Writing a file system from scratch in Rust

27 Jul 2020

Data produced by programs need to be stored somewhere for future reference, and there must be some sort of organisation so we can quickly retrieve the desired information. A file system (FS) is responsible for this task and provides an abstraction over the storage devices where the data is physically stored.

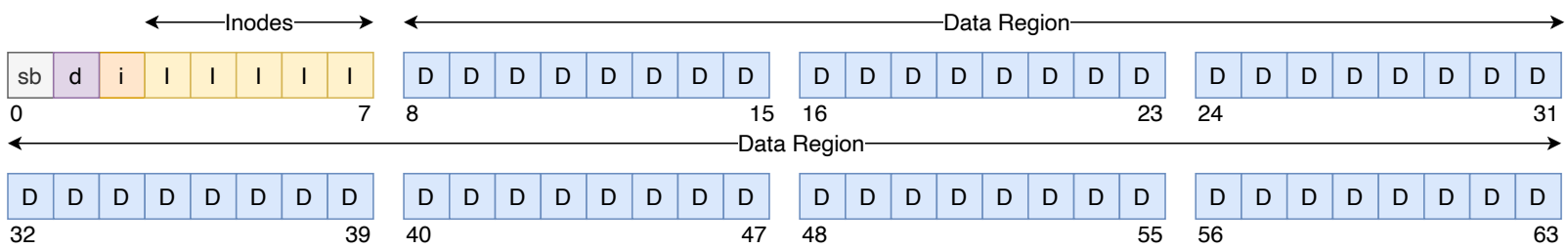
In this post, we will learn more about the concepts used by file systems, and how they fit together when writing your own.

Structuring the disk

When data is stored in a hard-disk drive (HDD) or solid-state drive (SSD), it is written in small units called sectors (or pages in the SSD case). The drives don't have any information about what that piece of data represents, for all it is worth, the disk is just a giant array of sectors. It is the job of the file system to make sense of it all.

A file system will divide the disk into fixed-sized blocks. The FS uses the majority of these blocks to store user data, but some blocks are used to store metadata that is essential for the file system operation.

The following figure gives an example of how a file system structures the information on the disk:



File system structure in a disk

In the next sections, we will understand what each type of block means.

Superblock and bitmaps

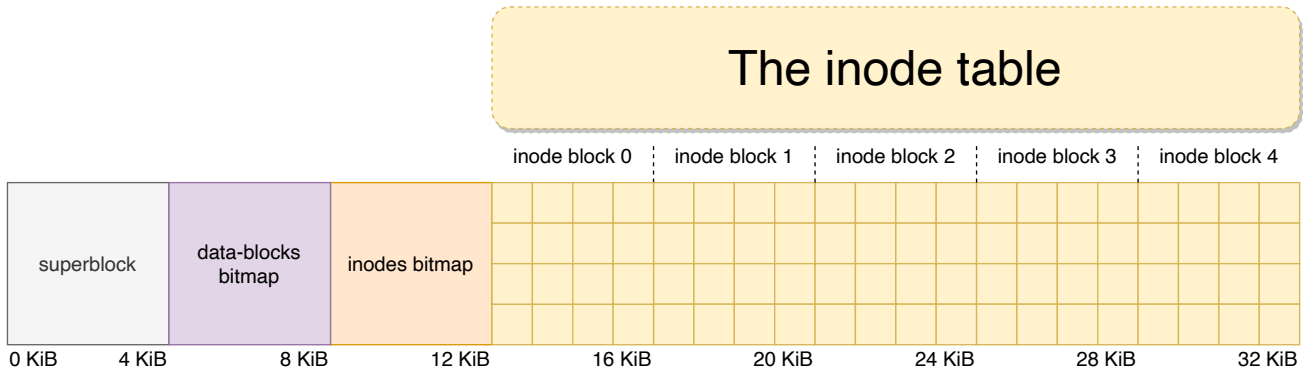
The superblock stores most of the metadata about the file system, such as the following: block size, timestamps, how many blocks and files are in use and how many are free, etc.

Bitmaps are one way of tracking which data blocks and inodes are free. An index in the bitmap set to `0` indicates a free slot, and an index set to `1` indicates an occupied slot.

Inode

The inode is the structure that stores metadata about a file. Attributes such as permissions, size, location of data blocks that form the file and more are saved in an inode.

The inodes are stored in blocks that together form the *inode table* as the following figure shows.



Inode table (detailed view)

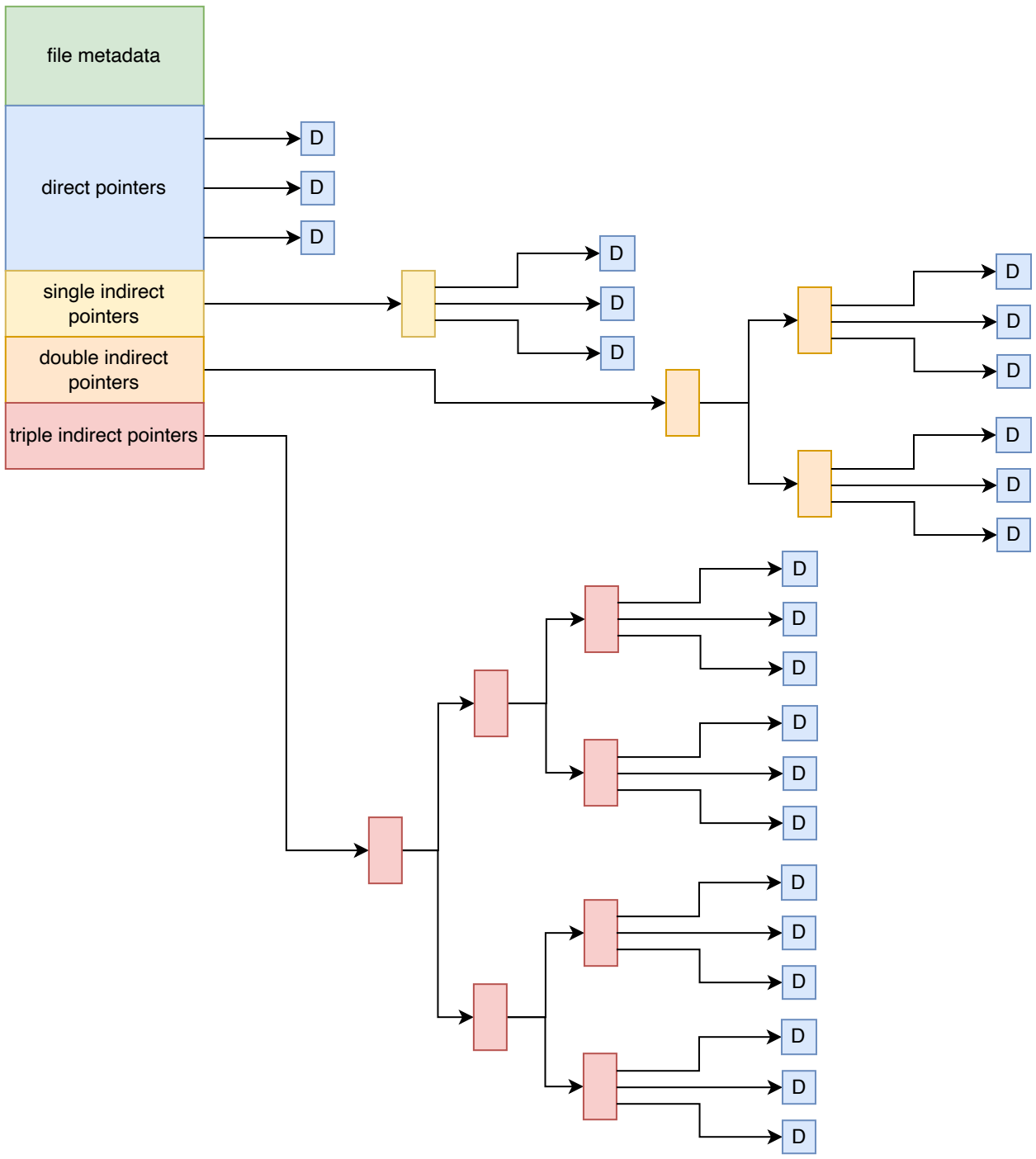
For each slot in the inode bitmap set to `1`, there will be a corresponding inode in the table. The index of the slot is the same as the index in the table. And this explains the name inode being a short name for *index node*.

Data blocks

As the name suggests, the data blocks are the blocks where the actual data belonging to a file is written. These blocks are also used for different purposes which we will see shortly.

Pointing to data

The inode needs to have a way of pointing to the data blocks that assemble the file. The simplest way is to have **direct pointers**. In this case, each pointer points to a block that has some of the file data. The problem is that large files; where the size exceeds the number of direct pointers an inode can have; are not supported in this mode. One way of overcoming this issue is to use **indirect pointers**, which instead of storing user data they store pointers to blocks that hold user data. For larger files, another layer of indirection is added with **double indirect pointers**. And for even larger files, **triple indirect pointers** are put to use.



Inode multi-level index

To give an idea of the largest file size that each level permit let's run an example considering that each block size is 4 KiB. Research has shown that the majority of files are small [1] so 12 direct pointers would allow for files up to 48 KiB. Considering that each pointer takes 4 bytes, a single indirect pointer would then allow a file to be up to around 4 MiB:

$$(12 + 1024) * 4 \text{ KiB}$$

With the addition of double indirect pointers the size would jump to around 4 GiB:

$$(12 + 1024 + 1024^2) * 4 \text{ KiB}$$

And finally, with triple indirect pointers the files could have a size of around 4 TiB:

$$(12 + 1024 + 1024^2 + 1024^3) * 4 \text{ KiB}$$

This approach might not be very efficient for handling large files. For example, a file of 100 MiB requires the allocation of 25600 blocks. The performance can be severely impacted in case the blocks were fragmented over the disk.

Some file systems use **extents** to help with this situation. In this approach, there is a single pointer and a length to tell that the data starts at the address of the pointer and runs for the given range of blocks. In our example above, describing the same file would use a single extent of size 100 MiB. Multiple extents can be used to support larger files.

Directories

You may have noticed that there isn't a specific structure for directories. The reason behind it is the fact that inodes represent both files and directories. The difference is in what is stored in the corresponding data blocks. Directories are simply a list of all files that it includes. Each entry has the form of `(name, index number)` so when looking up a particular file (or another directory), the system uses the `name` to find the corresponding inode.

Searching for a file can be slow if a directory contains a large number of files. This issue can be mitigated by maintaining the list sorted and using binary search, or instead of representing it as a list, a hash table or a balanced search tree could also be used.

Access paths

Read

When reading from a file, the file system needs to traverse the entire path, visiting each inode along the way until reaching the inode for the desired file. Assuming the user has permission to access the file, the file system consults which blocks are associated with it and then read the solicited data from them.

Write

When writing to a file, the same process has to happen to find the corresponding inode. If a new block is required for the write, the file system has to allocate the block, update the associated information (bitmap and inode), and write to the block. So one write operation requires five I/O operations: one read to the bitmap, one write to mark the new block as occupied, two to read and write to the inode, and one writing the data in the block. This number can increase when creating a new file because now the associated information of the directory also has to be read and written to reflect this new file, and operations to create a new inode.

GotenksFS

As a side project, I decided to write my own file system in Rust as I'm learning the language. Some aspects are inspired by ext4 [2] (and family), and in this section, you will learn more about it. The file system uses FUSE [3], and the disk is represented as a regular file. The block size can be configured as 1 KiB, 2 KiB, or 4 KiB. Files can have a size of up to 4 GiB for block sizes of 4 KiB while the file system could theoretically be up to 16 TiB in size.

`mkfs`

The first step is to create the image itself with the configuration values for the file system. This is achieved via the `mkfs` command:

```
$ ./gotenksfs mkfs disk.img -s "10 GiB" -b 4096
```

After running the command, the image is created with a total size of 10 GiB, and each block in the file system has a size of 4 KiB.

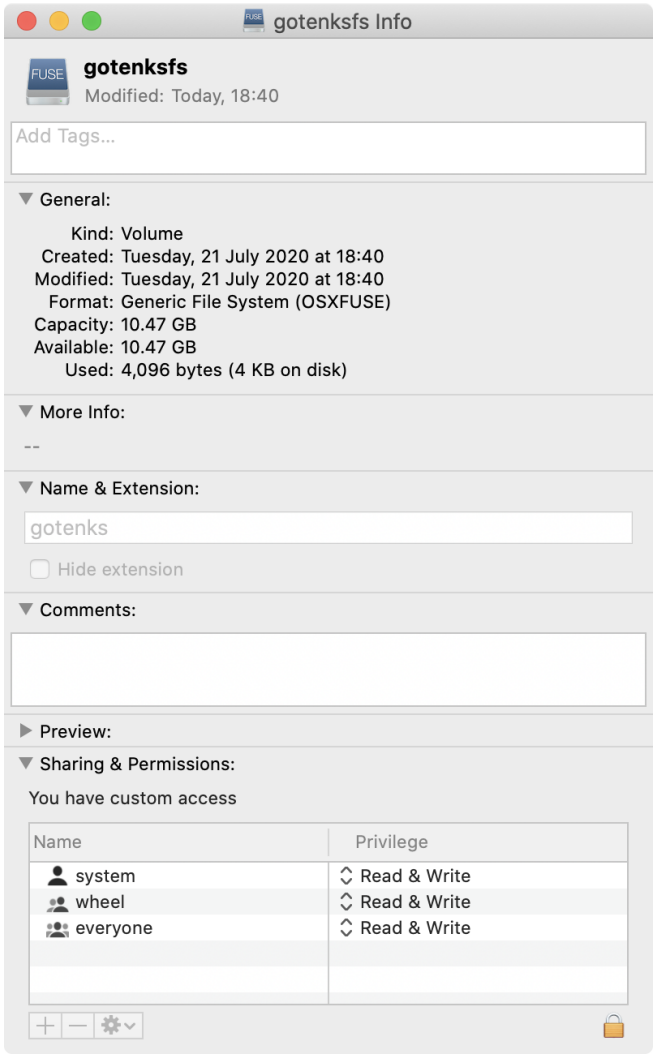
In this step, the configuration values and other structures such as a root directory are written to the image in the first block: the superblock. Its corresponding bitmap entries, and data are also written. These values will be necessary for the next step: mounting the file system.

`mount`

After creating the image, we need to mount it so we can start using it. The `mount` command is used for this:

```
$ ./gotenksfs mount disk.img gotenks
```

And you can see some information about it:



File system after mounting

On-disk structure

The superblock is written in the first 1024 bytes, and it holds the configuration values provided in the `mkfs` command.

```
pub struct Superblock {
    pub magic: u32,
    pub block_size: u32,
    pub created_at: u64,
    pub modified_at: Option<u64>,
    pub last_mounted_at: Option<u64>,
    pub block_count: u32,
    pub inode_count: u32,
    pub free_blocks: u32,
    pub free_inodes: u32,
    pub groups: u32,
    pub data_blocks_per_group: u32,
    pub uid: u32,
    pub gid: u32,
    pub checksum: u32,
}
```

The next two blocks represent the data bitmap and inode bitmap. Then, a run of `n` blocks are used for the inode table. And the blocks following that are the ones where user data will be written.

The inode is defined as follows:

```
pub struct Inode {
    pub mode: libc::mode_t,
    pub hard_links: u16,
    pub user_id: libc::uid_t,
    pub group_id: libc::gid_t,
    pub block_count: u32, // should be in 512 bytes blocks
    pub size: u64,
    pub created_at: u64,
    pub accessed_at: Option<i64>,
    pub modified_at: Option<i64>,
    pub changed_at: Option<i64>,
    pub direct_blocks: [u32; DIRECT_POINTERS as usize],
    pub indirect_block: u32,
    pub double_indirect_block: u32,
    pub checksum: u32,
}
```

As you can see above, inodes support double indirect pointers which means that for a disk with a block size of 4 KiB, the maximum capacity of a file is 4 GiB. The number of direct pointers is set to 12:

```
pub const DIRECT_POINTERS: u64 = 12;
```

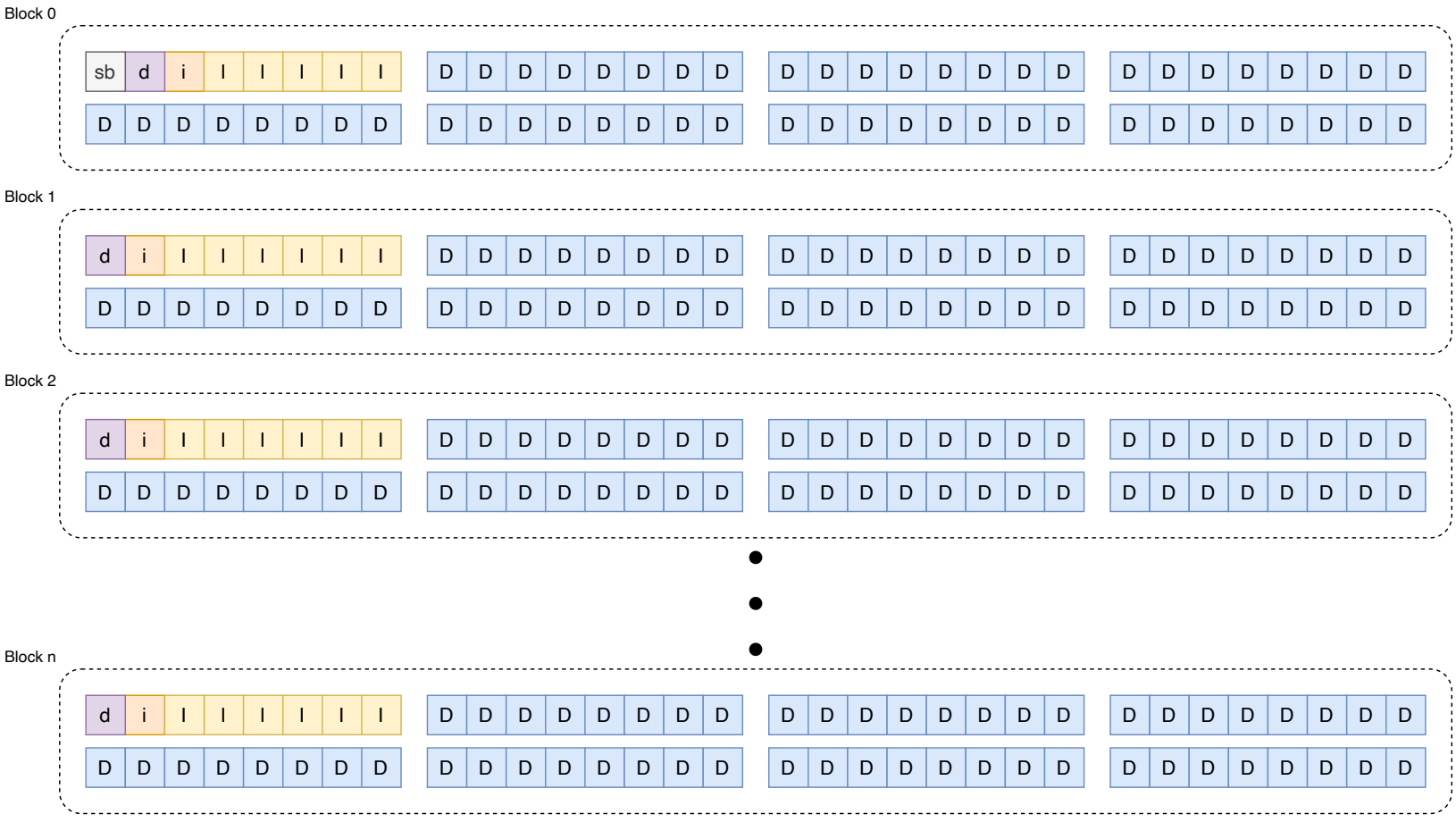
When starting the FS for the first time, it will create the root directory using the definition below:

```
pub struct Directory {
    pub entries: BTreeMap<OsString, u32>,
    checksum: u32,
}
```

Block groups

The inode bitmap has 4 KiB meaning that each bitmap block will have a capacity for 32768 inodes. Suppose we round up the size of an `Inode` to 128 bytes; the corresponding inode table will require 4 MiB of space. One of the ways for structuring them would be to have many blocks dedicated to the bitmaps, then the corresponding number blocks to store the inodes, and the remaining blocks for user data.

Instead of doing that, we can create *block groups* that will always have one block for the data bitmap and one for the inode bitmap. The next 1024 blocks contain the inode table, and following that, 32768 blocks which are used for user data.



Block groups

Reading and writing to files

Now that the “disk” is set up, we can start writing and reading files from it. Creating a new directory using `mkdir`:

Creating a directory

The process that occurs is the following: the system searches for the inode of the root directory, looks up in which data block the contents are being written to, allocates a new inode and data block for the new directory, writes the entry to the root directory, and finally, writes the new directory to its data block.

Writing a new file follows a similar method. The system traverses the given path until the final directory is reached and adds an entry representing the new file.

The write function provides the path, buffer with the data, offset, and a file info struct that may hold a file handle along with extra information about the file:

```
fn write(&mut self, path: &Path, buf: &[u8], offset: u64, file_info: &mut fuse_rs::fs::WriteFileInfo)
```

In this case, instead of traversing the path again to find the inode, the system uses the file handle (the FS previously sets it when creating the file). With the inode in hand, the FS can build the struct by seeking to the exact location where the `Inode` is written:

```
fn inode_offsets(&self, index: u32) -> (u64, u64) {
    let inodes_per_group = self.superblock().data_blocks_per_group as u64;
    let inode_bg = (index as u64 - 1) / inodes_per_group;
    let bitmap_index = (index as u64 - 1) & (inodes_per_group - 1);
    (inode_bg, bitmap_index)
}

fn inode_seek_position(&self, index: u32) -> u64 {
    let (group_index, bitmap_index) = self.inode_offsets(index);
    let block_size = self.superblock().block_size;
    group_index * util::block_group_size(block_size)
        + 2 * block_size as u64
        + bitmap_index * INODE_SIZE
        + SUPERBLOCK_SIZE
}
```

Now that the FS has information about which data blocks are currently allocated to the `Inode`, it can use them to find the exact location where to write the data, in a process similar as shown above. New blocks are first added to the `Inode` direct blocks array, and if the file size exceeds `(12 * BLOCK_SIZE)`, the FS allocates an `indirect_block` that holds the numbers identifying other blocks containing user data. And in the case the file is even bigger requiring more blocks, the system adds an extra layer of indirection using the `double_indirect_block` field which points to more indirect blocks. Reading from a file follows the same method. You can see it running here:

Reading and writing to files

Conclusion

The idea behind a file system is to define a structure for the contents on disk which then allow the system to operate on top of it when creating, writing and reading from files and directories.

We learned some of the concepts used for the on-disk format, how directories and files are represented. One interesting aspect is the usage of indirect pointers to allow storing large files. If you are interested in learning more about file systems, I would recommend reading more about other techniques that are used such as journaling, copy-on-write, log-structured file systems, and also how modern file systems apply these and other techniques. Maybe start learning about the one you are using right now.

The code for GotenksFS can be accessed here: `carlosgaldino/gotenksfs` .

References

- [1] Agrawal, N., Bolosky, W.J., Douceur, J.R. and Lorch, J.R., 2007. ***A five-year study of file-system metadata***. ACM Transactions on Storage (TOS), 3(3), pp.9-es.
- [2] ext4.wiki.kernel.org. 2020. ***Ext4 Disk Layout - Ext4***. Available at: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Overview.
- [3] en.wikipedia.org. 2020. ***Filesystem In Userspace***. Available at: https://en.wikipedia.org/wiki/Filesystem_in_Userspace.