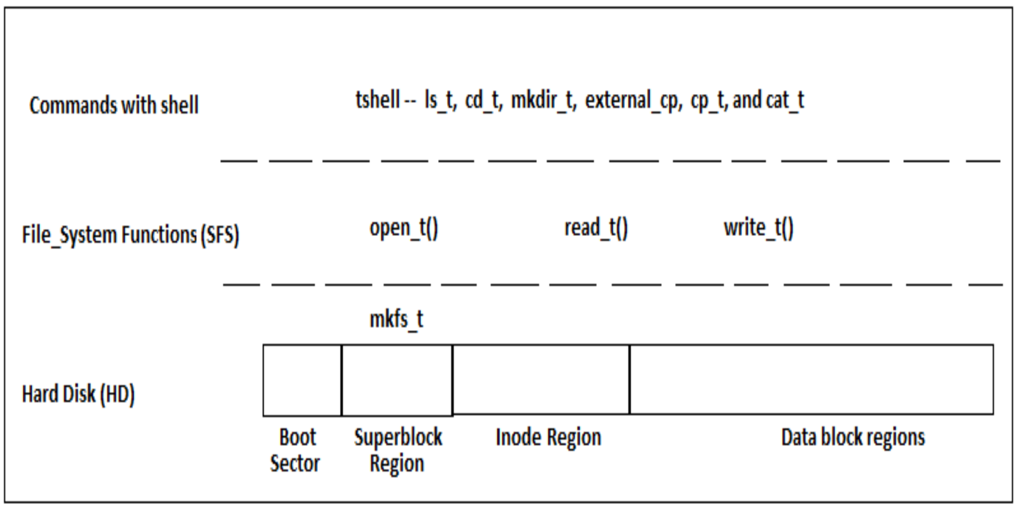
COMP 3438 SYSTEM PROGRAMMING

Assignment One

Deadline: 23:55, 5 March, 2017

In this assignment, you are required to implement a simple filesystem called ***SFS*** and a simple shell program called ***tshell***. After you finish this homework, you should have a better understanding of a file system and its organization/implementation. Moreover, you can practice what you have learnt about processes (how to generate a new process, parent waits for child, etc.) when implementing ***tshell***. An overview of the SFS and tshell is shown in the figure below.



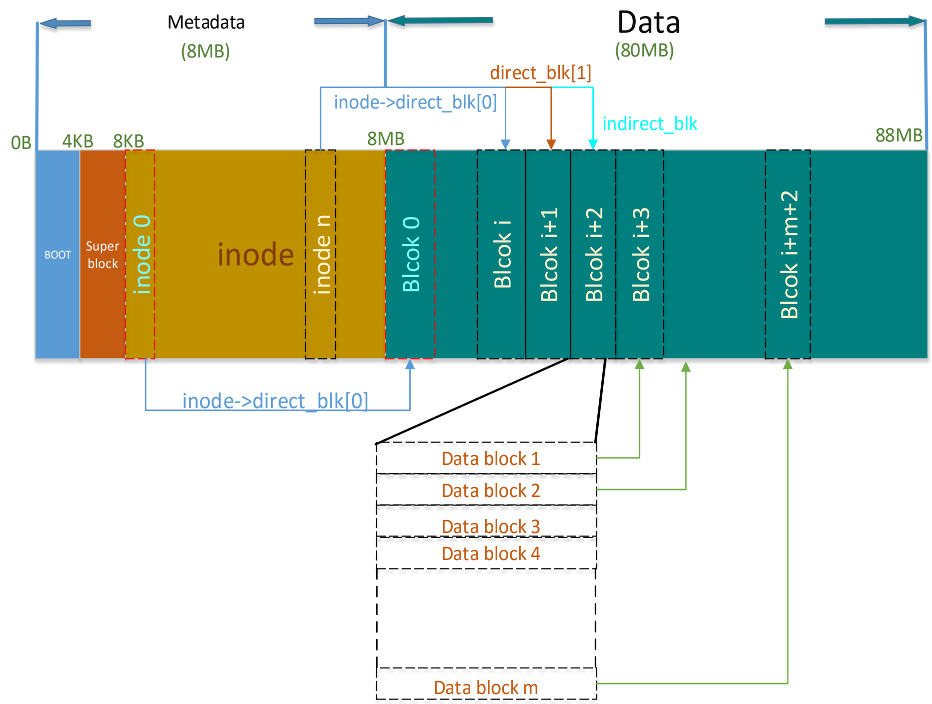
1. SFS (Simple File System)

SFS works on a file called **HD** that is a 110MB file (initially empty) and you can download from the Blackboard.

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

(1) Three filesystem-related functions.

Three functions are based on the simple filesystem with the following format on HD:



As shown above, in HD, there are two regions: the metadata and data regions. The metadata region is inside the first 8 MB; it contains a boot sector (the first 4096 bytes), the superblock and inode regions. The superblock region is from 4 KB to 8 KB, and the inode region from 8 KB to 8 MB. The data region is from 8 MB to 88 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 8 KB (inode_offset); the data region starts at 8 MB (data_offset), the maximum number of inodes is 170 (max_inode); the maximum number of data blocks is 20480; 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 1, directory file for 0 */
    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;       /* The number of file in directory, it is 0 if it is file*/
}
```

Some related parameters can be found as follows:

```
#define SB_OFFSET            4096        /* The offset of superblock region*/
#define INODE_OFFSET         8192   /* The offset of inode region */
#define DATA_OFFSET       8388608 /* The offset of data region */
#define MAX_INODE           170        /* The maximum number of inodes */
#define MAX_DATA_BLK      20480    /* The maximum number of blocks */
#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 pointer 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[10];            /* The file name in current directory */
    int inode_number;      /* The corresponding inode number */
}DIR_NODE;
```

Each directory file (***except the root directory***) 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.
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.

(2) Seven commands: mkfs_t, ls_t, cd_t, mkdir_t, external_cp, cp_t, and cat_t.

Based on the above, seven commands need to be implemented to support SFS. Among them, mkfs_t directly works under Linux, and the other six commands (ls_t, cd_t, mkdir_t, external_cp, cp_t, and cat_t) work with SFS under ***tshell*** and should be implemented based on the above three functions (open_t(), read_(), and write()). They are described as follows:

1) mkfs_t file_name
   Description: mkfs is used to build an SFX filesystem on a file with the name *file_name*.
   This should be the first step in order to use our SFS filesystem on a file, and the command should be executed in Linux. After this command is successfully executed,

the parameters in the superblock region discussed above should be set up correspondingly in the file.

2) ls_t

Description: ls_t lists the information of all files under the current working directory in ***tshell***. For each file, the information should include its inode number, creation time, file type (regular or directory), and the size of the file.

3) cd_t path_name

Description: cd_t changes the current working directory of ***tshell*** to the one specified with *path_name* (absolute path). It will report the error and keep the current working directory if the directory with *path_name* does not exist.

4) mkdir_t dname

Description: mkdir_t creates a new directory file with the name *dname* under the current working directory of ***tshell***. The new directory file will be created even if this is a directory file with the name *dname* (i.e. the new directory file will replace the old one under the working directory).

5) 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 under ***tshell***. 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.

6) 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) under ***tshell***. 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.

7) cat_t path_name

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

2. tshell

In order to make SFS work with the above six commands (ls_t, cd_t, mkdir_t, external_cp, cp_t, and cat_t), we need to implement a simple shell program called tshell. Basically, in *tshell*, it needs to maintain the current working directory and the root directory (the first inode with the inode number 0 is set up as the root directory file by default, *and* ***if it is not existent when tshell starts, it should be created***), and run the six commands in a parent/child mode. That is, when tshell is executing, it should print "tshell###" and wait for user input; to run one command, it will wait until the command has been finished.

3. Test

At least, you should make the following commands work with your implementation.

(1) The command directly works under Linux:

mkfs_t HD

(2) The commands work under **tshell**:

mkdir_t test1

mkdir_t test2

ls_t

cd_t /test1

mkdir_t test3

cd_t /test1/test3

external_cp /media/e.txt /test1/test3/e.txt

ls_t

external_cp /media/re.txt /test1/test3/re.txt

cat_t /test1/test3/re.txt

cp_t /test1/test3/e.txt /test1/e.txt

cat_t /test1/e.txt

(e.txt and re.txt can be any files with the sizes of 4KB+10B and 4MB+3KB, respectively)

What you need to submit – A zip file contains the following:

1. **The source code, and among them, the C files should be named as mkfs.c and tshell.c.**
2. **A readme file (.txt) to describe how to compile and run your programs.**