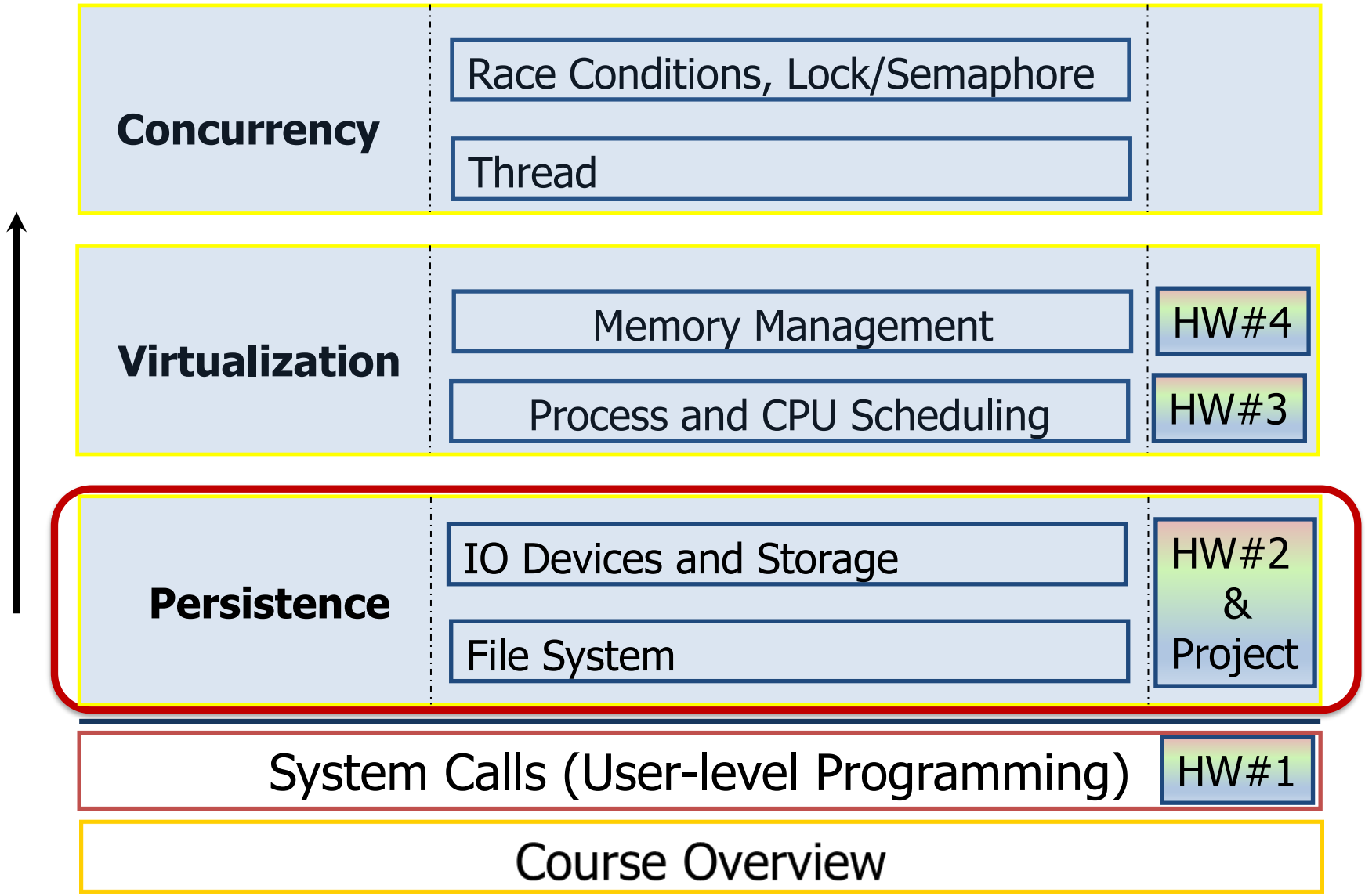
Lecture 5: File System Implementation

The Course Organization (Bottom-up)



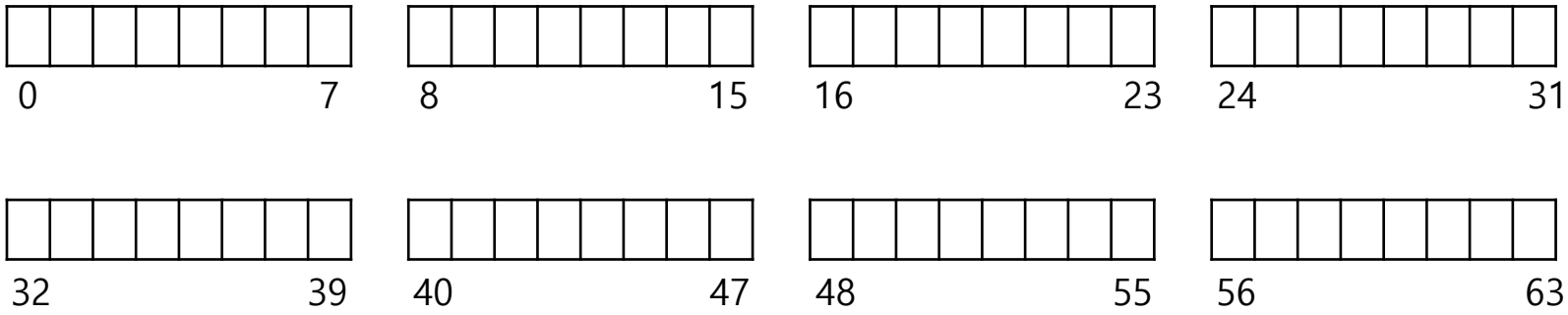
The Way To Think

- There are two different aspects to implement file system
  - **Data structures**
    - What types of on-disk structures are utilized by the file system to organize its data and metadata?
  - **Access methods**
    - How does it map the calls made by a process as `open()`, `read()`, `write()`, etc.
    - Which structures are read during the execution of a particular system call?

Overall Organization

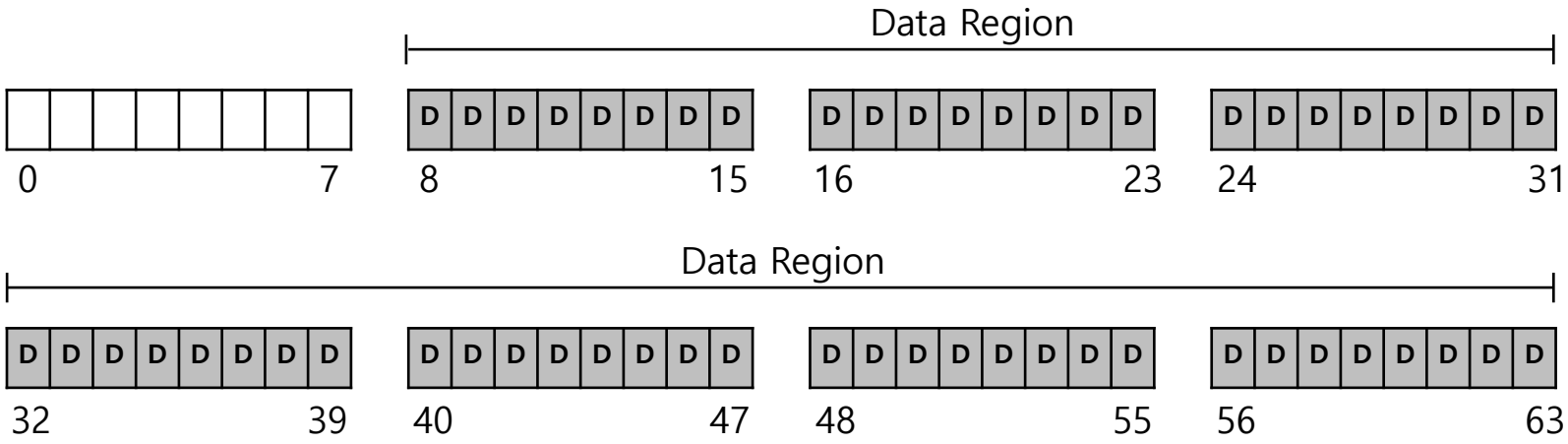
- Let's develop the overall organization of the file system data structure.

- Divide the disk into **blocks**.
  - Block size is 4 KB.
  - The blocks are addressed from *0 to N -1*.



Data region in file system

- Reserve **data region** to store user data

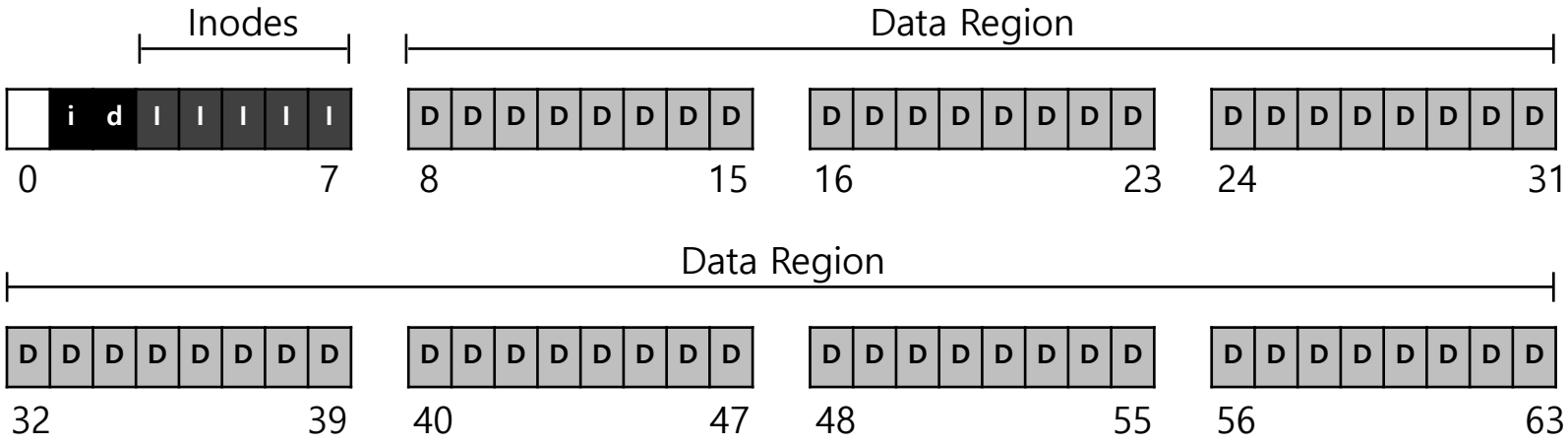


  - File system has to track which data block comprise a file, the size of the file, its owner, etc.

How we store these inodes in file system?

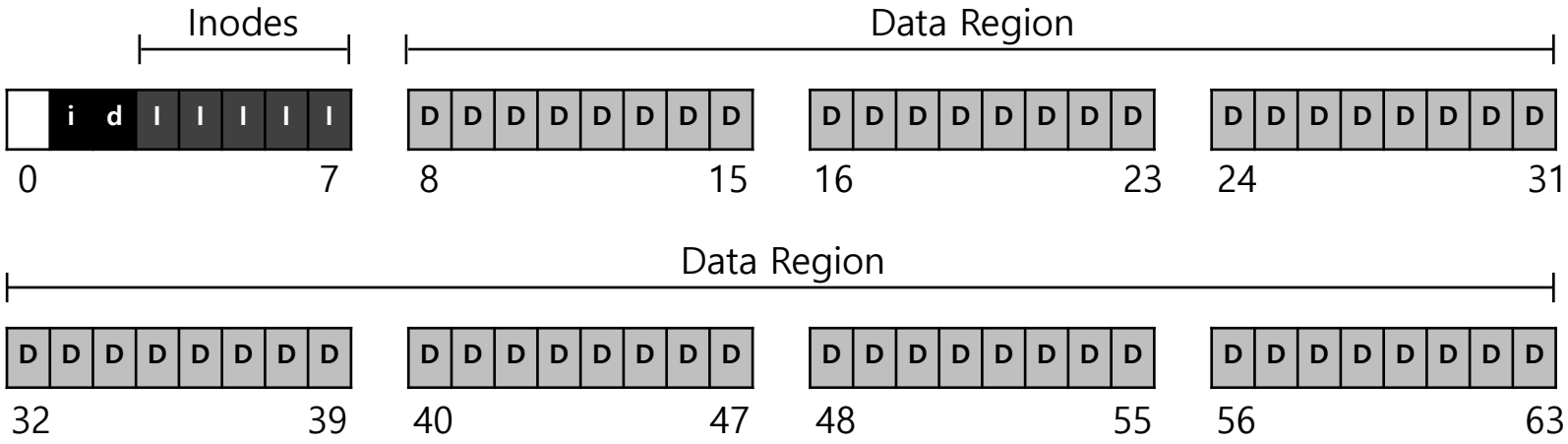
Inode table in file system

- 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)



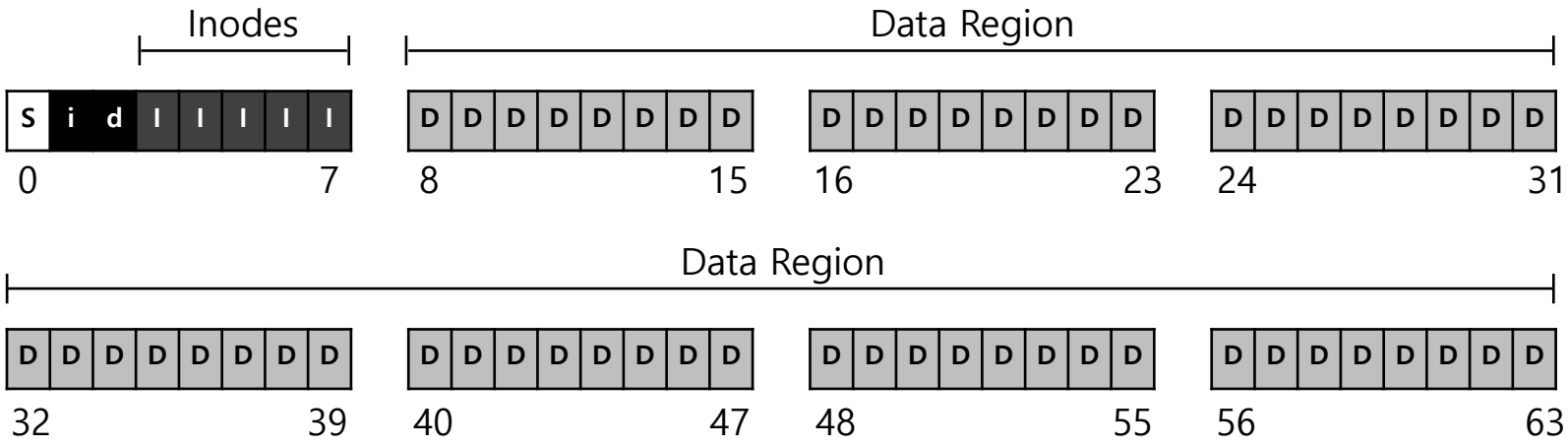
Allocation structures

- 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 for data region
  - **inode bitmap**: for inode table



Superblock

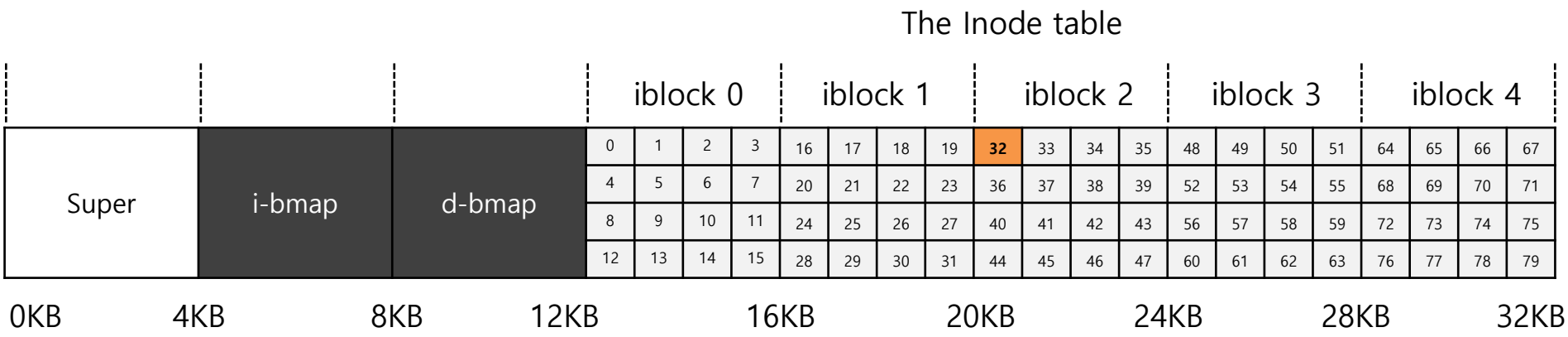
- Super block contains this **information** for **particular file system**
  - Ex) The number of inodes, begin location of inode table. etc



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

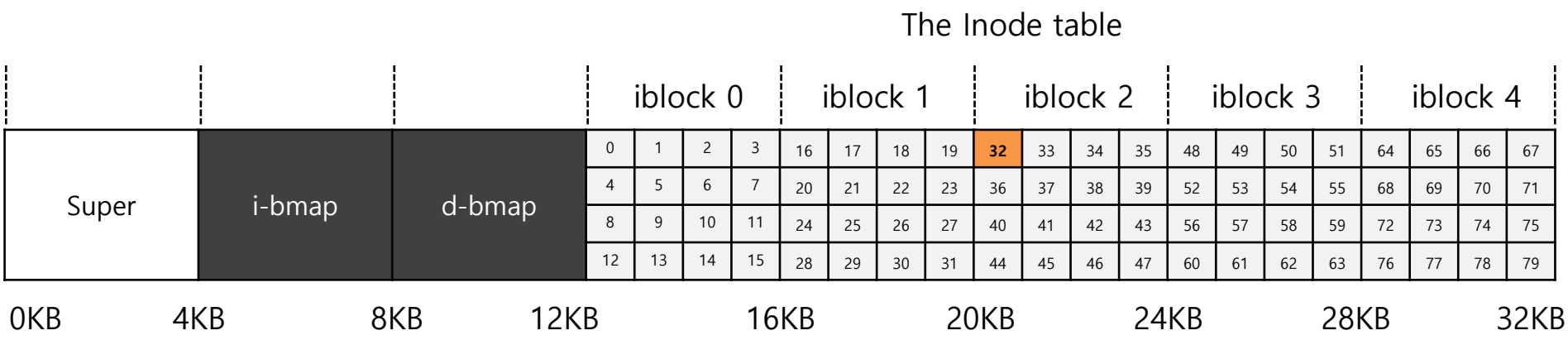
File Organization: The inode

- 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) (256 bytes) = 8192
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



File Organization: The inode (Cont.)

- Disks are not byte addressable, but sector addressable.
- Disk consist of a large number of addressable sectors, (512 bytes)
  - Ex) Fetch the block of inode (inode number: 32)
    - Sector address `iaddr` of the inode block:
      - `blk : (inumber * sizeof(inode)) / blocksize`
      - `sector : ((blk * blocksize) + inodeStartAddr ) /sectorsize`



File Organization: The inode (Cont.)

- `inode` have all of the information about a file
  - File type (regular file, directory, etc.),
  - Size, the number of blocks allocated to it.
  - Protection information(who ones the file, who can access, etc).
  - Time information.
  - Etc.

File Organization: The inode (Cont.)

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

The EXT2 Inode

The Multi-Level Index

- To support bigger files, we use multi-level index.
- **Indirect pointer** points to a block that contains more pointers.
  - inode have fixed number of direct pointers (12) and a single indirect pointer.
  - If a file grows large enough, an indirect block is allocated, inode's slot for an indirect pointer is set to point to it.
    - $(12 + 1024)$ x 4 K or 4144 KB

The Multi-Level Index (Cont.)

- Double indirect pointer points to a block that contains indirect blocks.
  - Allow file to grow with an additional 1024 x 1024 or 1 million 4KB blocks.
- Triple indirect pointer points to a block that contains double indirect blocks.
- Multi-Level Index approach to pointing to file blocks.
  - Ex) twelve direct pointers, a single and a double indirect block.
    - over 4GB in size $(12+1024+1024^2)$ x 4KB
- Many file system use a multi-level index.
  - Linux EXT2, EXT3, NetApp's WAFL, Unix file system.
  - Linux EXT4 use extents instead of simple pointers.

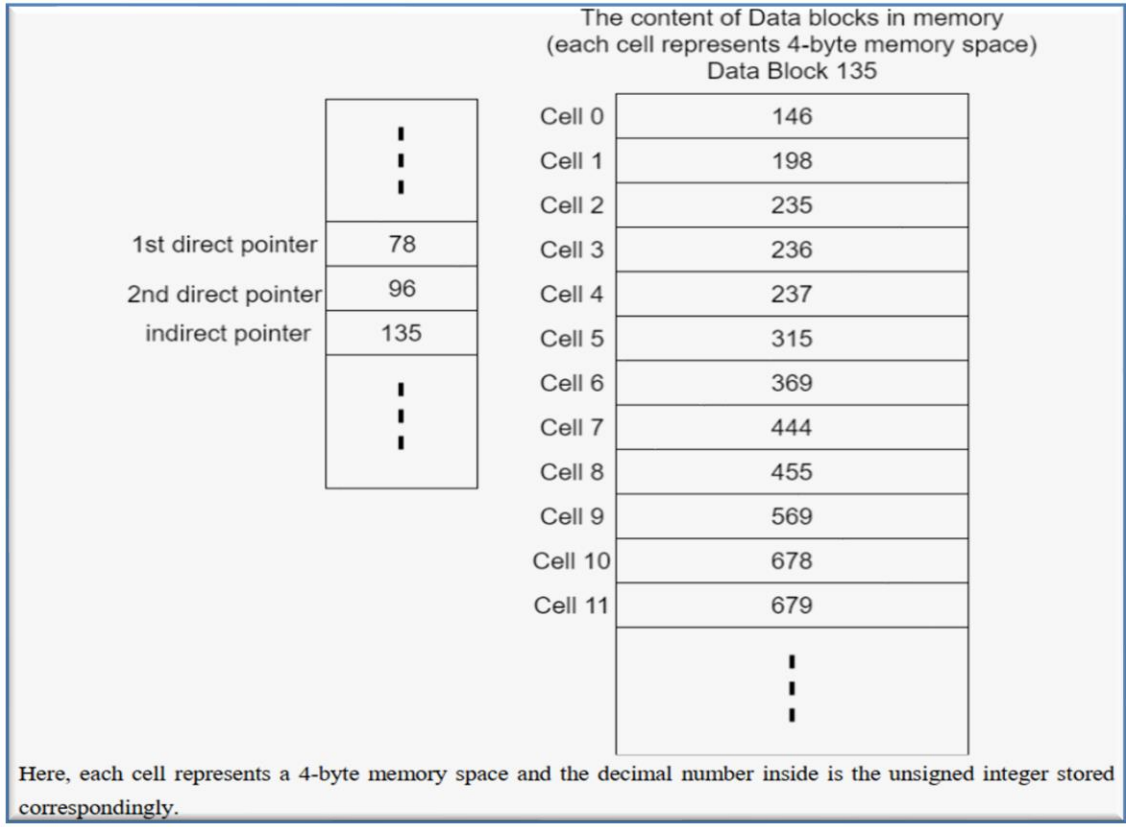
The Multi-Level Index (Cont.)

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file system remain -50% full
Directories are typically small	Many have few entries; most have 20 or fewer

File System Measurement Summary

Example

```
struct inode              /* The structure of inode, each file has only one inode */
 {
  int   i_number;      /* The inode number */
  time_t   i_mtime;      /* Creation time of inode*/
  int   i_type;      /* Regular file for 0, directory file for 1 */
  int   i_size;          /* The size of file */
  int   i_blocks;        /* The total numbers of data blocks   */
  int   direct_blk[2];   /*Two direct data block pointers   */
  int   indirect_blk;    /*One indirect data block pointer */
   int   file_num;       /* The number of file in directory, it is 0 if it is file*/
};
```



Here, each cell represents a 4-byte memory space and the decimal number inside is the unsigned integer stored correspondingly.

Answer the following questions:

(a) What is the biggest size we can have for a file with SFS?

(b) Provide data block numbers in sequence that will be read from the disk (only data blocks that contain file data) when read_t (inum, offset, buf1, count) is called in a user program, where inum is the corresponding inode number for the above inode, and buf1 is a pointer that points to a user-defined buffer.

	read_t (inum, offset, buf1, count)	The data block numbers in sequence that will be read from (only list the data blocks that contain file data)
Example 1	read_t(inum, 268, buf1, 400);	78
Example 2	read_t(inum, 268, buf1, 6000);	78, 96
(i)	read_t(inum, 7000, buf1, 9000);	
(ii)	read_t(inum, 15000, buf1, 25000);	
(iii)	read_t(inum, 20000, buf1, 36000);	
(iv)	read_t(inum, 1000, buf1, 21000);	

Solution

Given **offset** in read_t(), we can use the following function to determine which data block will be read (assume that offset is inside the range):

Let a = [**offset** / 4096] and b = **offset** mod 4096.

$$\text{start} = \begin{cases} direct\_blk[a] & a < 2 \\ Cell[a-2] where\ Cell\ is\ an\ integer\ pointer\ pointing\ to\ indirect\ block & a \geq 2 \end{cases}$$

***start* indicates at which data block that read_t() starts**. If a < 2, which means it starts at **direct_blk**. We can simply get the start data block number by **direct_block[a].** Otherwise, **indirect_blk** will be used. In this case (a >= 2), we can first read **indirect_blk** (that is the data block number for a 4KB data block) into the memory and initialize an integer pointer called **Cell** to point it. Because each data block number is represented by an integer (4 bytes), then we can get the data block number by **Cell[a - 2]**. After we obtain the data block number, b is the offset to start in this block.

To determine how many data blocks will be read, we can use the following functions:

Let a' = [(offset + count - 1) / 4096] and b' = (offset + count - 1) mod 4096

$$\text{end} = \begin{cases} direct\_blk[a'] & a < 2 \\ Cell[a'-2] where\ Cell\ is\ an\ integer\ poniter\ pointing\ to\ indirect\ block & a \geq 2 \end{cases}$$

Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files ."dot" for current directory and .."dot-dot" for parent directory
  - For example, `dir` has three files (`foo, bar, foobar_is_a_pretty_longname`)

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

on-disk for dir

Example

An example structure can be found below:

```
struct dir_mapping /* Record file information in directory file */
{
    char dir[20]; /* The file name in current directory */
    int inode_number; /* The corresponding inode number */
};
```

Question: What is the content of Block 0?

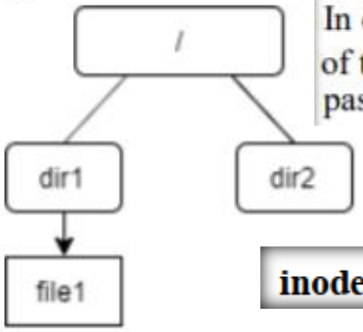
Answer:

.	0
..	0
dir1	1
dir2	2

Each directory file should at least contain two mapping items, namely "." and "..", for itself and its parent directory, respectively (the parent of the root directory is itself).

When we look for a specific file under a directory, we traverse ***dir_mapping*** one by one, comparing the file name with ***dir[]***. Once ***dir[]*** equals the file name, the corresponding ***inode_number*** will be returned.

For example, there is a simple directory below:



Question: Suppose a user provides the following absolute path: **/dir1/file1** In order to obtain the inode number of file1, the sequence of the inode numbers and data block numbers we need to pass (starting from the root directory)

inode 0 -> data block 0 -> inode 1 -> data block 1 -> inode 3

"/" is the root directory. "dir1" and "dir2" are two directory files. "file1" is a regular file.

Assume the inode numbers of "/", "dir1", "dir2", "file1" are 0, 1, 2 and 3, respectively. Moreover, each directory file only occupies one data block (4 KB for one data block), and the data block numbers allocated to "/", "dir1" and "dir2" are 0, 1 and 2, respectively.

Answer:

Free Space Management

- File system track which inode and data block are free or not.
- In order to manage free space, we have two simple bitmaps.
  - When file is newly created, it allocated inode by searching the inode bitmap and update on-disk bitmap.
  - Pre-allocation policy is commonly used for allocate contiguous blocks.

Access Paths: Reading a File From Disk

- Issue an `open("/foo/bar", O_RDONLY)`,
  - Traverse the pathname and thus locate the desired indoe.
  - 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 finale 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 write	read	read write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

File Creation Timeline (Time Increasing Downward)

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os.
  - For example, long pathname(/1/2/3/…./100/file.txt)
    - One to read the inode of the directory and at least one read its data.
    - Literally perform hundreds of reads just to open the file.
- In order to reduce I/O traffic, file systems aggressively use system memory(DRAM) to cache.
  - Early file system use fixed-size cache to hold popular blocks.
    - Static partitioning of memory can be wasteful;
  - Modem systems use dynamic partitioning approach, unified page cache.
- Read I/O can be avoided by large cache.

Caching and Buffering (Cont.)

- Write traffic has to go to disk for persistent. Thus, cache does not reduce write I/Os.
- File system use write buffering for write performance benefits.
  - delaying writes (file system batch some updates into a smaller set of I/Os).
  - By buffering a number of writes in memory, the file system can then schedule the subsequent I/Os.
  - By avoiding writes
- Some application force flush data to disk by calling `fsync()` or direct I/O.

Summary

- File System
  - Key data structures
    - Superblock
    - Inode
    - Directory
  - Access interface: Open/Read/Write
- Caching and Buffering
- Next: I/O Devices