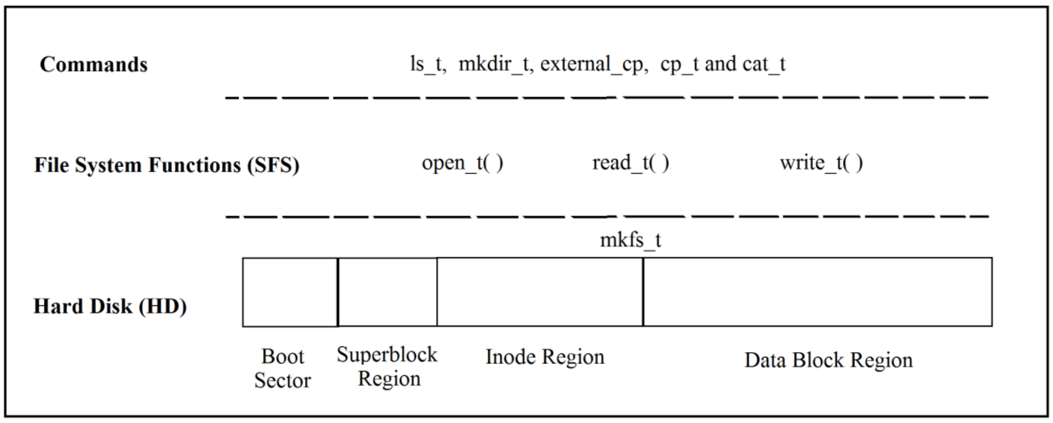
CSCI 3150 Introduction to Operating Systems
Assignment Three
Deadline: 23:55, 9 December 2018

In this assignment, you are required to implement a simple file system called ***SFS***. After you finish this homework, you should have better understanding of a file system and its organization/implementation. An overview of SFS is shown in the figure below.

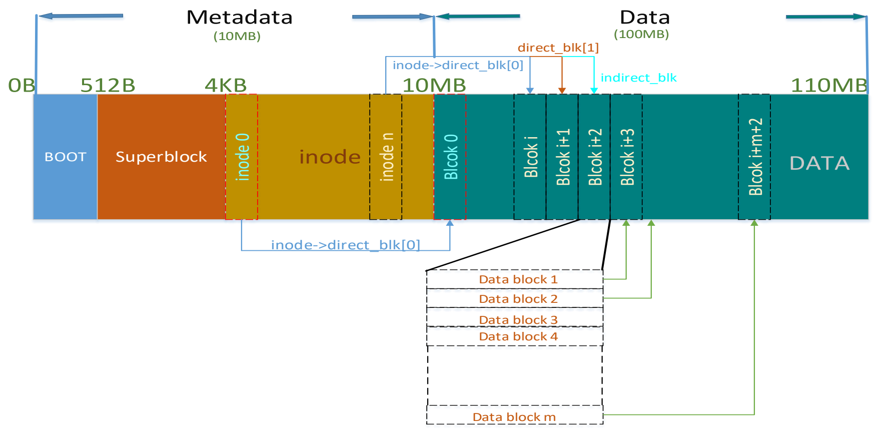


1. SFS (Simple File System)

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

The implementation of ***SFS*** consists two parts: three filesystem related functions (open_t(), read_t(), and write_t()), and six commands (mkfs_t, ls_t, mkdir_t, external_cp, cp_t, cat_t).

- Three file-system-related functions are based on the simple file system. An illustrative format on HD is shown below:



As shown above, in HD, there are two regions: the metadata and data regions. The metadata region is inside the first 10MB; it contains a boot sector (the first 512 bytes), the superblock and inode regions. The superblock region is from 512B to 4KB, and the inode region from 4KB to 10MB.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:

```
struct    superblock           /*The key information of filesystem */
{
        int   inode_offset;         /* The start offset of the inode region */
        int   data_offset;          /* The start offset of the data region */
        int   max_inode;            /* The maximum number of inodes */
        int   max_data_blk;         /* The maximum number of data blocks */
        int   next_available_inode; /* The index of the next free inode */
        int   next_available_blk;   /* The index of the next free block*/
        int   blk_size;             /* The size per block */
};
```

Basically, the inode region starts at 4 KB (inode_offset); the data region starts at 10 MB (data_offset), the maximum number of inodes is 100 (max_inode); the maximum number of data blocks is 25600; next_available_inode and next_available_blk are used to represent the indexes of the next free inode and the next free block, respectively; the block size is 4 KB. To make it simple, you do not need to reclaim inodes or data blocks, and **you can simply obtain the next available inode (data block) index based on next_available_inode (next_available_blk) when you create a file (allocate data blocks).**

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:

```
struct inode                /* The structure of inode, each file has only one inode */
{
        int   i_number;        /* The inode number */
        time_t   i_mtime;      /* Creation time of inode*/
        int   i_type;       /* Regular file for 0, directory file for 1 */
        int   i_size;          /* The size of file */
        int   i_blocks;        /* The total numbers of data blocks    */
        int   direct_blk[2];   /*Two direct data block pointers    */
        int   indirect_blk;    /*One indirect data block pointer */
        int   file_num;        /* Number of files under a directory (0 for regular file)*/
};
```

Some related parameters can be found as follows:

```
#define SB_OFFSET      512            /* The offset of superblock region*/
#define INODE_OFFSET       4096   /* The offset of inode region */
#define DATA_OFFSET      10485760 /* The offset of data region */
#define MAX_INODE          100   /* The maximum number of inode */
#define MAX_DATA_BLK      25600  /* The maximum number of block */
#define BLOCK_SIZE     4096                    /* The size per block */
#define MAX_NESTING_DIR 10                /* The nesting number of directory */
#define MAX_COMMAND_LENGTH    50 /* The maximum command length */
```

In SFS, an inode contains two direct data block pointers and one single 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[20];          /* The file name in current directory */
        int    inode_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 three filesystem-related functions are shown as follows:

1) int open_t(const char *pathname, int flags);

   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() and write_t().

   The argument *flags* can be one of the following three values: 0 (or 1) means that a new regular (or directory) file will be created (if one file with the same name exists, the new file will replace the old file); 2 means that the target is an existing file. The new inode number can be obtained from *next_available_inode* from the superblock; after it is obtained, *next_available_inode* should be incremented by one.

2) int read_t(int inode_number, int offset, 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.

3) int write_t(int inode_number, int offset, void *buf, int count);

Description: write_t() writes up to *count* bytes from the buffer pointed *buf* to the file referred to by the inode number *inode_number* starting at the file offset at *offset*. The number of bytes written may be less than *count* if there is insufficient space on the underlying physical medium or the maximum size of a file has been achieved. On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned.

A new block can be obtained from *next_available_blk*; after it is obtained, *next_available_blk* should be incremented by one.

- Six commands: mkfs_t, ls_t, mkdir_t, external_cp, cp_t, and cat_t. Based on the above, six commands are used to support SFS. Among them, mkfs_t directly works on HD, and the other commands (ls_t, mkdir_t, external_cp, cp_t, and cat_t) work with SFS based on the above three functions (open_t(), read_(), and write()). They are described as follows:
  1) mkfs_t

     Description: mkfs is used to build an SFX filesystem on HD. **This should be the first step in order to use our SFS filesystem on HD**. After this command is successfully executed, the parameters in the superblock region discussed above will be set up correspondingly in HD. Moreover, the first inode (whose inode number is 0) is created for the root directory, and its initial data (related to "." and "..") will be written into Block 0 based on DIR_NODE. See Lab 9 for details.
  2) ls_t *dname*

     Description: ls_t lists the information of all files under the directory *dname* and *dname* should be an absolute pathname. The information related to each file should include its inode number, creation time, file type (regular or directory), and the size of the file.
  3) mkdir_t dname

     Description: mkdir_t creates a new directory file with the name *dname* and *dname* includes the new directory name with the absolute path. The new directory file will be created even if this is a directory file with the same name (i.e. the new directory file will replace the old one under the directory).
  4) external_cp outside_path_name sfs_path_name

     Description: external_cp copies a regular file from Linux (specified by *out_side_path_name* that is the absolute path) to a file (with sfs_path_name as the absolute path and name) inside the SFS filesystem. A new regular file will be created and copied in the SFS if the path/names specified by outside_path_name and sfs_path_name are effective (the new regular file will be created and copied in the SFS even if there is a regular file with the same path/name *sfs_path_name)*; otherwise, the error will be reported.
  5) cp_t source_path_name destination_path_name

Description: cp_t copies a regular file (specified by *source_path_name*that is the absolute path) to the destination (specified by *destination_path_name* that is the absolute path). A new regular file will be created and copied in the SFS if the path/names specified by *source_path_name* and *destination_path_name*are effective (an old file with the same path/name as *destination_path_name* will be replaced by the new file); otherwise, the error will be reported.

6) ca_t path_name

Description: cat_t prints the contents of the file specified by the absolute path/name *file_name* to the standard output. If the file does not exist, the error will be reported.

2. Requirements

In this assignment, you need to implement open_t(), read_t() and write_t(), thereby making SFS work with the above five commands (ls_t, mkdir_t, external_cp, cp_t, and cat_t). Note that the source code for mkfs_t can be found in Lab 9, and the source code for the other five commands has been provided but open_t(), read_t() and write_t() have not been implemented.

To start, please download assign3-code.zip. After unzipping this zip file, you can find the following files:

- *call.c*: **The source code for open_t(), read_t() and write_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()", "wrtie_t()", and "read_t()" are allowed to call these auxiliary functions.**
- *cat_t.c, cp_t.c, external_cp.c, ls_t.c, mkdir_t.c*: The source code for the five commands mentioned above (cat_t, cp_t, external_cp, ls_t, and mkdir_t) that have been implemented.
- *call.h, inode.h , superblock.h*: The header files that define the data structures and function prototypes.
- *HD*: The hard disk file (110 MB);
- *Makefile*: The makefile file.

You can compile and generate all commands by inputting:

```
make
```

"make" will work based on "Makefile". Note that HD is empty in the beginning; so the first step is to format HD with "mkfs_t" (see Lab 9 for details).

This assignment will be graded by Mr. Lei Zhu (Email: leizhu@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.