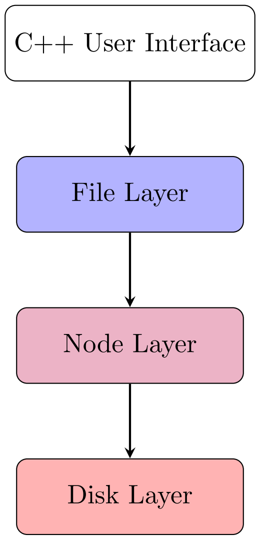
File System Design

1 Structural Overview

1.1 Layered Architecture & Naming Convention

This system is organised into a layered architecture of four main components: The Disk Layer, The Node Layer, The File Layer, and The C++ Interface [1]. Each component depends on the layer below it, and provides a wrapper around its interface. Because of this, the codebase includes many functions which seem to do the same things (`_disk_inode_alloc()`, `_inode_create()`, and `file_create()` for example). In each of these cases, the different functions each provide a portion of the desired effect, before calling the next one down the chain. To avoid confusion, the public interface functions are unmarked `snake_case()` by convention, while private interface functions are preceded by an underscore `_like_this()`.



[1]Read the User Guide

1.2 Persistence & The Disk

In this system, a "disk" file of appropriate format[2] is loaded entirely into memory on system startup. All manipulation of the disk's contents happens on this in-memory copy. Once all operations are completed, the memory is copied back onto the file in its entirety. This is less than ideal for many reasons, but it allows us to not have to think to much about resource management at run-time, among other benefits.

The disk file's contents are, for the most part, split into two parts: Data and Meta-Data (in the form of I-Nodes). These two categories are grouped together into discrete "blocks", the boundaries of which are defined by the disk file's header, and which correlate to the file's size (bigger disk == more blocks == more files).

All structures other than the I-Nodes are strictly non-persistent. They are not saved to the disk, and they only exist to operate on the disk's contents in a more user and application-friendly manner.

2 The Disk Layer

This Section Includes All The Contents Of The Disk File, In Order.

2.1 The Superblock

The file system works on a "disk" file formatted into 4KB "blocks" of data or meta-data [3]. The first of these blocks is always the "superblock", a file header which contains useful formatting data about the disk itself (The number of data blocks, the number of I-Node blocks, etc.). A file without a valid superblock should not be used with this system.

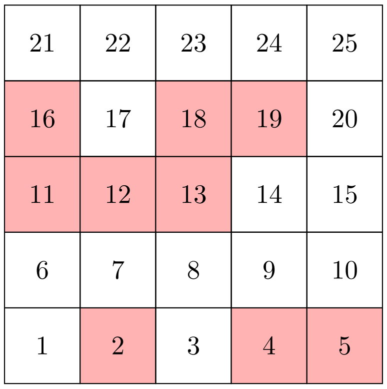
```
// Disk Superblock (File Header)
typedef struct superblock_s {
        uint32_t size;          // Total Number Of Blocks
        uint32_t data_size;     // Number Of Data Blocks
        uint32_t inode_size;    // Number Of Inode Blocks
        uint32_t data_start;    // Data Starting Block
        uint32_t inode_start;   // Inode Starting Block
} superblock_s;
```

You may notice that the superblock is considerably shorter than 4KB in length. This wasted space is unavoidable, since our design needs to be aligned into 4KB blocks.

[2]"data.dat" in the sample.
[3]For more details, see [1]

2.2 The Bit-Maps

The next two 4KB blocks after the superblock are also pre-determined. Blocks 1 & 2 in all valid disk files are the I-Node Bit-map and the Data Bit-Map, respectively. These are the allocation structures used to keep track of which I-Nodes or Data Blocks have already been allocated, and which are free to allocate next time one is requested. Each bit in the map corresponds to a single I-Node or Data Block: (1) means allocated and (0) means free. By updating the map during resource allocation & release, we can easily keep track of our resources.



2.3 The I-Node Blocks

The next few blocks are filled with dnode_s structures. Each of these is a ”disk view” representation of an I-Node in our system [4]. `dnode_s`’s contain only persistent information - the things we need to keep on disk.

```
// On-Disk Inode Structure
typedef struct dnode_s {
        uint8_t  type; // 0 - Free / 1 - I_FILE / 2 - I_DIR
        uint16_t size; // Number Of Blocks
        uint8_t  ref_count;
        uint16_t addr[NDIRECT+1];
} dnode_s;
// DNode Operations
dnode_s*       _disk_inode(const disk_s*,const uint16_t);
bool           _disk_inode_free(const disk_s*,const uint16_t);
const uint16_t _disk_inode_alloc(const disk_s*);
```

The number of I-Node blocks on-disk and the block number of the first I-Node block are both found in the superblock.

[4]See [2] for a better implementation of `dnode_s`’s

2.4 The Data Blocks

The rest of the blocks on the disk are reserved solely for data. These take up the vast majority of disk space, since file contents are usually orders of magnitude larger than the meta-data, even for relatively small files. Like the I-Node blocks, the number and starting block index are read from the superblock on startup. Also like I-nodes, Data Blocks are allocated via a Bit-Map structure. Unlike I-Nodes, Data Blocks have no fixed structure or format, they are simple raw data to be manipulated however the file to which they are allocated requires. For this reason, we can allocate whole blocks at a time to each file that needs them. This means that a file that uses just 4100B would get a whole 2 Blocks. Such internal fragmentation is just another unavoidable downside of our simplistic design.

3 The I-Node Layer

3.1 The I-Node Structure

As mentioned earlier, an I-Node is the underlying structure behind each file or directory in our system. Discussed in the Section 2.3 were the `dnode_s` structures - the "disk view" of an I-Node. However, not all meta-data is stored on the disk. Many crucial bits of information, such as the I-Number, would be totally redundant to store on-disk. It would be a waste of precious space. For this reason, we have the `inode_s` structure- the "application view" or the "logical view" of an I-Node.

```
// Logical Index Node Structure
typedef struct inode_s {
        struct dnode_s* info;
        disk_s*  dev;
        uint16_t inum;
        bool     valid;
} inode_s;
// Inode Operations
inode_s _inode_get(disk_s*,const uint16_t);
inode_s _inode_create(disk_s*,const uint8_t);
bool    _inode_destroy(inode_s*);
```

A few things to note about this structure: Firstly, that it holds a reference to the underlying `dnode_s`. This is made easy by our loading of the disk file into memory. Secondly, that the three operations defined here exactly correspond to the operations on `dnode_s` shown in the aforementioned section 2.3. This is an example of our layered design coming into effect - the next layer will mirror these functions yet again, and the Interface layer will do likewise.

3.2 The Directory Entry Structure

As mentioned numerous times before, directories in out system are simply a special type of file. In such files, the regular read / write interface doesn't apply. This is because the fundamental difference between a regular file and a directory lies in the contents of its data. In a regular file, data is simply a contiguous set of un-formatted binary chunks. Conversely, a directory's data section is formatted in a very simple structure: each entry in the data section of a directory is of type `dirent_s`. A `dirent_s` structure has only two members: the I-Number and the Name[5].

```
// On-Disk Directory Entry Structure
typedef struct dirent_s {
        uint16_t inum;
        char name[DIRENT_NAME_LEN];
} dirent_s;
```

4 The File Layer

4.1 The File Handle Structure

The File Layer follows the same pattern as the previous two: it provides a wrapper around the `_inode_get`, `_inode_create`, and `_inode_destroy` functions. In addition to those, the File Layer also exposes the lowest-level User Interface to the system[6]. Using objects of the `file_s` type with the appropriate permissions, a user can manipulate the underlying data content of a file or directory.

```
// Logical File Descriptor Structure
typedef struct file_s {
        inode_s  node;
        uint32_t iter;
        uint8_t  mode;
        bool     valid;
} file_s;
// Public File Interface
uint16_t dir_lookup (const inode_s*,const char*);
void     dir_print  (const inode_s*);
file_s   file_open  (const inode_s*,const char*,const uint8_t);
bool     file_close (file_s*);
uint16_t file_create(const inode_s*,const char*,const uint8_t type);
bool     file_delete(const inode_s*,const char*);
// Operations On Open File Handles
uint32_t file_write(file_s*,uint8_t*,const size_t);
uint32_t file_read (file_s*,uint8_t*,const size_t);
```

[5]These will be familiar as the outputs of the `FilSystem::ls_dir` method

[6]The C++ Bindings build on top of these functions

```
bool     file_seek (file_s*,const uint32_t);
uint32_t file_tell (const file_s*);
```

5 References

References

[1] Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Chapter 40

[2] https://github.com/mit-pdos/xv6-public

[3] https://en.wikipedia.org/wiki/Extent_(file_systems)

[4] https://fkohlgrueber.github.io/blog/tree-structure-of-file-systems/