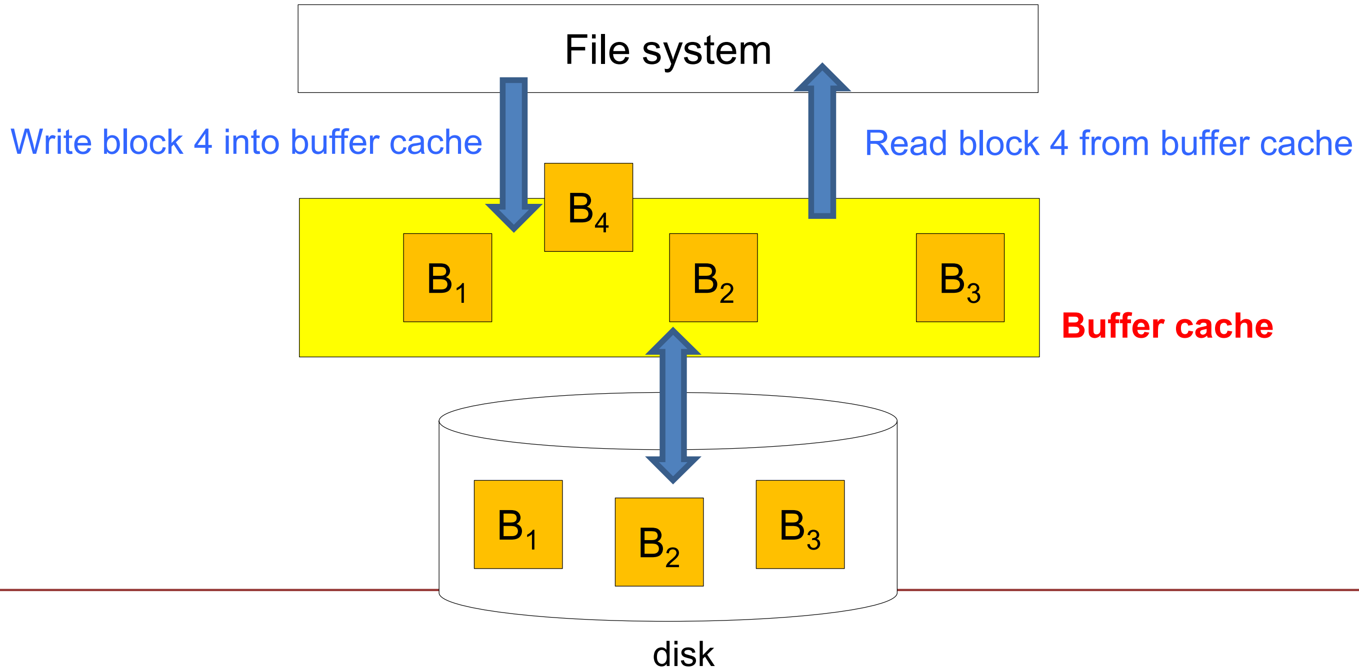
Performance and Recovery

- Efficiency
  - Keeps a file's data blocks near that file's inode block to reduce seek time
  - The "last access time" information in inode requires a block read and write
  - The size of pointers used to access data? 16bit or 32bit

- Performance
  - Buffer cache
    - ➢ Separate section of main memory for frequently used blocks
  - read-ahead
    - ➢ Techniques to optimize sequential access

Performance and Recovery (Cont.)

- Buffer cache
  - Whenever files are accessed, file data should be fetched from disks
  - However, the disk accesses incurs large I/O overhead
  - On file access patterns, data that are accessed once will be used again soon(temporal locality)
  - Buffer cache keeps the blocks that will be used again soon to reduce disk accesses



Performance and Recovery (Cont.)

- Inconsistency can occur
  - Some part of file system is kept in memory for speed up
  - What if system crashes? Not all data can be saved to disk
  - If directory or metadata (inode) are lost?

- Solutions for consistency
  - Consistency checker (e.g. fsck in UNIX)
    - Compare data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Synchronous write for important metadata
  - Journaling

- Use system programs to back up data
  - From disk to another storage device
  - Recover lost file or disk by restoring data from backup