UNIX I-Node File System Structure

Problem:

In this task implement a Unix file system structure and a management system to manage the bytes of our system wanted functionalities(dir, mkdir, rmdir, dumpe2fs, write, read, del).

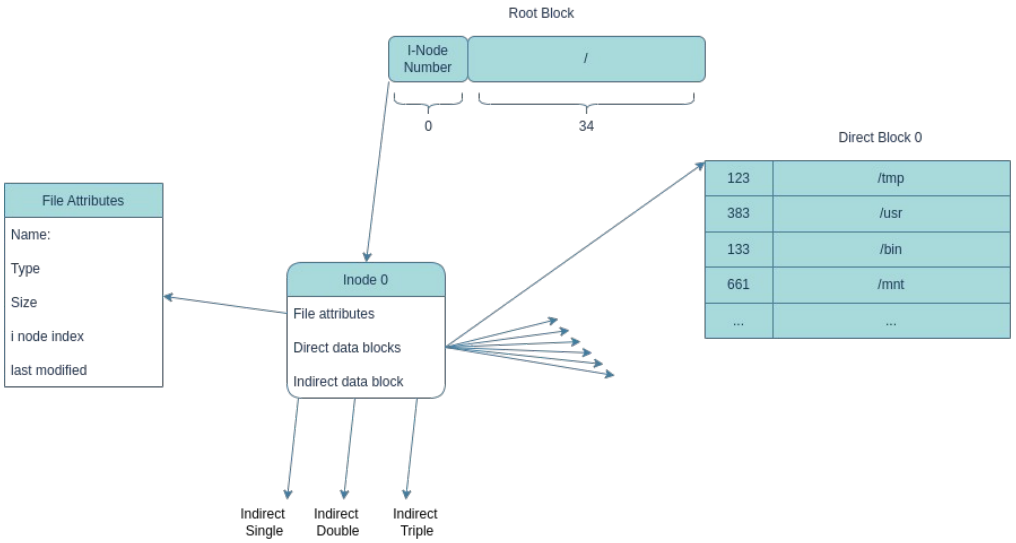
First, let's look at the file system in general and its terminology.

Disk divides some parts, and these parts require stored data and some of them tell us where these data are stored in the disk.

Let us take a closer look at the Linux file system before going deep into them.

Directories and Folders:

Directories are also files in the Linux system and their meta data not the content of the directory is stored in the I-node. These I-nodes keep the metadata of the file for a directory it keeps the size of the directory, last modified time, creation time, name of the directory and it keeps the type of variable because the I-node uses also files we need to keep a variable to determine these I-nodes is directory I-node. I use two different values to avoid this mixing. We will examine it in more detail in the I-node section.



Basic diagram of Directories

Files:

We would mention that directories and folders are similar, like files. Ironically, the concept of the file is just a concept that has no conceptual place in the implementation of storage created for the user to understand it more easily. We are abstracting the complexity of the I-Node and blocks in Unix file system implementation. Files are a bunch of characters or more technically bunch of bytes. But we have some issues with these definitions. Where to keep these bunch of bytes, how to access them when we need them and how to organize them. There is a particularly good scheme for the file structure in our book.

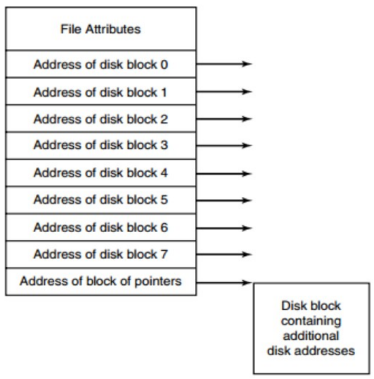


Figure 4-13. An example i-node.

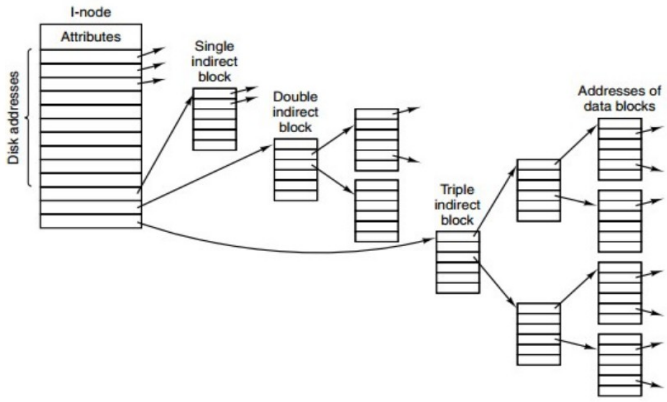


Figure 4-33. A UNIX i-node.

Let us answer the question;

1-) Where to keep these bunch of bytes?

We have a special disk place divided into a specified sized disk block. It is the end of our disk and stores these bytes.

2-) How to access them?

We talked about I-nodes. I-Nodes keep the metadata about files and keep the file data location in our disk. There are two different address pointers. The first pointer group directly points to the direct block address. The second group indirectly points to the data block. These pointers point to a block and these blocks keep the data block of the file.

3-) How to organize them?

Before the I-Node structure, there's also another file structure, this is a contagious file structure. This structure is very easy to organize the data blocks because its linear stores the data (But there is another disadvantage). In the I-Node structure, we keep the address of the data block. These allow the blocks to be separated on the disk. But we need the address in I-Node. We organize the data using these direct and indirect pointers.

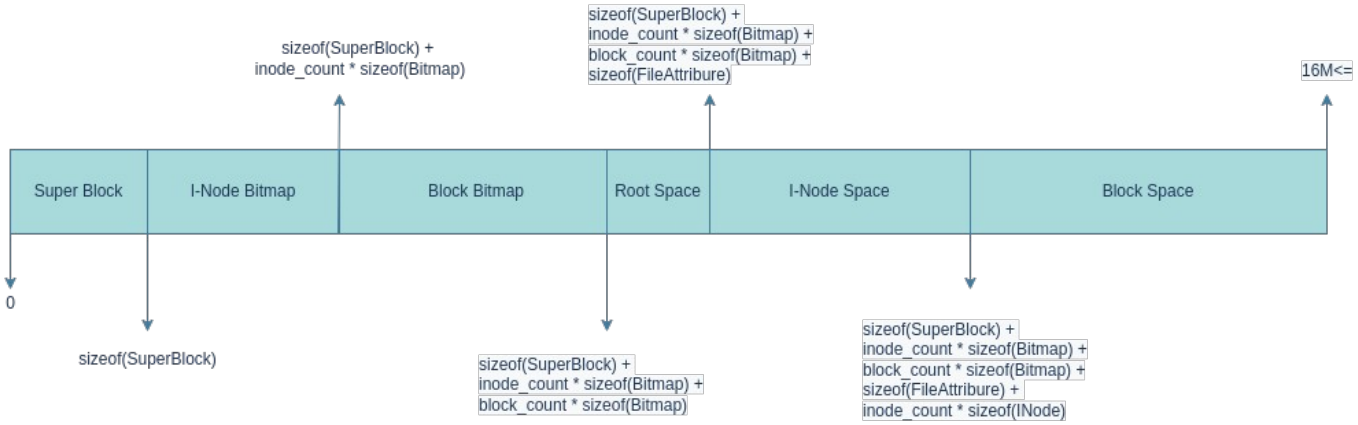
But how to access the I-Node to get data?

We will get to that in the my Implementation.

If we understand the basics, we can move on to my implementation and deeper information about the blocks.

My I-Node File System Implementation:

General Disk Design:



I try to avoid it as much as possible by using static numbers to make it easy to change and to make it more generic according to the parameters. For example, if we increase the i-Node Number it causes less block space or if we decrease the I-Node number it causes more space for the data blocks.

It is like what I told you. Except there are some untouched blocks like bitmap and super block let us dive into them.

Super Block:

```
You, 5 hours ago | 1 author (You)
typedef struct SuperBlock{

    public:
        long long file_system_size;         // File system size in bytes

        long long block_bitmap_start;       // Block bitmap start in bytes
        long long inode_bitmap_start;       // Inode bitmap start in bytes

        long long root_start;               // Root start in bytes

        long long block_count;              // Block count
        long long block_size;               // Block size in bytes
        long long block_start;              // Block start in bytes

        long long inode_count;              // Inode count
        long long inode_size;               // Inode size in bytes
        long long inode_start;              // Inode start in bytes

}SuperBlock;
```

Super Blocks' duty is to keep the information about which block starts where so We can access them easily using super block variables. And Superblock is the first initialize in the disk (ignore the boot space) So it starts from 0 to sizeof(SuperBlock). There is information about which fields for which offset or count.

- file_system_size; // File system size in bytes
- block_bitmap_start; // Block bitmap start in bytes
- node_bitmap_start; // Inode bitmap start in bytes
- root_start; // Root start in bytes
- block_count; // Block count
- block_size; // Block size in bytes
- block_start; // Block start in bytes
- inode_count; // Inode count
- inode_size; // Inode size in bytes
- inode_start; // Inode start in bytes

We told about reaching file is happening from the I-Node block and then ask the question about so how we reach the I-Node. It is reaching from the super block I-Node offset and size. So we can easily reach the all file in the current directory and the file system.

Free Space Management I-Node Bitmap & Block Bitmap:

It keeps a dynamic array for the organize and manage the whole I-Node and Data Block space. It is very necessary to keep track of which block has data or which block is available to store data.

- If block in use array value is 1 this means that it blocks uses from another file or data.
- If block in use array value is 0 this means that it blocks is available to store data.
- If free blocks array value is 1 this means that it blocks is available to store data.
- If free blocks array value is 0 this means that it blocks uses from another file or data.

If a file needs more space, then a block size then we find the next available block space and we store the rest of the data until all data is stored in the blocks. We have two block spaces in the file system one for the I-Nodes Blocks and the other for the Data Blocks, so we need two Bitmap Block and each one has block in use and free blocks field variables. That is needed for avoiding overwriting a file data or storing data in available space.

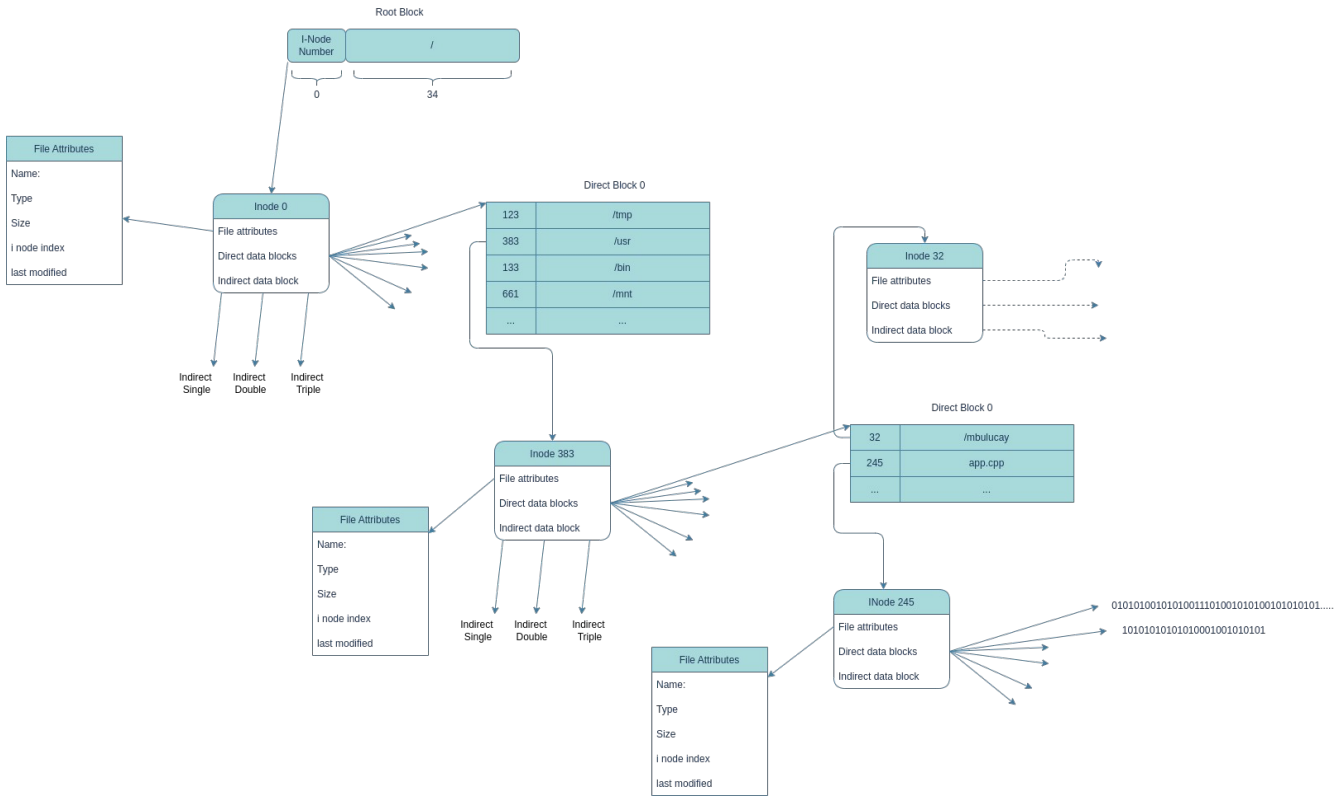
Root Space:

It is actually not needed as much as the others if we keep the root directory in a specific I-Node. Root Block keeps the I-Node Number, and we are accessing data using these I-Node if this I-Node determines the start of the program. We can directly access this I-Node. But in the task, we were asked to implement a root block in our file system.

I-Node Space:

```
You, 3 seconds ago | 1 author (You)
typedef struct FileAttribute{

    public:
        char name[MAX_FILE_NAME_SIZE];                  // File name
        long long size;                                 // File size in bytes
        long long type;                                 // File type
        long long i_node_index;                         // Inode index

        time_t last_modified;                           // Last modified time

}FileAttribute;


You, 3 seconds ago | 1 author (You)
typedef struct INode{

    public:
        FileAttribute file_attribute;                   // File attribute in INode

        long long direct_data_block[INODE_DIRECT_BLOCK_COUNT];      // Direct acces data block
        long long indirect_data_block[INODE_INDIRECT_BLOCK_COUNT];  // Indirect acces data block

}INode;
```

It is the most important block in our File System. Because the whole system is designed over the I-Node. Each I-Nodes points to file data blocks and file attributes keep the data.



I think it can be better understood if we go through the example. Let’s say we have app.cpp file in my file system and we are trying to execute it. What is its path to reach its data ”/usr/mbulucay/app.cpp” what we need to do by queue.

1 Getting the root, I-Node number from the root block
2 Getting the root directory data from our direct access data block address.
3 Reading the stored file attributes in the data block and search for the /usr file
4 After finding the /usr we are getting the /usr directory file I-Node
5 Again accessing directory data file using direct access data block addresses.
6 Searching the for /mbulucay directory and getting its I-Node
7 Then moving its I-Node and getting its data block for searchind app.cpp file
8 After finding app.cpp file we are getting its I-Node
9 And using this I-Node's Data Access parameter getting data from data block
10 And Store them in memory and execute the file

As we seen in the example Its necessary for getting file data from root to file.

Block Space:

```
70      You, 2 days ago * part2 done I guess finally
        You, 2 seconds ago | 1 author (You)
71  typedef struct Block{
72
73      public:
74          char *data;                 // Block data
75
76  }Block;
77
78
```

It is the most basic block structure because it is just seen as store value as you want. And it does not contain any metadata, it just contains data. We are calculating all blocks that need space except the data block. And we are subtracting the required size from the maximum size and the rest of them using as data blocks.

Size Calculation:

The size is calculating the as I mention the general structure part.

```
35
36  void FileSystemCreator::set_size_settings(){
37
38      long long total_size = 0;
39
40      total_size += sizeof(SuperBlock);
41
42      super_block.inode_bitmap_start = total_size;
43      total_size += sizeof(long long) * super_block.inode_count;
44      total_size += sizeof(long long) * super_block.inode_count;
45
46      calculate_block_count(total_size);
47
48      super_block.block_bitmap_start = total_size;
49      total_size += sizeof(long long) * super_block.block_count;
50      total_size += sizeof(long long) * super_block.block_count;
51
52      super_block.root_start = total_size;
53      total_size += sizeof(FileAttribute);
54
55      super_block.inode_start = total_size;
56      total_size += sizeof(INode) * super_block.inode_count;
57
58      super_block.block_start = total_size;
59      total_size += super_block.block_size * super_block.block_count;
60
61      std::cout << "Setting size settings" << std::endl;
62  }
63
64
65  void FileSystemCreator::calculate_block_count(long long used){
66
67      std::cout << "Calculating block count" << std::endl;
68
69      used += sizeof(INode) * super_block.inode_count;
70      used += sizeof(FileAttribute);
71
72      long long rest = MAX_FILE_SYSTEM_SIZE - used;
73      long long block_needed_size = super_block.block_size + (2 * sizeof(long long));
74
75      super_block.block_count = rest / block_needed_size;
76
77  }       You, now * Uncommitted changes
78
```

File System Management Used Functions:

void list_directory(char* path);

Listing directories from getting path.

void create_directory(char* path);

Create a directory file in the getting path.

void remove_directory(char* path);

Remove the the directory getting as path parameter.

void dumpe2fs();

Listing the all properties of the file system about super block, free space, files etc.

void write(char* path, char* file_path);

Creating a file in our directory and store the data block. Data is getting as file_path parameter file.

void read(char* path, char* file_path);

Reading the file content in our file system and extract it as a file normal unix system.

void delete_file(char* path);

Deleting file from the file system. And frees the used data blocks.

// File System Reading and initialization functions

void open_in_disk();

It is opening the ifstream for the file system and we can read from the file.

void open_out_disk();

It is opening the fstream for the file system and we can write to the file.

void close_in_disk();

It is closing the ifstrem not allow for reading any more.

void close_out_disk();

It is closing the fstrem not allow for writing any more.

void read_super_block();

It is reading the super block from the file system and store the data in the ram.

void print_super_block();

Printing the content of the super block in file system.

void read_inode_bitmap();

Reading the i-node bitmap from the file system and store it in the ram.

void print_inode_bitmap();

Printing the content of the i-node bitmap in file system.

void read_block_bitmap();

Reading the data block bitmap from the file system and store it in the ram.

void print_block_bitmap();

Printing the content of the inode bitmap in file system.

void read_root_block();

Reading the root file attributes in from file system and store them in the memory.

void print_root_block();

Printing the content of the root block in file system.

void read_inode_table();

Read the whole i-nodes in file system.

void print_inode_table();

Print the whole i-node in the file system

void read_block_table();

Reading the whole data block in the system.

void print_block_table();

Printing the data block in the system.

void read_inode_table_entry(int index);

Reading the i-node in file system from getting index as parameter and store it in the memory.

void print_inode_table_entry();

Printing the getting specific i-node in the memory

void read_block_table_entry(int index);

Reading the data block in file system from getting index as parameter and store it in the memory.

void print_block_table_entry();

Printing the getting specific data block in the memory

// File System Writing and updating functions

```
void write_super_block();
```

Writing the modified super block to the file.

```
void write_inode_bitmap();
```

Writing the modified i-node bitmap to the file.

```
void write_block_bitmap();
```

Writing the modified block bitmap to the file.

```
void write_inode_table_entry(int index);
```

Writing the modified i-node value in specific index location to the file.

```
void write_block_table_entry(int index);
```

Writing the modified data block value in specific index location to the file.

// Setters and Getters

Generally these functions helps us the manipulating the field area in the class.

```
void set_super_block(const SuperBlock& _super_block);
void set_block_ptr_data(const Block& block);
void set_inode_ptr_data(const INode& inode);
void set_inode_bitmap(BitMap _inode_bitmap);
void set_block_bitmap(BitMap _block_bitmap);
SuperBlock get_super_block();
```

// Helper functions

```
void parse_path(char* path, std::vector<string>& path_list);
```

Parsing the path from getting as parameter and return it.

// Helper functions for reading and writing

```
int get_next_use_inode_index();
```

It is return the next available i-node index for putting a new file using the i-node bitmap.

```
int get_next_use_block_index();
```

It is return the next available data block index for putting a new file using the i-node bitmap.

```
int get_next_free_inode_index();
```

It is return the next available i-node index for putting a new file using the i-node bitmap.

```
int get_next_free_block_index();
```

It is return the next available i-node index for putting a new file using the i-node bitmap.

```
int get_inode_number(int index, std::string str);
```

It is returning the i-node number of a file in stores in the data block it uses for the reaching the directories files i-nodes.

```
int get_block_index_number(int index, std::string str);
```

Similar like `get_inode_number` **but these returning the index of the file location in data block.**

```
void add_file_to_block(int block_index, int file_index, FileAttribute file);
```

Adding a new file to data block to specific location.

```
void remove_file_from_block(int block_index, int file_index);
```

Adding an exiting file from data block to specific location.

```
int get_next_file_index_block(int index);
```

It is returning the next available space in the data block.

```
void list_directory_in_block_index(int index);
```

Listing the directory in located by the index value.