CS111 Spring 2024

Discussion 1C F (week7)

Yadi Cao

Logistics

- Lab4 will be due May/31
- This week is for the concepts only

Agenda

- Lab4 background
  - File systems overview
  - EXT2
- Lab4 starter (coding)
- Q&A

Lab4 Background: File System

Lab4 Overview

Implement a program

- Initialize a file system img

Start the lab early!
Otherwise you would be struggling this time!

Very useful references for you

Textbook

- Chapter 39: Interlude: File and Directories
- Chapter 40: File System Implementation

Lab demonstration by Jon in Spring 2021

- https://www.youtube.com/watch?v=YRpUVGZ2uB4

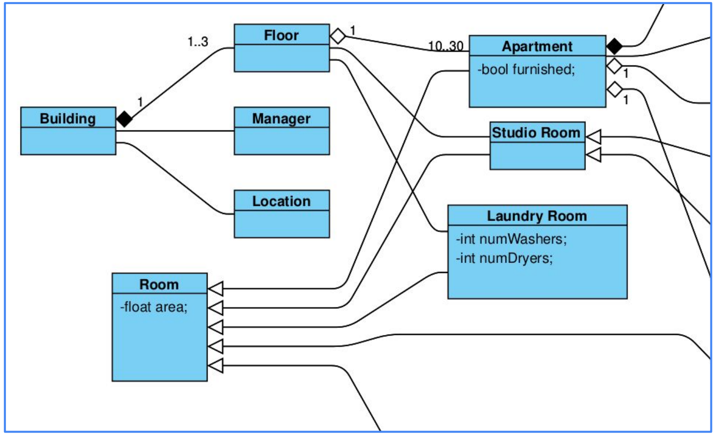
High level summary of EXT2 structure

- The Ext2 Filesystem: http://www.science.smith.edu/~nhowe/262/oldlabs/ext2.html

Detailed Doc on EXT2 Fields

- The Second Extended File System: http://www.nongnu.org/ext2-doc/ext2.html

Apartment manager is a file system (EXT2)



- There are many floors
- Each floor has many rooms
- Need a quick glance of each floor’s situation
- Need to record which apartments are occupied
- Also need to record who owns/uses which rooms

A “apartment manager” needs structural and logical rules used to manage the floors of rooms, their occupancy and ownership.

Lab4 Background: Formal descriptions

Formal description of file system

- A file system organizes, stores, retrieves data in a storage device.
- Without a file system, the data on a device would become a meaningless chunk of zero and ones.
- A file system allows us to separate the data into smaller pieces and give each of them a meaningful name, keep track of their location (starting and ending positions). It also helps us control the permissions (access control) of each piece of data.
- We call these pieces of data **files**.

A “file system” is the structure and logic rules used to manage the groups of data and their metadata.

Main tasks of a file system

- **Space management**
  - Allocate space (physical units) in a granular manner
  - Organize files and directories, and keep track of which areas of the media belong to which file and which are not being used

- **Filenames**
  - We need to identify the storage space location (i.e, we need to name our files to be able to distinguish them from one another)
  - The set of acceptable characters in file names, their lengths are controlled by the file system

Main tasks of a file system

- **Directories**
  - A file system typically makes it possible for a user to group a bunch of files together via the support for directories

- **Metadata**
  - We need to know the length of a file (how many blocks does it span?)
  - A file's creation time, modification time, access time
  - Owners (root, regular user, groups etc), access permissions (read-only, executable, etc)
  - Others

Analogy

- Many floors
- Many apartments per floor
- A quick glance of each floor
- Record occupancy
- Record ownership/meta

EXT2

- Many group blocks
- Many blocks per group
- Group descriptor
- Block bitmap
- Inode bitmap/tables

Both need structural and logical rules used to manage 1)Allocation and 2) Keep tracking(meta)

Files

A file is a collection of data

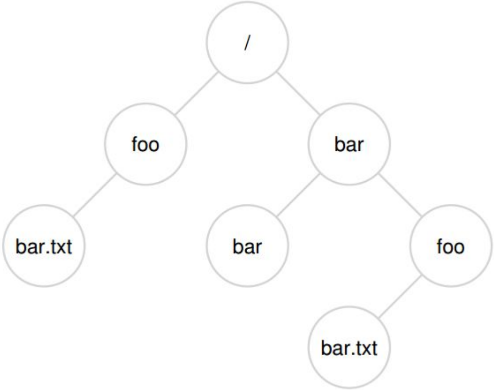
- A file is a collection of data that is stored on disk and that can be manipulated as a single unit by its name.
- A file is simply a linear array of bytes, each of which you can read or write.
- Each file has some kind of low-level name, referred to as its inode number.
- Each file has an **inode number** associated with it.

```
cs111@cs111 cs111/lab4 (main *%) » ls -ain
total 68
942569 drwxr-xr-x  2 1000 1000    4096 Nov 18 21:25 .
942013 drwxr-xr-x 15 1000 1000    4096 Nov 18 21:19 ..
942550 -rw-r--r--  1 1000 1000 1048576 Nov 18 21:25 cs111-base.img
942347 -rwxr-xr-x  1 1000 1000   16792 Nov 18 21:25 ext2-create
942572 -rw-r--r--  1 1000 1000   11020 Nov 18 21:29 ext2-create.c
942363 -rw-r--r--  1 1000 1000    7984 Nov 18 21:25 ext2-create.o
942570 -rw-r--r--  1 1000 1000     253 Oct  6 19:38 Makefile
942571 -rw-r--r--  1 1000 1000    1320 Oct  6 19:38 README.md
```

Directories

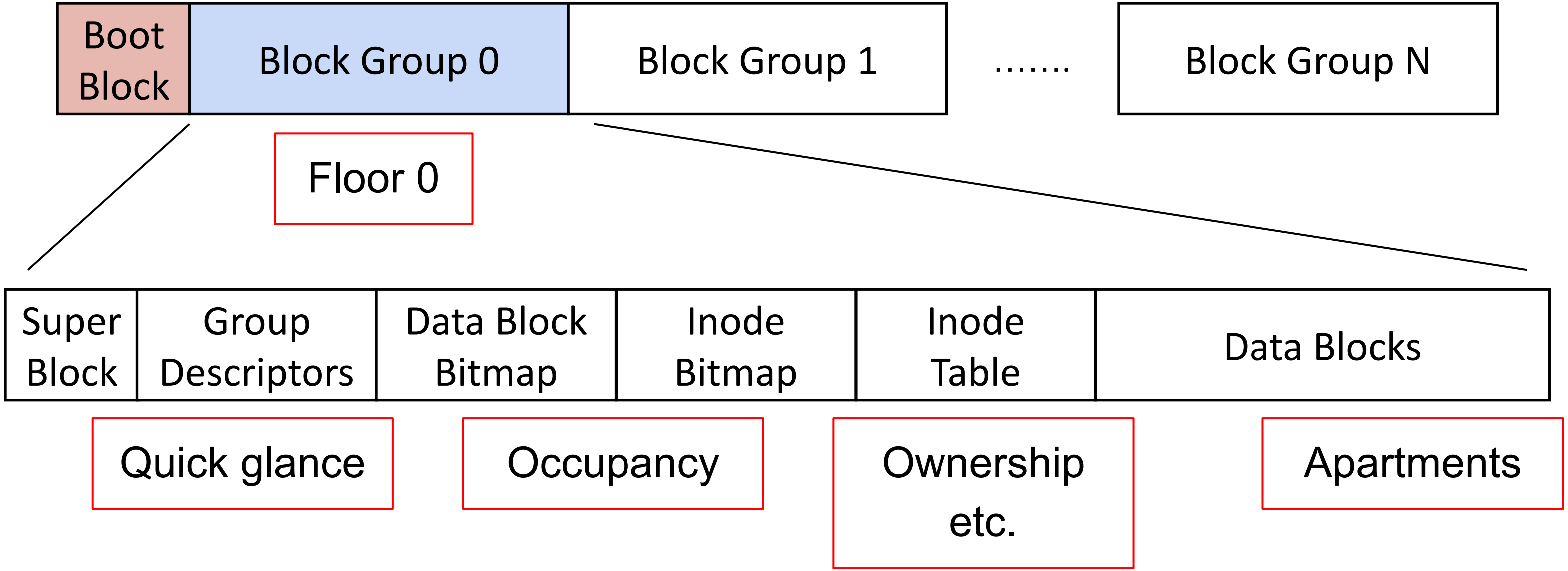
A directory is also a file

- A directory store file names and meta information about each of the files.
- A directory file is basically a list of inodes with their assigned names, the list includes an entry for itself, its parent, and each of its children.
- A directory can also contain other directories (subdirectories)
- A directory tree includes a directory and all of its files, and starts at root



Lab4 Background:
Back to EXT2

Structure of an Ext2 file system



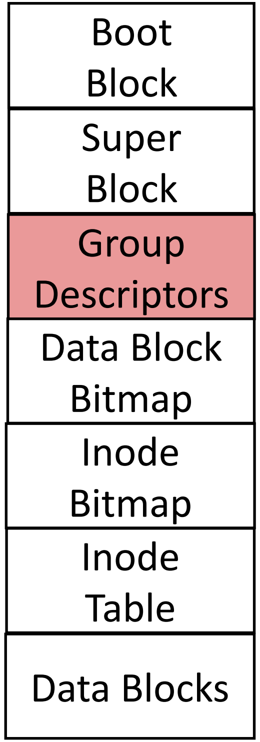
What is a block anyway? How are bytes organized inside?

- Continuous chunk of bytes (simply speaking)
- Check Appendix 1; recommended as highly related to detailed implementation.
- We skip super/boot block in the discussion; Check Appendix 2 if interested

Block Group Descriptor

- Located in the *next block after super block*
- Provides an overview of how the volume is split into block groups and where to find the inode bitmap, the block bitmap, and the inode table for each block group.
- An array of block group descriptors:
    - each representing a block group on the disk
    - records the general information of the block groups
        - for example: number of free blocks in the block group, the location of block bitmap, inode bitmap, inode table.
        - Apartment analogy: number of free apartment, the location of the records: apartment occupancy, ownerships etc.

In Ext2, each block group consists of a copy superblock, a copy of the block group descriptor table, a data block bitmap, an inode bitmap, an inode table, and data blocks.



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];
};
```

```
struct ext2_block_group_descriptor
{
    u32 bg_block_bitmap;
    u32 bg_inode_bitmap;
    u32 bg_inode_table;
    u16 bg_free_blocks_count;
    u16 bg_free_inodes_count;
    u16 bg_used_dirs_count;
    u16 bg_pad;
    u32 bg_reserved[3];
};
```

Boot Block

Super Block

Group Descriptors

Data Block Bitmap

Inode Bitmap

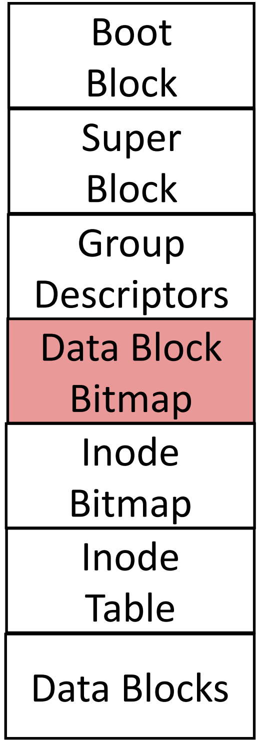
Inode Table

Data Blocks

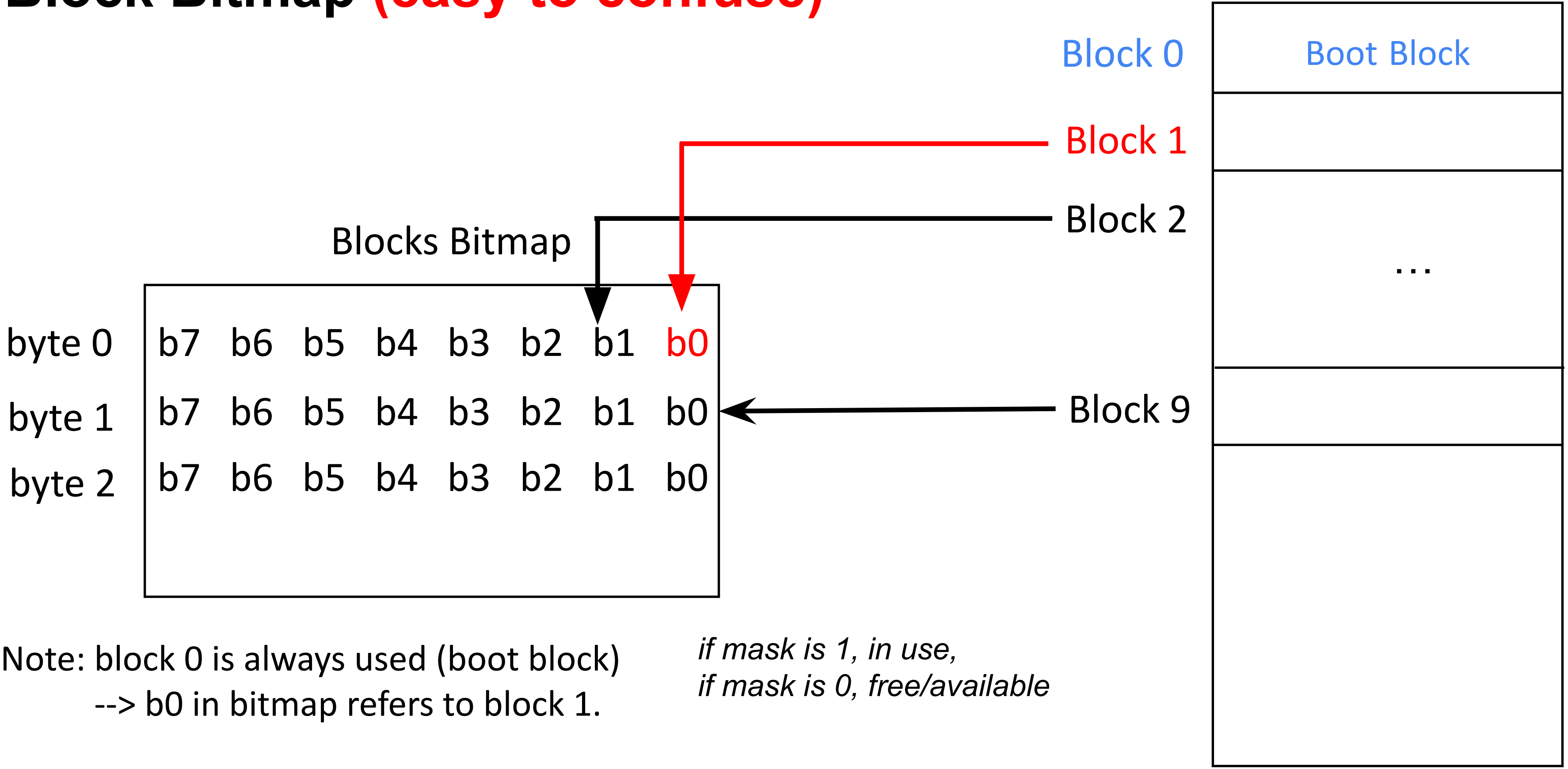
Block Bitmap (Apartment analogy: occupancy)

- A sequence of 0 & 1 bits
- Each bit represents a specific block in the block group
- Indicating whether a block has been used or not
    - 1: the block is used (occupied by files or used by file system)
    - 0: the block is free (can be used by newly created/enlarged files)

- The block bitmap must be stored in a single block
    - if the block size in bytes is b
    - there can be at most 8 * b blocks in each block group



Block Bitmap (easy to confuse)



Note: block 0 is always used (boot block)
--> b0 in bitmap refers to block 1.

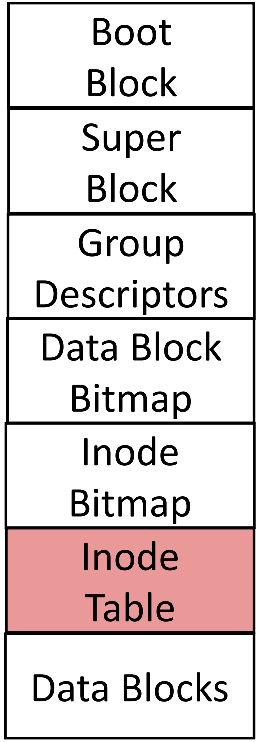
if mask is 1, in use,
if mask is 0, free/available

Inode Table (Apartment analogy: ownership etc)

- 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 */
}
```

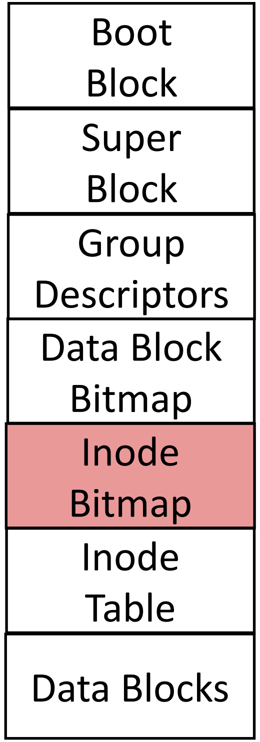
```
void write_inode_table(int fd) {
    u32 current_time = get_current_time();

    struct ext2_inode lost_and_found_inode = {0};
    lost_and_found_inode.i_mode = EXT2_S_IFDIR
                              | EXT2_S_IRUSR
                              | EXT2_S_IWUSR
                              | EXT2_S_IXUSR
                              | EXT2_S_IRGRP
                              | EXT2_S_IXGRP
                              | EXT2_S_IROTH
                              | EXT2_S_IXOTH;
    lost_and_found_inode.i_uid = 0;
    lost_and_found_inode.i_size = 1024;
    lost_and_found_inode.i_atime = current_time;
    lost_and_found_inode.i_ctime = current_time;
    lost_and_found_inode.i_mtime = current_time;
    lost_and_found_inode.i_dtime = 0;
    lost_and_found_inode.i_gid = 0;
    lost_and_found_inode.i_links_count = 2;
    lost_and_found_inode.i_blocks = 2; /* These are oddly 512 blocks */
    lost_and_found_inode.i_block[0] = LOST_AND_FOUND_DIR_BLOCKNO;
    write_inode(fd, LOST_AND_FOUND_INO, &lost_and_found_inode);

    /* You should add your 3 other inodes in this function and delete this
       comment */
}
```



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

Inode Bitmap

- Indicate whether an `inode` is used or not.
- Exactly the same as block bitmap. Similarities and differences:
  - Also starts at 1 (in ext2)
  - But each bit now refers to a inode, instead of a data block
- No analogy in apartment example, because we usually don't have a limit of records in reality; Here, the limit is 8*block_bytes



How data is stored

- `i_block` can be used to retrieval all the data belonging to the file

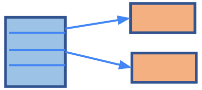
```
struct ext2_inode {
        __u16  i_mode;       /* File mode */
        …
        __u32  i_block[EXT2_N_BLOCKS];
               /* Pointers to data blocks of file */
}
```

```
#define EXT2_NDIR_BLOCKS 12                     /* number of direct blocks */
#define EXT2_IND_BLOCK   EXT2_NDIR_BLOCKS       /* single indirect block   */
#define EXT2_DIND_BLOCK  (EXT2_IND_BLOCK + 1)   /* double indirect block   */
#define EXT2_TIND_BLOCK  (EXT2_DIND_BLOCK + 1)  /* trible indirect block   */
#define EXT2_N_BLOCKS    (EXT2_TIND_BLOCK + 1)  /* total number of blocks  */
```

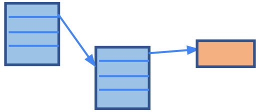
Lab demonstration by Jon

```
struct ext2_inode {
    u16 i_mode;
    u16 i_uid;
    u32 i_size;
    u32 i_atime;
    u32 i_ctime;
    u32 i_mtime;
    u32 i_dtime;
    u16 i_gid;
    u16 i_links_count;
    u32 i_blocks;
    u32 i_flags;
    u32 i_reserved1;
    u32 i_block[EXT2_N_BLOCKS];
```

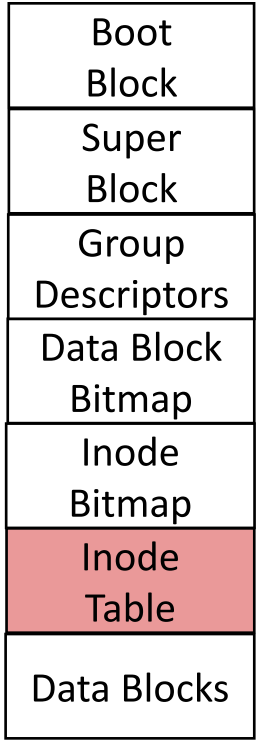
`i_block[0..11]` point directly to the first 12 data blocks of the file



`i_block[12]` points to a single indirect block

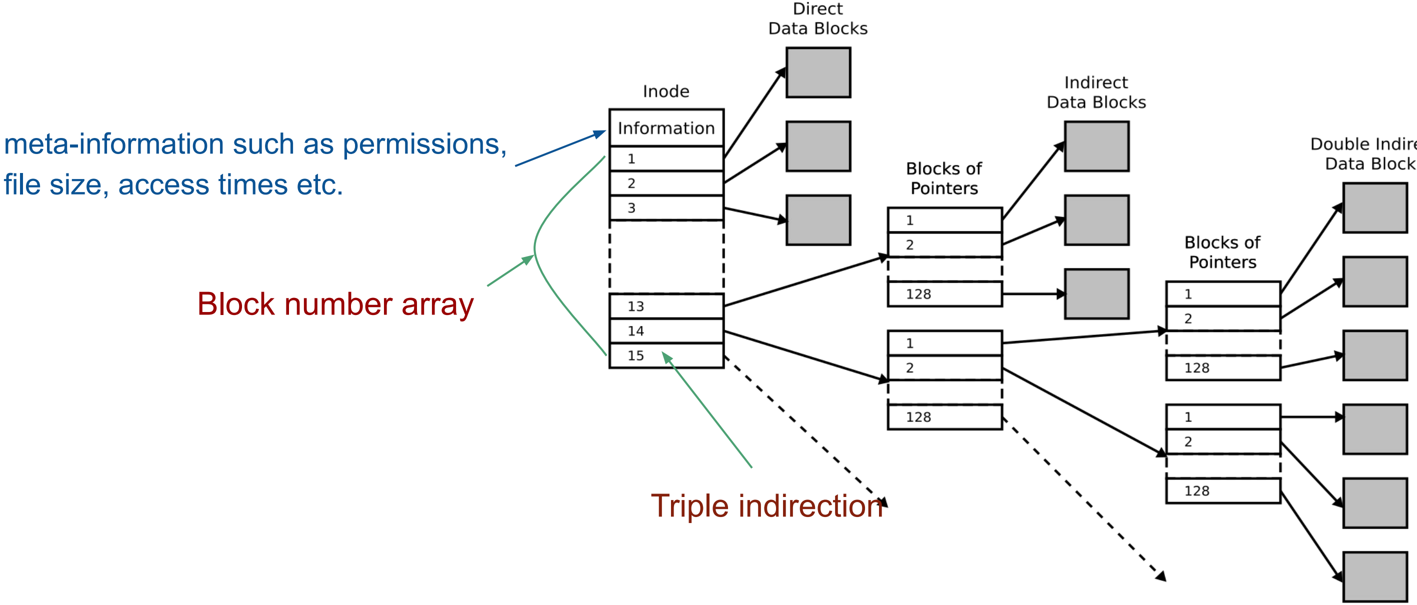


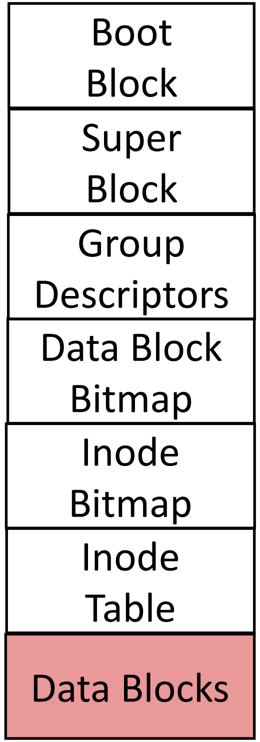
`i_block[13]` points to a double indirect block
`i_block[14]` points to a triple indirect block



How data is stored

- Here the array elements 1-12 point directly to the actual data blocks
- From 13-15 point to pointer blocks, which eventually also point to some data blocks
- This is done so to support big file size.
- In most cases, those 12 directly pointed blocks can store the whole file content.





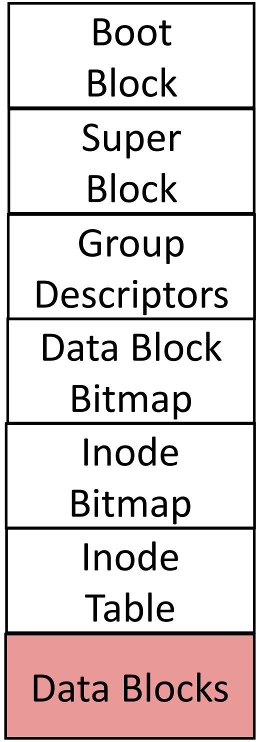
Special file type: directory

- Ext2 implements directories as a special kind of file, which contain file names together with the corresponding inode numbers
- A directory file has a list of ext2_dir_entry
  - Each entry refers to a file in this directory
  - has a variable size depending on the length of the file name
  - The maximum length of a file name is EXT2_NAME_LEN (usually 255)
- Each entry contains the following information:

```
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 */
};
```

```
struct ext2_dir_entry {
    u32 inode;
    u16 rec_len;
    u16 name_len;
    u8  name[EXT2_NAME_LEN];
};
```

```
#define EXT2_NAME_LEN 255
```



Lab4 Starter

What is it about ?

“In this lab you’ll be making a 1 MiB ext2 file system with 2 directories, 1 regular file, and 1 symbolic link”

What are you given?

- `ext2` structures and some initial skeleton code which creates a file called `cs111-base.img` in the current working directory

What is it about ?

- **Make an Ext2 FS**
  - 2 directories
  - 1 file
  - 1 symbolic link

- **Skeleton Code**
  - Fill in missing field values
  - Implement incomplete functions

How to get started?

Download the skeleton for Lab 4 from BruinLearn.

You should be able to run `make` in the lab4 directory to create a ext2-create executable.

When you run the executable (`./ext2-create`) it creates `cs111-base.img`.

```
cs111@cs111 cs111/lab4 (main *%) » make
cc -std=gnu17 -Wpedantic -Wall -O0 -pipe -fno-plt -fPIC   -c -o ext2-create.o ext2-create.c
ext2-create.c: In function ‘write_superblock’:
ext2-create.c:193:13: warning: unused variable ‘current_time’ [-Wunused-variable]
  193 |         u32 current_time = get_current_time();
      |             ^~~~~~~~~~~~
cc -lrt -Wl,-O1,--sort-common,--as-needed,-z,relro,-z,now  ext2-create.o   -o ext2-create
cs111@cs111 cs111/lab4 (main *%) » ls
ext2-create  ext2-create.c  ext2-create.o  Makefile  README.md
cs111@cs111 cs111/lab4 (main *%) » ./ext2-create
cs111@cs111 cs111/lab4 (main *%) » ls
cs111-base.img  ext2-create  ext2-create.c  ext2-create.o  Makefile  README.md
```

How to get started?

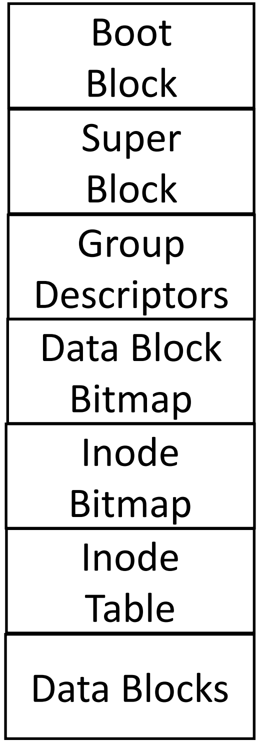
You can then run `fsck.ext2 cs111-base.img`, and will likely be asked to fix **(many) errors**.

```
cs111@cs111 cs111/lab4 (main *%) » fsck.ext2 cs111-base.img
e2fsck 1.46.4 (18-Aug-2021)
ext2fs_open2: Bad magic number in super-block
fsck.ext2: Superblock invalid, trying backup blocks...
fsck.ext2: Bad magic number in super-block while trying to open cs111-base.img

The superblock could not be read or does not describe a valid ext2/ext3/ext4
filesystem.  If the device is valid and it really contains an ext2/ext3/ext4
filesystem (and not swap or ufs or something else), then the superblock
is corrupt, and you might try running e2fsck with an alternate superblock:
    e2fsck -b 8193 <device>
 or
    e2fsck -b 32768 <device>
```

At the end of this lab you're expected to have **no errors** after running `fsck.ext2`.

```
cs111@cs111 cs111/lab4-pan (main *%) » make
cc -std=gnu17 -Wpedantic -Wall -O0 -pipe -fno-plt -fPIC   -c -o ext2-create.o ext2-create.c
cc -lrt -Wl,-O1,--sort-common,--as-needed,-z,relro,-z,now  ext2-create.o   -o ext2-create
cs111@cs111 cs111/lab4-pan (main *%) » ./ext2-create
cs111@cs111 cs111/lab4-pan (main *%) » fsck.ext2 cs111-base.img
e2fsck 1.46.4 (18-Aug-2021)
cs111-base has gone 0 days without being checked, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
cs111-base: 13/128 files (0.0% non-contiguous), 24/1024 blocks
```

Files to modify (`ext2-create.c`)



- `write_superblock`
- `write_block_group_descriptor_table`
- `write_block_bitmap`
- `write_inode_bitmap`
- `write_inode_table`
- `write_root_dir_block`
- `write_hello_world_file_block`

What’s your todo list?

1. Define your user defined data types (`typedef`)
2. Define three Macros (`#define`)
3. Set `superblock` fields correctly in function `write_superblock()`
4. Set `block_group_descriptor` fields correctly in function `write_block_group_descriptor_table()`

5. Implement the whole function `write_block_bitmap()`
6. Implement the whole function `write_inode_bitmap()`
7. Add 3 other inodes in function `write_inode_table()`
8. Implement the whole function `write_root_dir_block()`
9. Implement the whole function `write_hello_world_file_block()`

10. Modify `README.md`

Lab demonstration by Jon !!!

Debugging Commands

These are all the commands you'll likely want to use:

```
make # compile the executable
./ext2-create # run the executable to create cs111-base.img


dumpe2fs cs111-base.img # dumps the filesystem information to help debug


fsck.ext2 cs111-base.img # this will check that your filesystem is correct


mkdir mnt # create a directory to mnt your filesystem to
sudo mount -o loop cs111-base.img mnt # mount your filesystem, loop lets you use a file
ls -ain mnt/ # list all files

sudo umount mnt # unmount the filesystem when you're done
rmdir mnt # delete the directory used for mounting when you're done
```

Debugging Commands

Dump FS information

- print the super block and blocks group information for the FS

```
dumpe2fs cs111-base.img # dumps the filesystem information to help debug
```

```
cs111@cs111 cs111/lab4-pan (main *%) » dumpe2fs cs111-base.img
dumpe2fs 1.46.4 (18-Aug-2021)
Filesystem volume name:   cs111-base
Last mounted on:          <not available>
Filesystem UUID:          5a1eab1e-1337-1337-1337-c0ffeec0ffee
Filesystem magic number:  0xEF53
Filesystem revision #:    0 (original)
Filesystem features:      (none)
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              128
Block count:              1024
Reserved block count:     0
Free blocks:              1000
Free inodes:              115
First block:              1
Block size:               1024
Fragment size:            1024
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         128
Inode blocks per group:   16
Last mount time:          n/a
Last write time:          Thu Nov 18 21:29:22 2021
```

Debugging Commands

Check FS correctness

```
fsck.ext2 cs111-base.img # this will check that your filesystem is correct
```

```
cs111@cs111 cs111/lab4-pan (main *%) » fsck.ext2 cs111-base.img
e2fsck 1.46.4 (18-Aug-2021)
cs111-base has gone 0 days without being checked, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
cs111-base: 13/128 files (0.0% non-contiguous), 24/1024 blocks
```

Debugging Commands

Check directory and permissions

- print the super block and blocks group information for the FS

```
mkdir mnt # create a directory to mnt your filesystem to
sudo mount -o loop cs111-base.img mnt # mount your filesystem, loop lets you use a file
ls -ain mnt/ # list all files
```

```
cs111@cs111 cs111/lab4-pan (main *%) » mkdir mnt
cs111@cs111 cs111/lab4-pan (main *%) » sudo mount -o loop cs111-base.img mnt
cs111@cs111 cs111/lab4-pan (main *%) » ls -ain mnt/
total 7
      2 drwxr-xr-x 3    0    0 1024 Nov 18 21:29 .
942565 drwxr-xr-x 3 1000 1000 4096 Nov 18 22:07 ..
     13 lrw-r--r-- 1 1000 1000   11 Nov 18 21:29 hello -> hello-world
     12 -rw-r--r-- 1 1000 1000   12 Nov 18 21:29 hello-world
     11 drwxr-xr-x 2    0    0 1024 Nov 18 21:29 lost+found
```

- `ls -a` # list all files and directories, including hidden files
- `ls -i` # list inode informations
- `ls -n` # list UID and GID number of Owner and Groups to which the files and directories are belongs.

Acknowledgement

1. Previous TAs: Pan Lu, Tianxiang Li, Salekh Parkhati, and Alex Tiard
2. Jonathan Eylofson, UCLA Computer Science, Fall21, cs111 course slides
3. https://en.wikipedia.org/wiki/File_system
4. https://en.wikipedia.org/wiki/Inode
5. https://www.youtube.com/watch?v=YRpUVGZ2uB4
6. http://www.science.smith.edu/~nhowe/262/oldlabs/ext2.html
7. http://www.nongnu.org/ext2-doc/ext2.html
8. https://piazza.com/class_profile/get_resource/il71xfllx3l16f/inz4wsb2m0w2oz
9. https://students.mimuw.edu.pl/ZSO/Wyklady/11_extXfs/extXfs.pdf

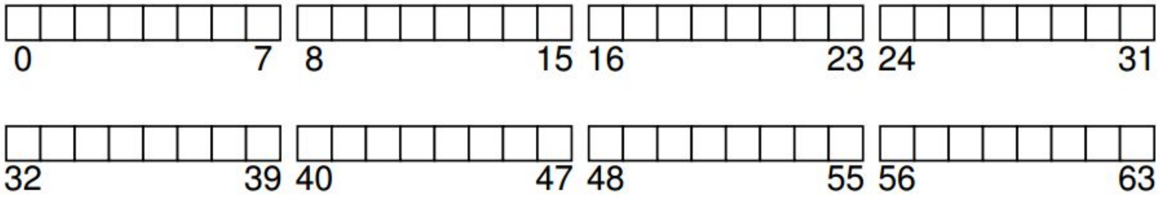
Thanks!

Appendix 1:
Detailed formal descriptions

Data structures of the file system

Blocks

- We divide the disk into blocks
- For example
  - We can choose a commonly-used size of 4 KB
  - We're building our file system on a series of blocks, each of size 4 KB.
  - The blocks are addressed from 0 to N − 1, in a partition of size N 4-KB blocks
  - Assume we have a really small disk, with just 64 blocks

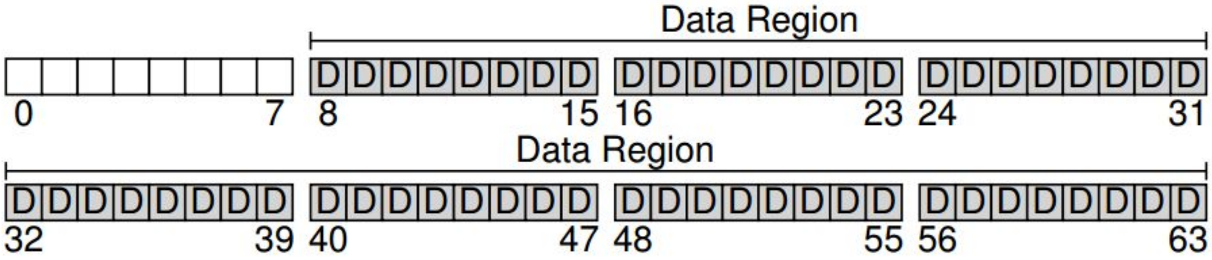


Question: what do we need to store in these blocks to build a file system?

Data structures of the file system

Data Region

- Most of the space in any file system is user data
- The region of the disk we use for user data is called the data region
- We can reserve a fixed portion of the disk for these blocks
    - e.g., the last 56 of 64 blocks on the disk



Question: Aside from the user data, what else do we need to store in these blocks?

Data structures of the file system

metadata

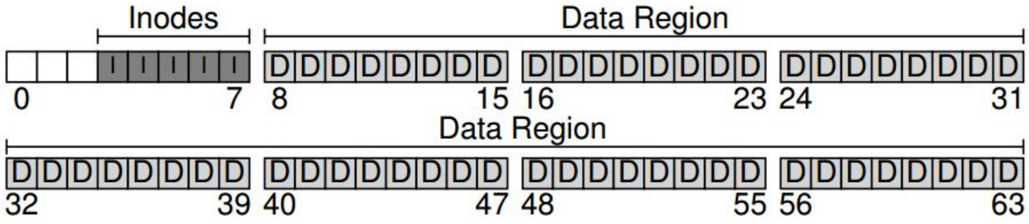
- The file system has to track information (metadata) about each file
- Tracks which data blocks comprise a file, the size of the file, its owner and access rights, access and modify times, ...

inode

- To store this information, file system usually have a structure called an inode

inode Table

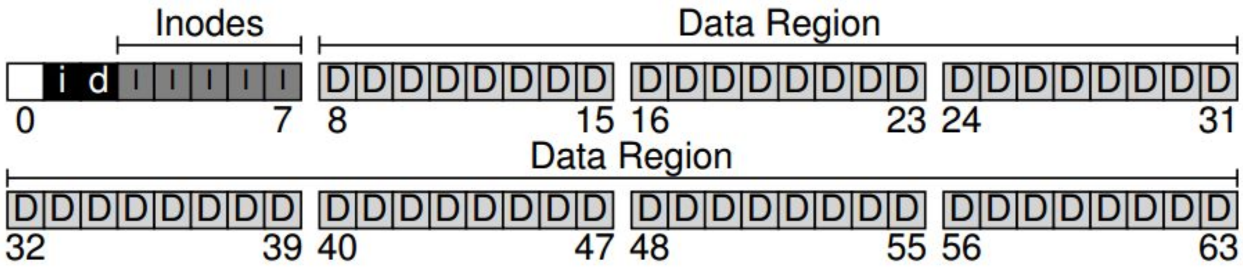
- The space on the disk reserved to accommodate inodes is called the inode table



Data structures of the file system

bitmap

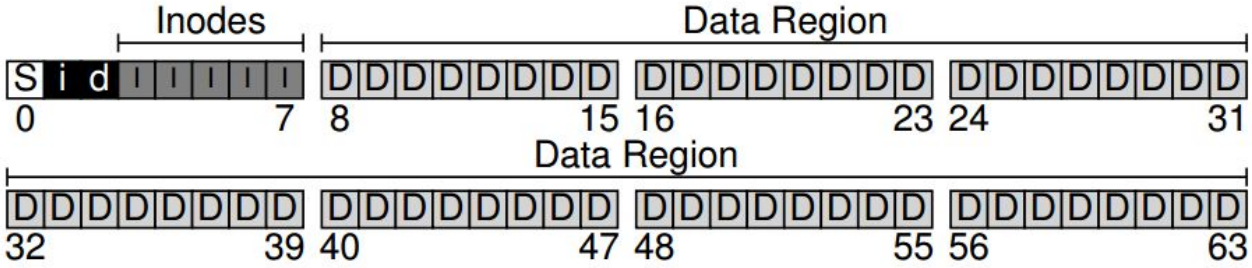
- We need to track whether inodes or data blocks are free or allocated.
- Such allocation structures are implemented by a simple structure: a bitmap
- We have two bitmaps
  - the **data bitmap** for the data region
  - the **inode bitmap** for the inode table
- Each bit is used to indicate whether the corresponding block is free (0) or in-use (1)



Data structures of the file system

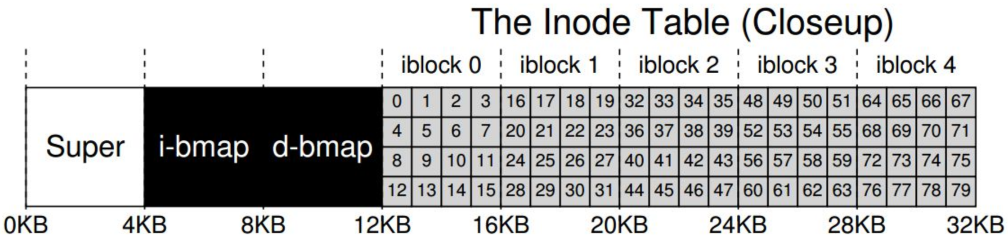
superblock

- The first block is reserved for the superblock
- The superblock contains information about this particular file system:
    - how many inodes and data blocks (56) are in the file system
    - where the inode table begins (block 3)
    - a magic number of some kind to identify the file system type
- When mounting a file system, the operating system will read the superblock first



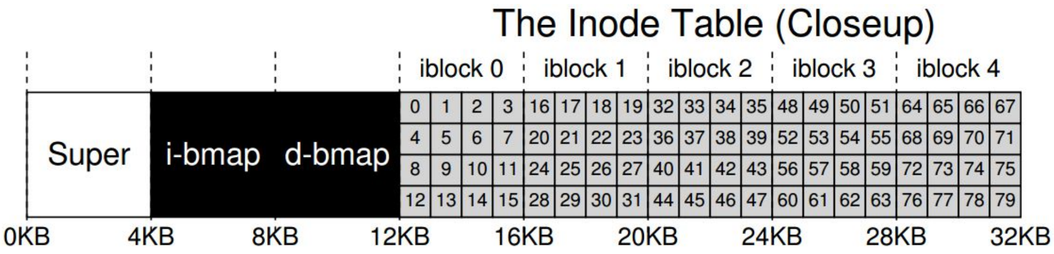
File organization: the inode

- Each inode is implicitly referred to by a number (called the inumber)
- Given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located.



- For example,
  - take the inode table: 20-KB in size (5 4-KB blocks)
  - 80 inodes (assuming each inode is 256 bytes)
  - superblock starts at 0KB, inode bitmap is at address 4KB, data bitmap at 8KB
  - And thus the inode table comes right after, starts at 12KB

File organization: the inode



- To read inode number 32,
  - the file system calculates the offset into the inode region:
    - 32·`sizeof(inode)` or 8192 = 8KB
  - add it to the start address of the inode table on disk
    - `inodeStartAddr` = 12KB
  - the correct byte address of the desired block of inodes
    - 8 + 12 = 20KB

File organization: the ext2 inode

Inside each inode is virtually all of the information you need about a file:

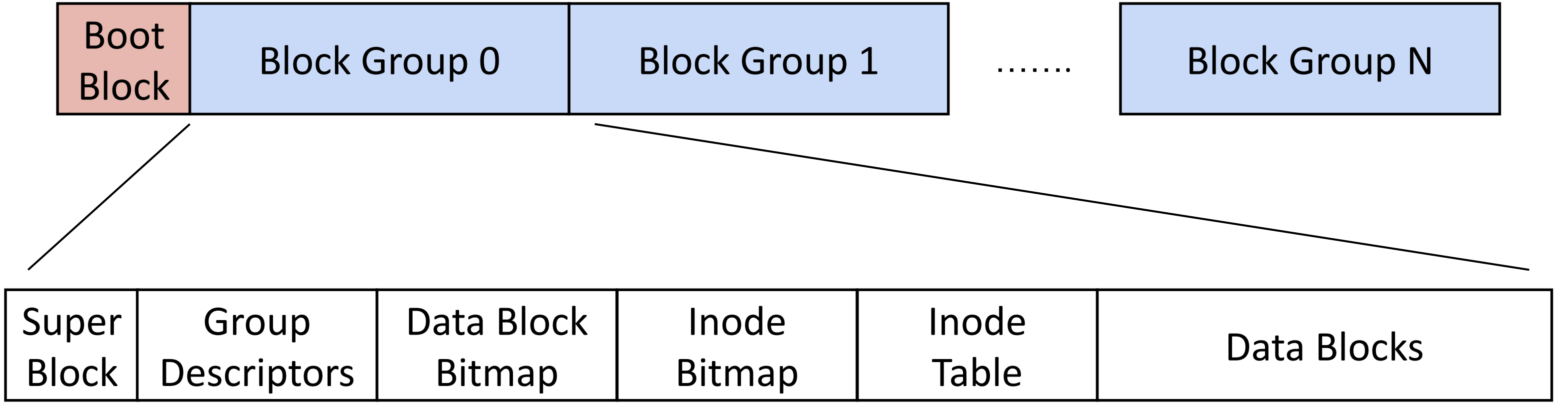
Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	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

```
struct ext2_inode {
    u16 i_mode;
    u16 i_uid;
    u32 i_size;
    u32 i_atime;
    u32 i_ctime;
    u32 i_mtime;
    u32 i_dtime;
    u16 i_gid;
    u16 i_links_count;
    u32 i_blocks;
    u32 i_flags;
    u32 i_reserved1;
    u32 i_block[EXT2_N_BLOCKS];
    u32 i_version;
    u32 i_file_acl;
    u32 i_dir_acl;
    u32 i_faddr;
    u8  i_frag;
    u8  i_fsize;
    u16 i_pad1;
    u32 i_reserved2[2];
};
```

We refer to all such information about a file as metadata.

Appendix 2:
Boot/Super Block

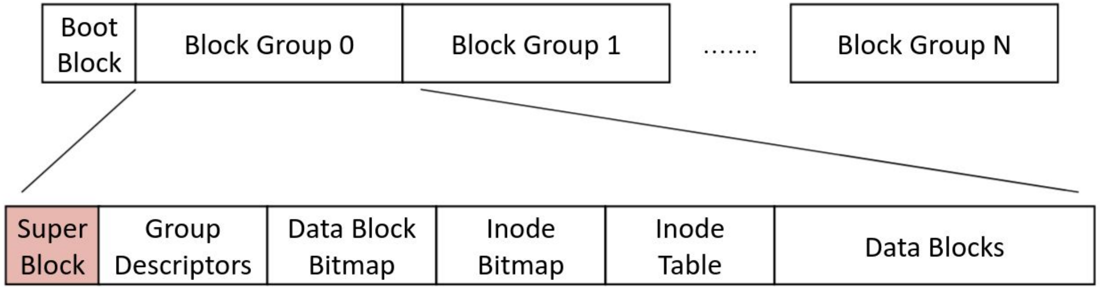
Structure of an Ext2 file system



- **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

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



Note: Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption.

Super Block: 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 */
        ...
}
```

```
struct ext2_superblock {
    u32 s_inodes_count;
    u32 s_blocks_count;
    u32 s_r_blocks_count;
    u32 s_free_blocks_count;
    u32 s_free_inodes_count;
    u32 s_first_data_block;
    u32 s_log_block_size;
    i32 s_log_frag_size;
    u32 s_blocks_per_group;
    u32 s_frags_per_group;
    u32 s_inodes_per_group;
    u32 s_mtime;
    u32 s_wtime;
    u16 s_mnt_count;
    i16 s_max_mnt_count;
    u16 s_magic;
```

Boot Block
Super Block
Group Descriptors
Data Block Bitmap
Inode Bitmap
Inode Table
Data Blocks

`s_blocks_count / s_blocks_per_group` = Number of block groups

`s_magic`: shows the file system on the disk

- This allows the mounting software to check that this is indeed the Superblock for an Ext2 file system.
- For the current version of Ext2 this is **0xEF53**.

Appendix 3: Related APIs

Creating files

- By calling `open()` and passing it the `O_CREAT` flag, a program can create a new file
- It returns a file descriptor
    - A file descriptor is just an **integer**, private per process, and is used in UNIX systems to access files
    - Thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so

```
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);


// two examples
int fd = open("foo.txt", O_CREAT | O_WRONLY | O_TRUNC);
int fd = open("cs111-base.img", O_CREAT | O_WRONLY, 0666);
```

https://linux.die.net/man/3/open

Reading and writing files sequentially

- Using the system calls: `read()` and `write()`

```
size_t read (int fd, void* buf, size_t cnt);
size_t write (int fd, void* buf, size_t cnt);

// example
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
sz = read(fd, c, 10); // read 10 bytes
sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));
```

*Thus far, we've discussed how to read and write files, but all access has been **sequential**; that is, we have either read a file from the beginning to the end, or written a file out from beginning to end.*

Reading and writing files with `lseek`

- Sometimes, it is useful to be able to read or write to a specific **offset within a file**
- For example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some random offsets within the document.
- To do so, we will use the `lseek()` system call.

```
off_t lseek(int fildes, off_t offset, int whence);
```

- `fildes`: a file descriptor
- `offset`: positions the file offset to a particular location within the file
- `whence`: determines exactly how the seek is performed
  - `SEEK_SET`: the offset is set to offset bytes
  - `SEEK_CUR`: the offset is set to its current location plus offset bytes
  - `SEEK_END`: the offset is set to the size of the file plus offset bytes

Appendix 4:
Pros and cons of large group size

Advantages of large block groups:

- Reduced metadata overhead: Large block groups can result in fewer metadata structures.
- Efficient space utilization: With larger block groups, there is less wasted space due to internal fragmentation.
- Simplified administration: Large block groups can simplify file system administration by reducing the number of groups that need to be managed.

Disadvantages of large block groups:

- Increased recovery time: In the event of a file system corruption or failure, recovering large block groups can be more time-consuming.
- Reduced flexibility: Large block groups can limit the ability to efficiently allocate small files. If most of the block group's size is larger than the average file size, it can result in wasted space when storing numerous small files.
- Uneven space distribution.