CS 111 week 6
Project 3A: Analyze Ext2 FS

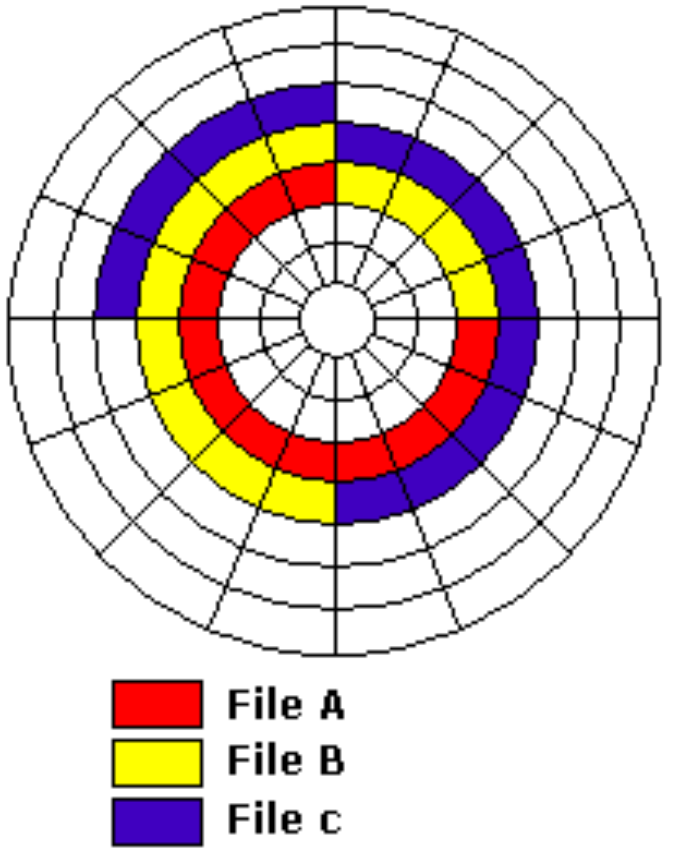
Tianxiang Li

Task Overview

- Implement a program
  - to read the on-disk representation of a file system
  - analyze it, and summarize its contents.
- In the next project
  - we will write a program to analyze this summary for evidence of corruption.

Block in a disk

- File system groups **multiple contiguous bytes** into a **single unit** called **block**
  - Similar to OS/HW group bytes in memory to page
  - Reason: Simplify management & reduce overhead
- Example: Identify which part of the disk/memory is free (Not used by processes/files).
  - Naive approach: one free/use bit for every byte → Overhead: 12.5%
  - Assume page/block size is 4096byte → Overhead: ~0%
  - Drawback: Internal fragmentation
- The **size of the block** is always 2^n. Commonly used sizes are (512B, 1024B, 2048B, 4096B)

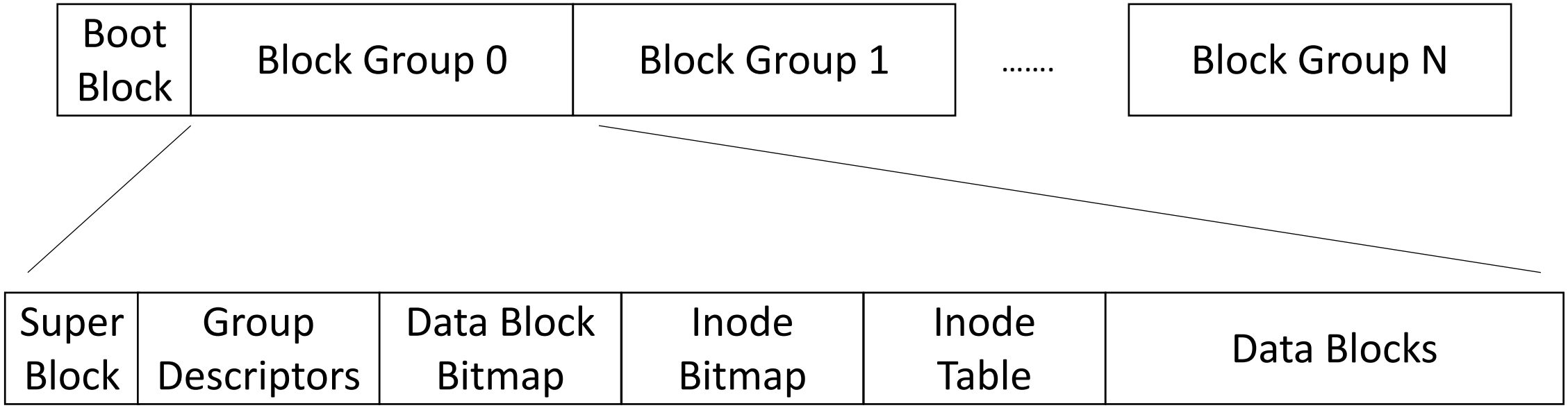


Proj3A Overview

- Implement a program
  - to read the on-disk representation of a file system
    - Input: A file(e.g. trivial.img) containing data of disk
  - analyze it, and summarize its contents
    - E.g. max number of files (i.e. inodes) allowed in the FS

Ext2 file system overview

Structure of an Ext2 file system



- The first 1024 bytes: boot block, the rest of the disk is partitioned into multiple block groups.
- Boot block: Contains the data/code used for booting up the operating system.
  - BIOS needs to load OS from disk to memory → Boot block contains the location of the operating system on the disk
- Block groups: contiguous bytes on adjacent location of the disk, a single file can only on one block group → Speed up the performance, while increase fragmentation

Terminology

- Blocks
  - A **partition**, **disk**, **file** or **block device** formatted with a Second Extended Filesystem is divided into small groups of **sectors** called “blocks”.
  - These blocks are then grouped into larger units called block groups.
- Block group
  - Blocks are clustered into block groups in order to reduce fragmentation and minimize the amount of head seeking when reading a large amount of consecutive data.
  - **Information about each block group** is kept in a **descriptor table** stored in the block(s) immediately after the superblock.
  - **Two blocks** near the **start** of each **group** are reserved for the **block usage bitmap** and the **inode usage bitmap** which show which blocks and inodes are in use.
  - The block(s) following the bitmaps in each block group are designated as the **inode table** for that block group and the remainder are the **data blocks**. The block allocation algorithm attempts to allocate data blocks in the same block group as the inode which contains them.

Terminology

- **Directories**
  - A directory is a **filesystem object** and has an inode just like a file. It is **a specially formatted file** containing **records** which associate each name with an inode number.
  - The inode allocation code should try to assign inodes which are in the same block group as the directory in which they are first created.
- **Inodes**
  - The (**index node**). Each object in the filesystem is represented by an inode.
  - The inode structure contains **pointers** to the filesystem **blocks** which contain the data held in the object and all of the metadata about an object except its name.
  - The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time, modification time, deletion time, number of links, fragments, version (for NFS) and extended attributes (EAs) and/or Access Control Lists (ACLs).
-

Super block

- Located after the boot block:
  - Starting at 1024 bytes offset from the beginning of the disk

- Size: 1024 bytes

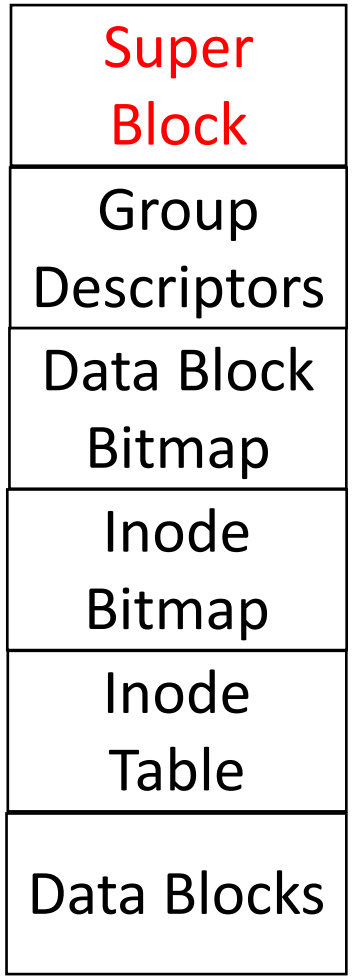
- Describe the **general information** of the file system
  - e.g. what is the file system on the disk, ext2 or FAT
  - how many blocks are there in the file system
  - What is the block size of the file system

Super structure

```
struct ext2_super_block {
        __u32  s_inodes_count;          /* Inodes count */
        __u32  s_blocks_count;          /* Blocks count */
        ...
        __u32  s_free_blocks_count;     /* Free blocks count */
        __u32  s_free_inodes_count;     /* Free inodes count */
        __u32  s_first_data_block;      /* First Data Block */
        __u32  s_log_block_size;        /* Block size */
        ...
        __u32  s_blocks_per_group;      /*  Blocks per group */
        ...
        __u16  s_magic;                 /* Magic signature */
        ...
}
```

s_blocks_count,
s_blocks_per_group
→ Number of block groups

s_magic: shows the file system on the disk



Parse and output data in superblock: naive approach

```
int fd = 0;
unsigned int inodes_count = 0, blocks_count = 0, log_block_size = 0;
fd = open("test.img", O_RDONLY);
//pread: Same as read, except adding a fourth argument to specify the offset of the
//file descriptor as a starting point to read

//Note: Need to check the return value of pread to make sure it reads the
//specified size.
pread(fd, &inodes_count, sizeof(inodes_count), 1024);
pread(fd, &blocks_count, sizeof(blocks_count), 1028);
…
pread(fd, &log_block_size, sizeof(log_block_size), 1048);
block_size = 1024 << log_block_size;  /* calculate block size in bytes */
```

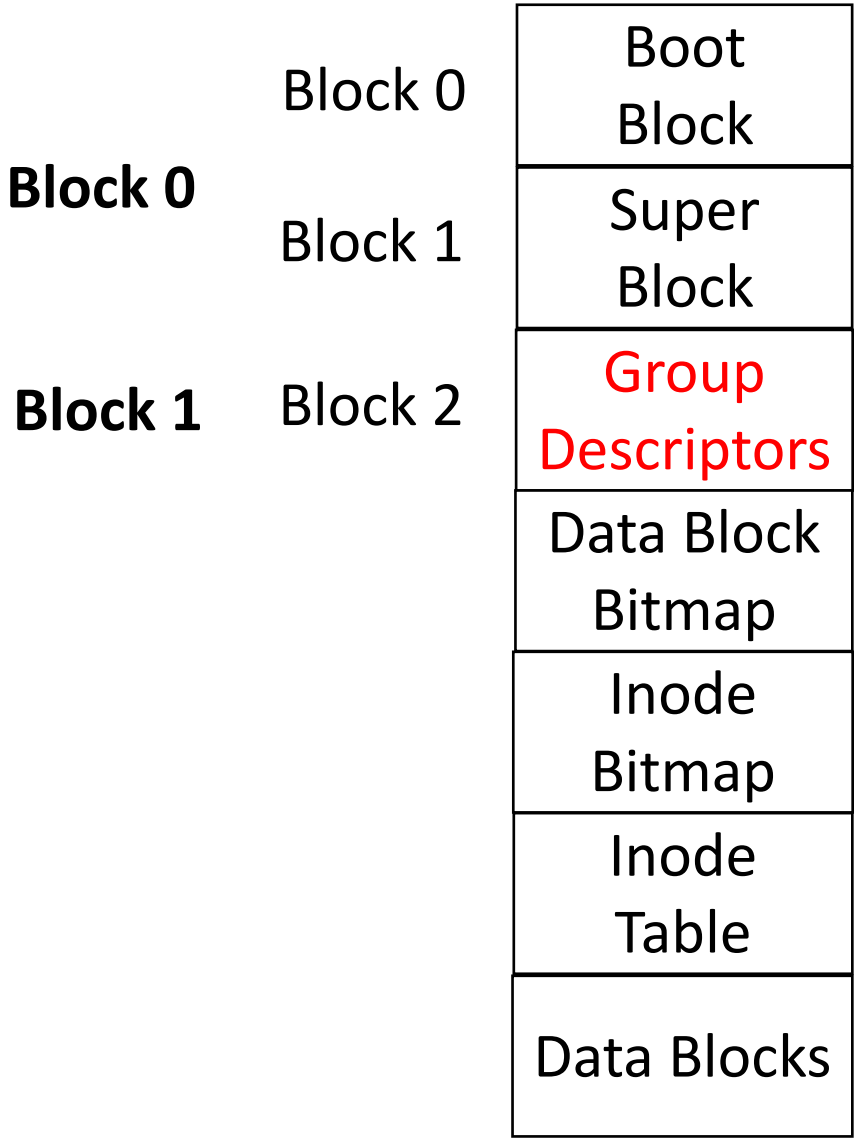
Parse and output data in superblock: recommended approach

```
int fd = 0;
unsigned int inodes_count = 0, blocks_count = 0, log_block_size = 0;
struct ext2_super_block super;
fd = open("test.img", O_RDONLY);

//Note: Need to check the return value of pread to make sure it reads the
//specified size.
pread(fd, &super, sizeof(super), 1024);

inodes_count = super.s_inodes_count;
blocks_count = super.s_blocks_count;
…
block_size = 1024 << super.s_log_block_size;  /* calculate block size in bytes */
```

Block Group descriptor

- Located in the *next block after super block*
- Three conditions:
  - block size = 1024, start at block 2
  - **block size > 1024 (at least 2048), start block 1**
  - block size < 1024, not possible, minimum block size: 1024

- An array of block group descriptors:
  - each representing a block group on the disk
  - records the general information of the block
    - for example: number of free blocks in the block group, the location of block bitmap, inode bitmap, inode table.

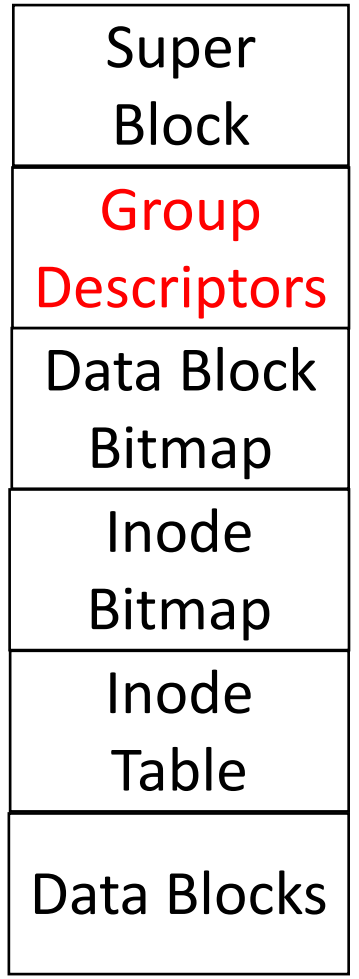


Block 0

Block 1

Block Group descriptor

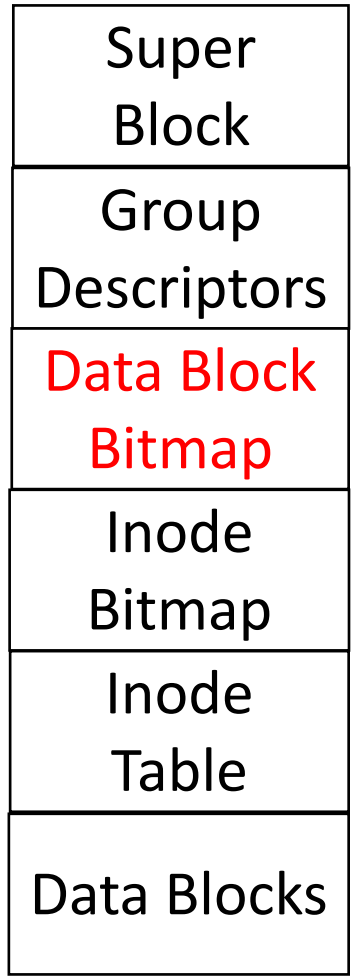
```
struct ext2_group_desc
{
        //marks the starting location of important blocks
        __u32   bg_block_bitmap;        /* Block bitmap block */
        __u32   bg_inode_bitmap;        /* Inodes bitmap block */
        __u32   bg_inode_table;         /* Inodes table block */

        __u16   bg_free_blocks_count;   /* Free blocks count */
        __u16   bg_free_inodes_count;   /* Free inodes count */
        __u16   bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;
        __u32   bg_reserved[3];
};
```



Block bitmap

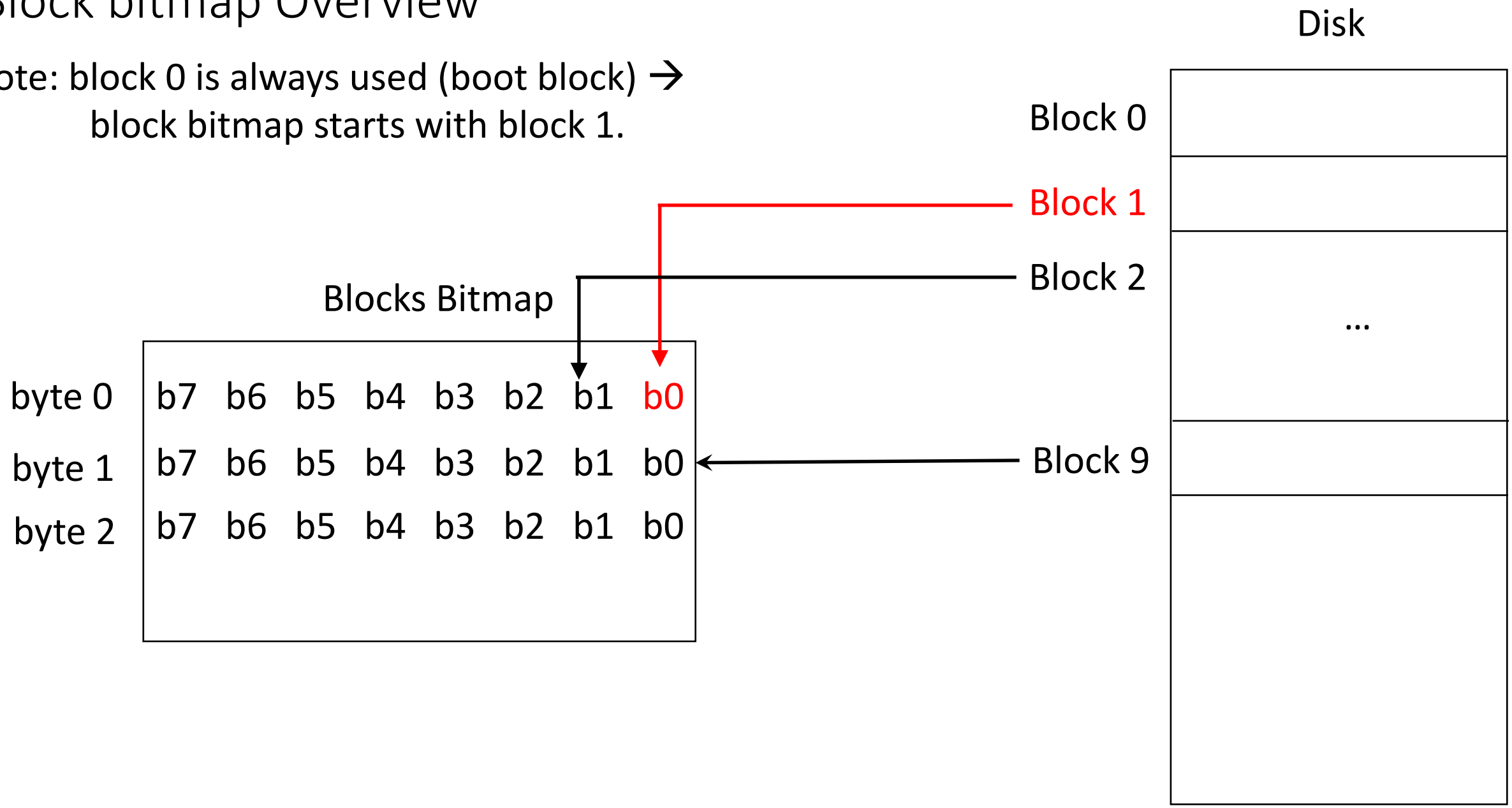
- A sequence of 0 & 1 bit Indicating whether a block has been used or not.

- 1 indicates the block is used (occupied by files or used by file system),
  0 indicates the block is free (can be used by newly created/enlarged files)



Block bitmap Overview

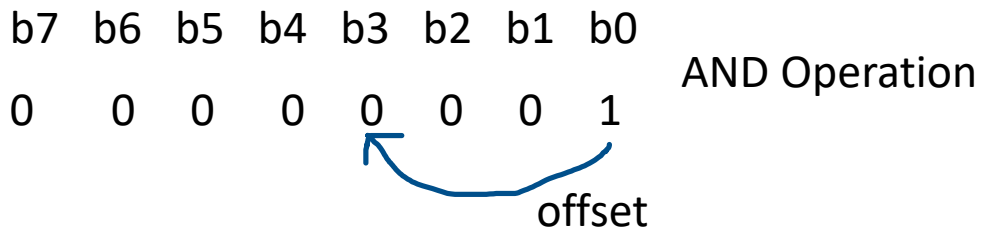
Note: block 0 is always used (boot block) → block bitmap starts with block 1.



Sample code for block bitmap

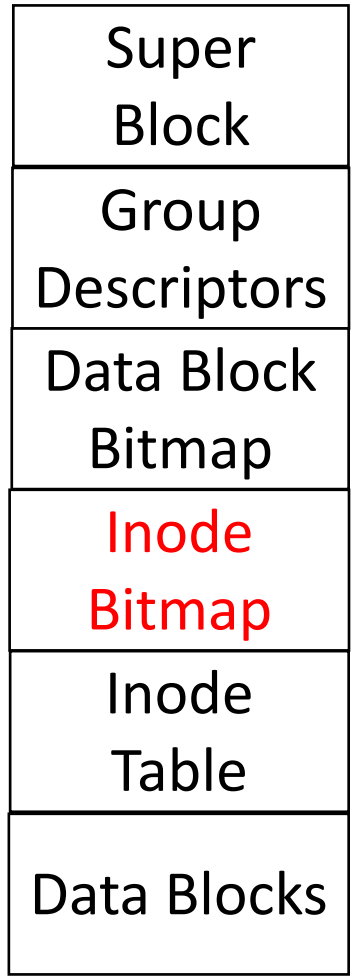
- Note: not tested. just to show the basic principle.

```
int is_block_used(int bno, char * bitmap)
{
        int index = 0, offset = 0;
        if (bno == 0)
                return 1;

        index = (bno – 1)/8;  //which byte within the bitmap stores the info of this block#
        offset = (bno – 1)%8;
        return ((bitmap[index] & (1 << offset)) )
}
```



inode bitmap:

- Indicate whether an ext2_inode in inode table is used or not.
- Inode 0 is reserved, inode bitmap starts at inode 1.
- Exactly the same as block bitmap.

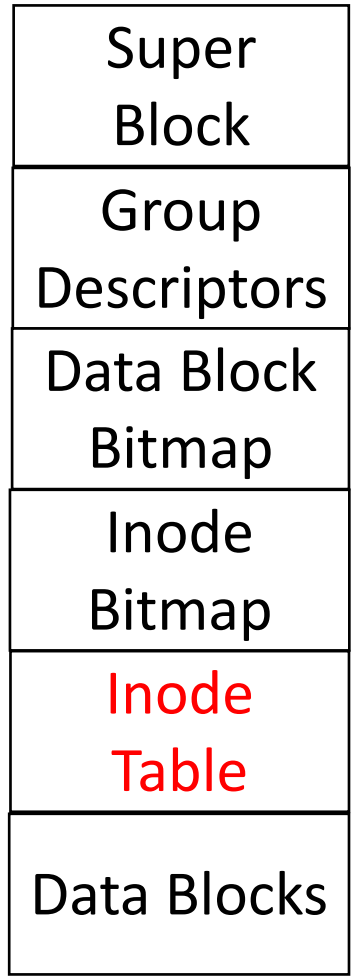


inode table

- An array of inode descriptor.
- Each inode describes the metadata of a file.

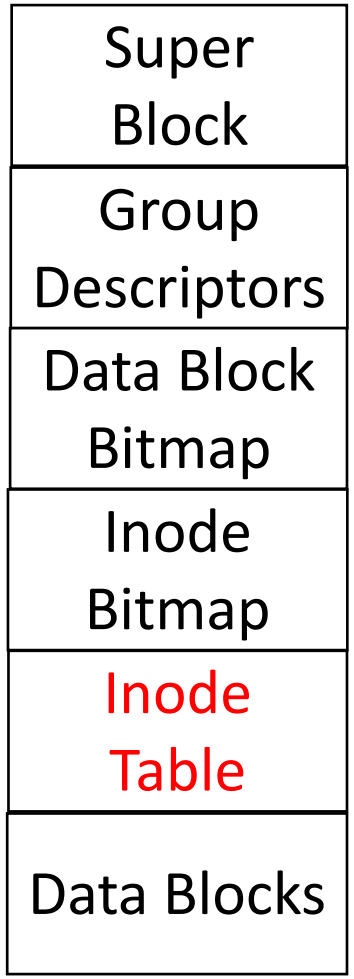
```
struct ext2_inode {
        __u16  i_mode;         /* File mode */
        __u16  i_uid;          /* Owner Uid */
        __u32  i_size;         /* Size in bytes */
        __u32  i_atime;        /* Access time */
        __u32  i_ctime;        /* Creation time */
        __u32  i_mtime;        /* Modification time */
        __u32  i_dtime;        /* Deletion Time */
        __u16  i_gid;          /* Group Id */
        __u16  i_links_count;  /* Links count */
        …
        __u32   i_block[EXT2_N_BLOCKS];  /* Pointers to data blocks of file */
}
```

Note: inode starts with 1 → ext2 assigns inode 0 to special files (e.g. /dev/null) that do not require a backup inode



i_mode: Type of the file and access permission

- i_mode in ext2_inode stores the type of the file (regular file, directory, symbolic link) and the accession permissions (rwx for ugo)
- Get the type of the file: Use macro defined in <sys/stats.h>, returns 0 if false, non zero if true.
  - S_ISDIR(i_mode): Test for a directory,
  - S_ISREG(i_mode): Test for a regular file.
  - S_ISLNK(i_mode): Test for a symbolic link.



Convert access time/modification/creation time to GMT

- i_atime, i_ctime and i_mtime stores the number of elapsed seconds since epoch (00:00:00 UTC on 1 January 1970) → i_atime = 1573239972
- Spec requires us to translate these times into human readable time in GMT

- Use *gmtime* for translation

How data is stored

i_block stores all the data belongs to the file

```
#define EXT2_NDIR_BLOCKS          12
#define EXT2_IND_BLOCK            EXT2_NDIR_BLOCKS          //12
#define EXT2_DIND_BLOCK           (EXT2_IND_BLOCK + 1)      //13
#define EXT2_TIND_BLOCK           (EXT2_DIND_BLOCK + 1)     //14
#define EXT2_N_BLOCKS             (EXT2_TIND_BLOCK + 1)     //15
```

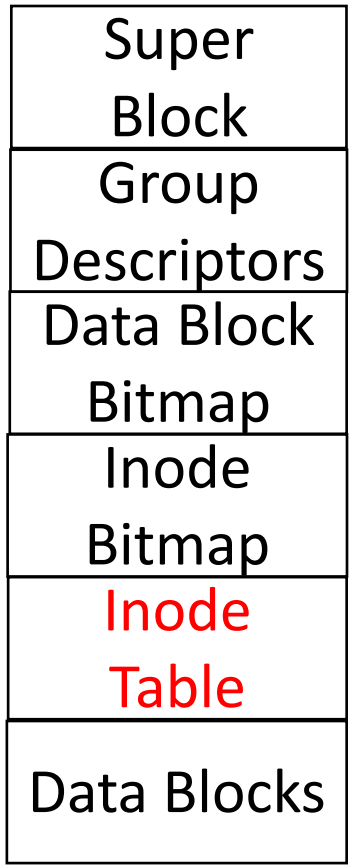
```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        …
        __u32   i_block[EXT2_N_BLOCKS];  /* Pointers to blocks */
}
```

i_block[0..11] point directly to the first 12 data blocks of the file.

i_block[12] points to a single indirect block

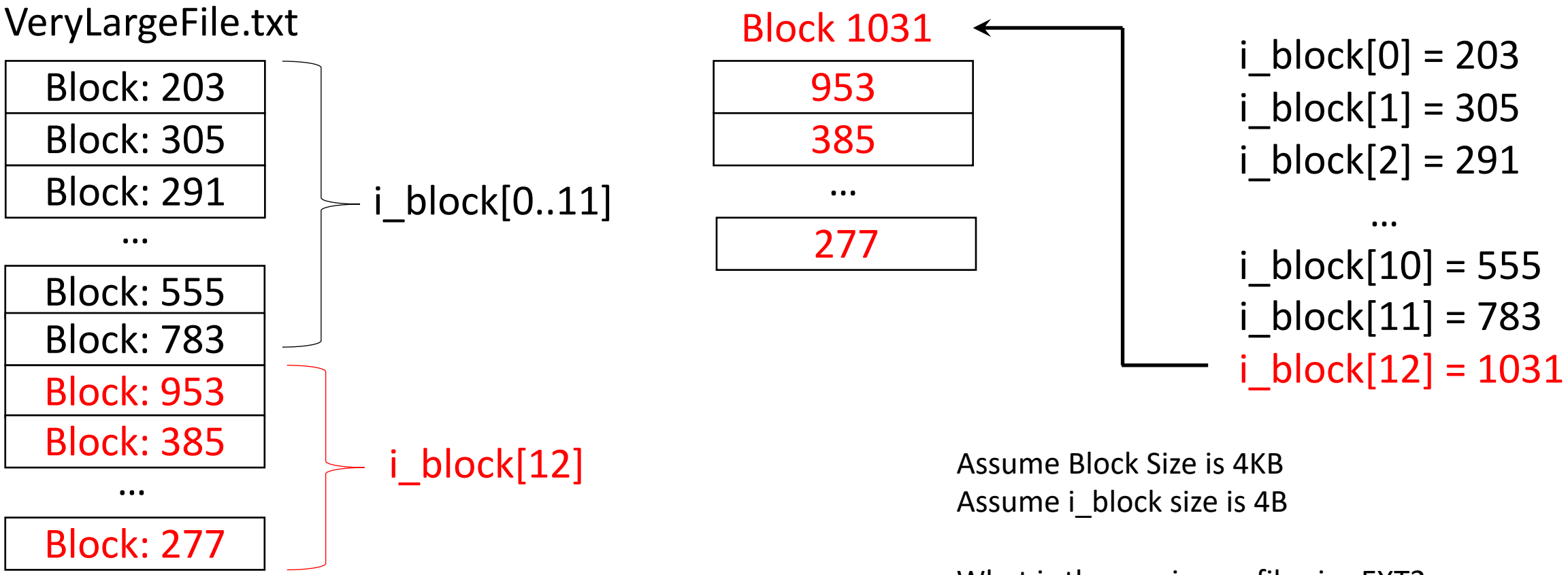
i_block[13] points to a double indirect block

i_block[14] points to a triple indirect block



Only 15 i_block → At most store 15 block number?

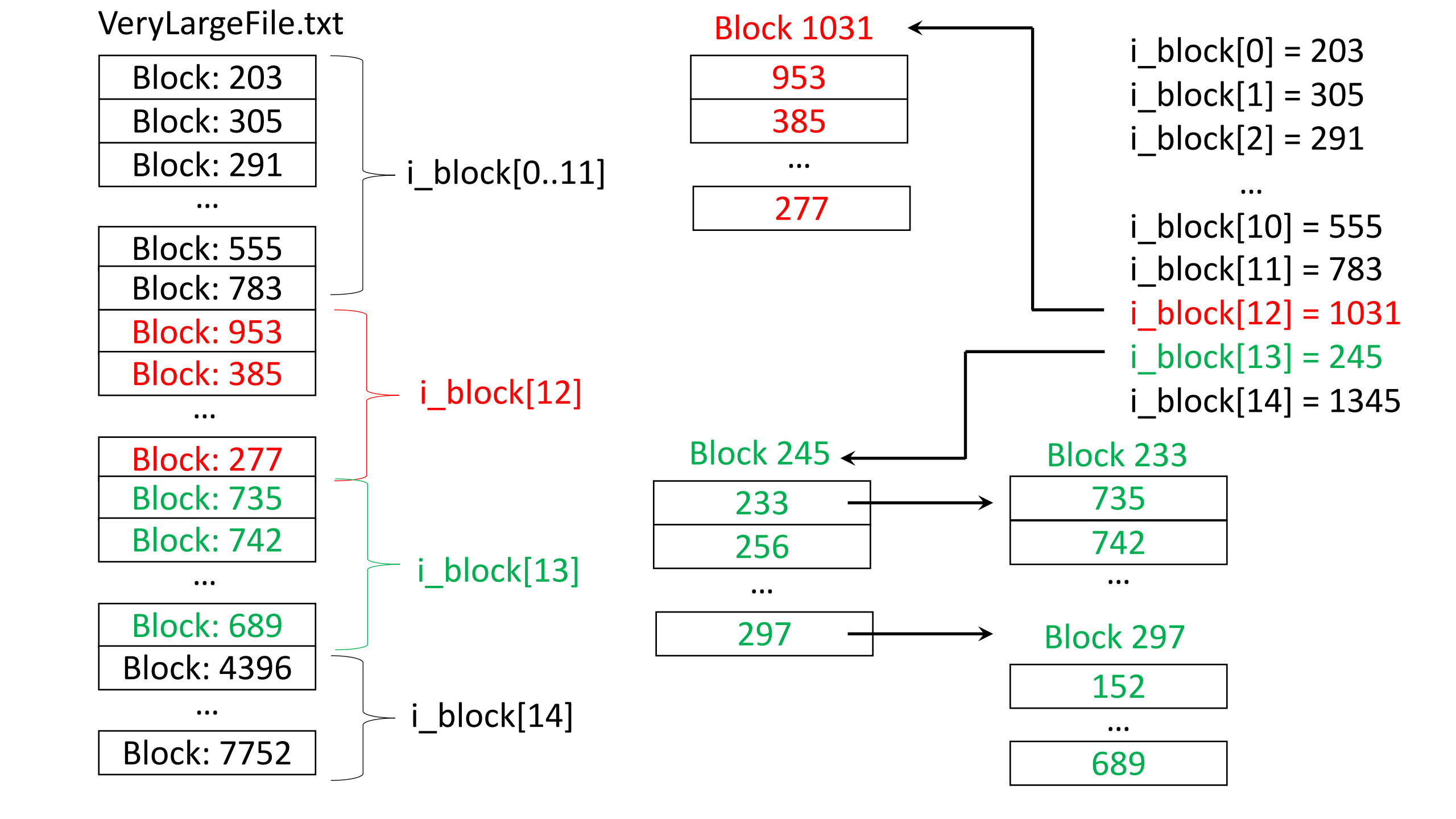
How does ext2 store large files (Several GB or Even TB?)



Assume Block Size is 4KB
Assume i_block size is 4B

What is the maximum file size EXT2 can support if we use i_block[0] to i_block[12]?

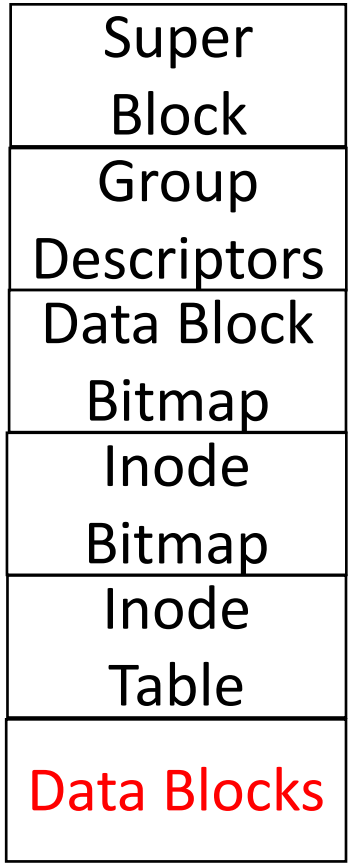
Key: how many i_blocks are there?
i_block [0 to 11] → 12 i_blocks
i_block[12] points to a block which is 4KB
→ Which can store 4KB/4B = 1024 i_blocks
→ Total i_block entries = 1024 + 12 = 1036
→ Total file size = 1036 x 4KB ~ 4MB


VeryLargeFile.txt
Block: 203
Block: 305
Block: 291
...
Block: 555
Block: 783
Block: 953
Block: 385
...
Block: 277
Block: 735
Block: 742
...
Block: 689
Block: 4396
...
Block: 7752
i_block[0..11]
i_block[12]
i_block[13]
i_block[14]
Block 1031
953
385
...
277
i_block[0] = 203
i_block[1] = 305
i_block[2] = 291
...
i_block[10] = 555
i_block[11] = 783
i_block[12] = 1031
i_block[13] = 245
i_block[14] = 1345
Block 245
233
256
...
297
Block 233
735
742
...
Block 297
152
...
689

Special file type: directory

- Directory need to record information on what are the files under it → stored in the data block of the directory
- Challenge: Need to record the name of the file → each entry may take variable size
- Each file under the directory is stored in data structure below.

```
struct ext2_dir_entry {
        __u32  inode;                   /* Inode number of the file */
        __u16  rec_len;                 /* Directory entry length */
        __u8   name_len;                /* name length*/
        __u8   file_type;               /* file type of the file*/
        char   name[EXT2_NAME_LEN];     /* File name */
};
```

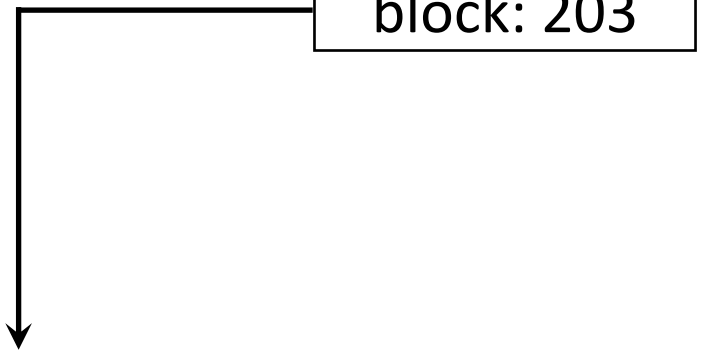


$ls /home/user0/
music/
test.txt
bin/

/home/user0

block: 203

i_ node = 13
i_block[0] = 203



block 203

	inode	rec_len	name_len	file_type	name								
offset: 0	13	12	1	2	.	\0	\0	\0					file: .
offset: 12	10	12	2	2	.	.	\0	\0					file: ..
offset: 24	18	16	5	2	m	u	s	i	c	\0	\0	\0	file: music
offset: 40	15	16	8	1	t	e	s	t	.	t	x	t	file: test.txt
offset: 56	19	12	3	2	b	i	n	\0					file: bin

Sample code to read file names and inode number from directory

Note: Below code only assumes there is only one block in the directory

```
entry = (struct ext2_dir_entry *) inode->i_block[0];
iter = 0;
while(iter < inode->i_size)  { //size is less than the total size of the inode
        char file_name[EXT2_NAME_LEN+1];
        memcpy(file_name, entry->name, entry->name_len);
        file_name[entry->name_len] = 0;              /* append null char to the file name */
        printf("%u %s\n", entry->inode, file_name); /* print inode number and file name */
        entry = (void*) entry + entry->rec_len;      /* move to the next entry */
        iter += entry->rec_len;
}
```

What happens when access a file at ext2?

cat /home/file.txt

root directory → reserved inode 2 → inode_table[1] → inode_table[1].i_block
→ ext2_entry → the inode of /home: 15

inode_table[14] → inode_table[14].i_block → ext2_entry → the inode of /home/file.txt: 23

inode_table[22] → inode_table[22].i_block → read from disk to get the data of file.txt!

A very useful reference

http://cs.smith.edu/~nhowe/Teaching/csc262/oldlabs/ext2.html