CSCI 3150 Introduction to Operating Systems
Project: Implementation of SFS
Deadline: 23:59, December 5, 2021

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

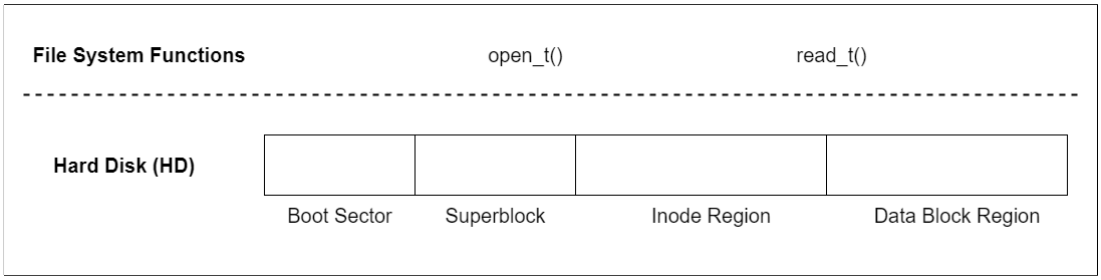

Fig.1 SFS architecture

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.

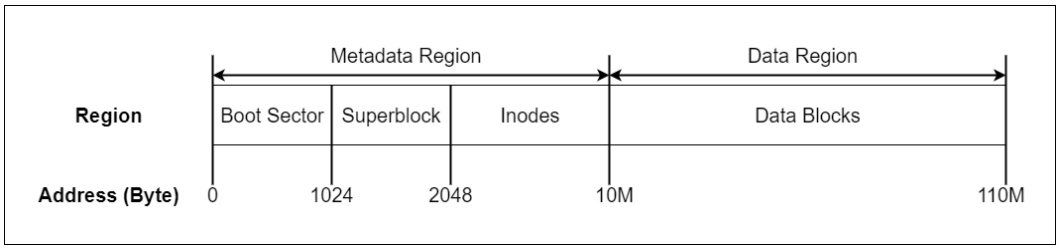

Fig.2 SFS regions and addresses

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_/*The key information of filesystem */
{
        int i_offset;
        int d_offset;
        int max_inode;
```

```
    int max_data_blk;
    int next_available_inode;
    int next_available_blk;
    int blk_size;
}superblock;
```

Basically, the inode region starts at 2048 bytes (*i_offset*); the data region starts at 10 MB (*d_offset*); the block size (*blk_size*) is 4 KB.

The inode region contains inodes that can be retrieved based on its index in the inode region (called the *i_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 i_number; /* The inode number */
    time_t c_time; /* Creation time of inode*/
    int f_type; /* 0 for regular file, 1 for directory file */
    int f_size; /* The size of file */
    int blk_num; /* The total number of data blocks */
    int direct_blk[2]; /*Two direct data block pointers */
    int indirect_blk; /*One indirect data block pointer */
    int f_num; /* Number of files under a directory (0 for regular file)*/
}inode;
```

Some related parameters can be found as follows:

```
#define SB_OFFSET      1024          /* The offset of superblock region*/
#define I_OFFSET       2048          /* The offset of inode region */
#define D_OFFSET       10485760      /* The offset of data region */
#define BLK_SIZE       4096          /* The size per 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 should follow the following structure:

```
typedef  struct  dir_mapping   /* Record file information in directory file */
{
    char  dir_name[20];          /* The file name in current directory */
    int   i_number;     /* The corresponding inode number */
}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 i_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 *i_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);

This project will be graded by Mr. HUANG Kecheng (Email: kchuang21@cse.cuhk.edu.hk). 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 case! Test cases are used for self-checking only.**