CSCI 3150 Introduction to Operating Systems
Project: Implementation of SFS
Deadline: 23:59, April 10, 2022

In this project, you are required to implement a simple file system called ***SFS***. An overview of SFS is shown in Figure 1.

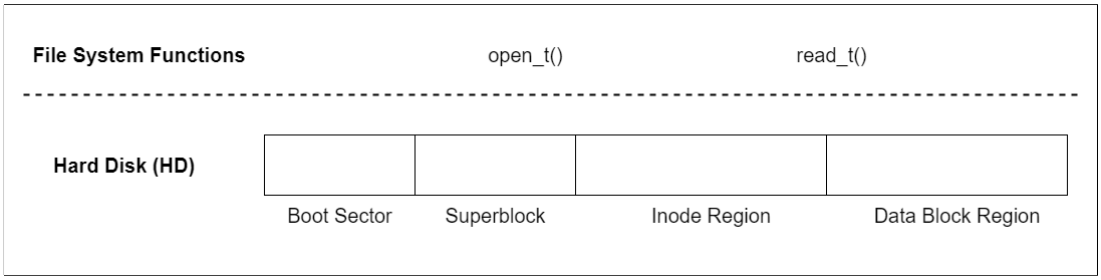

Fig.1 SFS architecture

1. SFS (Simple File System)

SFS works on a file called **HD** that is a 110MB file (initialized properly) and you can find it from the zip file provided.

What you are required to do is implementing two functions **open_t()** and **read_t()**.

- These two file-system-related functions are based on the simple file system. An illustrative format on HD is shown in Figure 2.

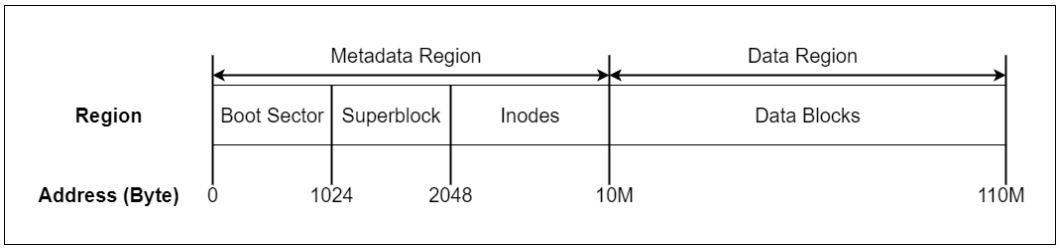

Fig.2 SFS regions and addresses

As shown in Figure 2, in HD, there are two regions: the metadata region and data region. The metadata region is inside the first 10MB, it contains a boot sector (the first 1024 bytes), the superblock and inodes. The superblock region is from 1024 bytes to 2048 bytes, and the inode region is from 2048 bytes to 10 MB. The data region is from 10 MB to 110 MB, in which it is divided into data blocks (each data block is 4 KB).

The superblock region defines the layout and its format can be found from the following structure:

```
typedef struct _super_block_
{
    int inode_offset; /* The starting position of inodes */
    int data_block_offset; /* The starting position of data blocks */
    int max_inode; /* The maximum number of inodes */
```

```
    int max_data_block; /* The maximum number of data blocks */
    int next_available_inode; /* The index of next free inode */
    int next_available_data_block; /* The index of next free data block */
    int block_size; /* The size of a data block */
} superblock;
```

Basically, the inode region starts at 2048 bytes (*inode_offset*); the data region starts at 10 MB (*data_block_offset*); the block size (*block_size*) is 4 KB.

The inode region contains inodes that can be retrieved based on its index in the inode region (called the *inode_number*). An inode is used to represent a file, and is defined based on the following structure:

```
typedef struct _inode_ /* The structure of inode, each file has only one inode */
{
    int inode_number; /* The inode number */
    time_t creation_time; /* Creation time of the file*/
    int file_type; /* 0 for regular file, 1 for directory*/
    int file_size; /* The size of the file (bytes)*/
    int block_number; /* The number of data blocks occupied by this file*/
    int direct_block[2]; /*Two direct data block pointers */
    int indirect_block; /*One indirect data block pointer */
    int sub_file_number; /* Number of files under a directory (0 for regular file)*/
} inode;
```

Some related parameters can be found as follows:

```
#define SUPER_BLOCK_OFFSET 1024 /* The offset of the superblock*/
#define INODE_OFFSET 2048 /* The offset of the inode region */
#define DATA_BLOCK_OFFSET 10485760 /* The offset of the data region */
#define BLOCK_SIZE 4096 /* The bytes per data block */
```

In SFS, an inode contains two direct data block pointers and one indirect data block pointer. There are two types of files: regular and directory files. The content of a directory file follows the structure below:

```
typedef struct directory_mapping /* Record file information in directory file */
{
    char file_name[20]; /* The file name of the file */
    int inode_number; /* The inode number of the file*/
} DIR_NODE;
```

Each directory file should at least contain two mapping items, “.” and “..”, for itself and its parent directory, respectively.

Based on SFS, the prototypes of the two filesystem-related functions are shown as follows:

1) **int open_t(char *pathname);**
   Description: Given an absolute *pathname* for a file, open_t() returns the corresponding inode number of the file or -1 if an error occurs. The returned inode number will be used in subsequent functions in read_t().
2) **int read_t(int inode_number, int offest, void *buf, int count);**
   Description: read_t() attempts to read up to *count* bytes from the file starting at *offset* (with the inode number *inode_number*) into the buffer starting at *buf*. It commences at the file offset specified by *offset*. If *offset* is at or past the end of file, no bytes are read, and read_t() returns zero. On success, the number of bytes read is returned (zero indicates end of file), and on error, -1 is returned.

2. Requirements

In this project, you need to implement **open_t()** and **read_t()**.

After unzipping this zip file, you can find the following files:

- *call.c*: **The source code for open_t() and read_t() that you should implement**. **In call.c, you are allowed to create any auxiliary functions that can help your implementation. But only "open_t()" and "read_t()" are allowed to call these auxiliary functions.**
- *call.h, inode.h , superblock.h*: The header files that define the data structures and function prototypes.
- *HD*: The hard disk file, which has been initialized properly (110 MB);

Your programs **must be able to be compiled/run under the XUbuntu environment (in Lab One)**.

What to submit – A zip file that ONLY contains call.c.

Noted:

1. **Your programs must be compiled under XUbuntu! Other VM, Windows or MAC may incur incompatible issues.**
2. **The grading scheme is different from the test cases! Test cases are used for self-checking only.**
3. **The test data can be found in /test-cases/Plato.txt. Because it's quite long, you can use: ./read_test >> test_result.txt, to save the result.**