Project 2 RSFS: A Ridiculously Simple File System

(Total: 80 points + bonus)

1. Introduction

In this project, you will develop a highly simplified in-memory file system, named **RSFS** – Ridiculously Simple File System.

To facilitate your development, the major data structures and the code for many low-level operations have been provided. Your main task is to implement the file system API based on them and develop your test code to test the API. Three API functions, including RSFS_init(), RSFS_create() and RSFS_stat(), have also been provided as examples. Sample testing codes are provided as well. However, you are strongly encouraged to develop and evaluate your system with more testing code on your own, as we may use more testing code in grading.

We will evaluate your system in two levels:

- As the basic level of evaluation, which is **mandatory**, your system is expected to work correctly when no file is concurrently accessed. That is, at a time a file can only be opened by at most one thread.
- As the advanced level of evaluation, which is **optional** and resulting in **bonus** credits, your system is also expected to work correctly when a file may be concurrently request for access by multiple readers (who want to open the file in the Ready-Only, i.e., RSFS_RDONLY, mode) or potential writers (who want to open the file in the Read-Write, i.e., RSFS_RDWR, mode).

You can work on the project in pairs (i.e., groups of two students) or individually.

You are expected to read through and understand the whole document before working on the required items. Like in Project 1.C, the required items are part of a bigger system that cannot be directly addressed without understanding the bigger picture.

Evaluation Level	Task	Points (for Pair)	Points (for Individual)
Mandatory (the functions are expected to work correctly when no file is concurrently accessed)	RSFS_open()	15	15+2
	RSFS_write()	15	15+2
	RSFS_read()	15	15+2
	RSFS_fseek()	10	10+1
	RSFS_close()	5	5+1
	RSFS_delete()	15	15+2
	Documentation	5	5
Optional (Bonus opportunity)	A file can be accessed by multiple readers concurrently, or by a writer mutually-exclusively, at a time	10	10

2. Implementation Guide

The project expects you to be familiar with file system interface and file system implementation (L21 and L22 of the class lectures).

2.1 Provided code package

The data structure, low-level code, sample API code, and sample test code are attached as a zipped package. In your working environment (pyrite is recommended), which must run a Linux system, unzip the package to a directory named RSFS.

$unzip project-2.zip

The directory RSFS has the following content:

- Makefile: After you complete the system, by "make" to compile the system and get executable application named "app"; by "make clean" to clean up all but the source code.
- def.h: definitions of global constants, main data structures and API functions.
- dir.c: declaration of root_dir (root-level directory) and implementation of low-level code to search, insert and delete directory entry in the root directory.
- inode.c: declaration of inodes, inode bitmap, and their guarding mutex; the low-level code to allocate and free inode.
- open_file_table.c: declaration of open_file_table and its guarding mutex; the low-level code to allocate and free open file entry.
- data_block.c: declaration of data_blocks, data bitmap and its guarding mutex; the low-level code to allocate and free data block.
- api.c: implementation of basic functions provided by a typical file system.
- application.c: application (testing) code that calls the API to create, delete, open, close, read, write, reposition files.
- sample_output.txt: sample outputs of runing the provided application (testing) code.

2.2 Main data structures

Recall from our in-class discussion of file system implementation, the main data structures include inodes, data blocks, bitmaps for inodes and data blocks, and directories. For a real file system, these data structures are scattered to disks and main memory. In this project, however, all these data structures are implemented in the main memory for simplicity (and so it is ridiculously simple).

2.2.1 Data blocks, data block bitmap, and mutex

In this project, each data block is allocated from main memory (heap) and its size is specified by constant BLOCK_SIZE. The pointers to all the blocks are recorded in array:

- void *data_blocks[NUM_DBLOCKS];

The data block bitmap that tracks the allocation of data blocks is declared as array:

- int data_bitmap[NUM_DBLOCKS];

Note that, we use an integer instead of a bit to indicate the status of a block. Also, to assure mutually-exclusive access of data_bitmap, data_bitmap_mutex is declared.

Two basic operations are provided to manage the data blocks:

- int allocate_data_block()
- void free_data_block(int block_num)

Read file data_block.c for more descriptions of these functions.

2.2.2 Inode, inode bitmap, and their mutexes

An inode is defined as "struct inode" with two elements:

- int block[NUM_POINTER];
- int length;

Here, array block tracks a fixed number (specified as NUM_POINTER) of data block numbers; hence we implement only a direct indexing approach. The field length tracks the length of a file in the unit of byte. The whole space for storing a number (specified as NUM_INODES) of inodes is declared as array:

- struct inode inodes[NUM_INODES];

Array inode_bitmap[NUM_INODES] is used as bitmap for tracking the usage of inodes, and mutexes inodes_mutex and inode_bitmap_mutex are used to guard mutually-exclusive access to the inodes and inode_bitmap arrays, respectively.

Two basic operations are provided to manage the space for inodes:

- int allocate_inode()
- void free_inode(int inode_number)

Read file inode.c for more descriptions of these functions.

2.2.3 Directory entry, root directory, and mutexes

The directory entry for each file is declared as "struct dir_entry", which records two pieces of information for the file: name (i.e., file name) and inode_number (where is the inode of the file).

The root directory (defined as struct root_dir) is organized as a linked list of directory entries. Hence, each entry has pointers to its prev and next, and struct root_dir has pointers to the head and the tail. The root_dir also has a mutex to guard mutually-exclusive access to the root directory.

In file dir.c, global variables are declared based on the above data structure definitions. Three operations for directory management have been provided:

- struct dir_entry *search_dir(char *file_name)

- struct dir_entry *insert_dir(char *file_name)
- int delete_dir(char *file_name)

Read file dir.c for more descriptions of these functions.

2.2.4 Open file entry, open file table, and mutexes

The open file table (open_file_table) is implemented as an array of open file entries. Each open file entry (struct open_file_entry) has the following fields:

- "int used" - indicates if this entry is already used or not (note that there is no bitmap for open file entries)
- "struct dir_entry *dir_entry" - pointer to the directory entry of this file
- "int access_flag" - it takes the value of RSFS_RDONLY or RSFS_RDWR indicating the the file is opened for read-only or read-write
- "int position" - the current position for read/write the file
- "pthread_mutex_t entry_mutex" - mutex to assure mutually-exclusive access to this entry

In addition, the open file table has its mutex (I.e., open_file_table_mutex) to assure the mutually-exclusive access to the table for entry allocation and freeing.

Two operations for open file entry and table have been provided:

- int allocate_open_file_entry(int access_flag, struct dir_entry *dir_entry)
- void free_open_file_entry(int fd)

Read file open_file_table.c for more descriptions of these functions.

2.3 API functions (Mandatory)

In this project, RSFS should provide the following API functions. Some of them have been provided to you, and others should be implemented by you.

2.3.1 RSFS_init() - initialize the RSFS system

This provided function initializes the data blocks, the bitmaps for data blocks and inodes, the open file table, and the root directory.

2.3.2 RSFS_create(char *file_name) - create a file with the given name

The provided function works mainly as follows:

- Search the root directory for the directory entry that matches the provided file name. If such entry exists, the function returns with –1; otherwise, the procedure continues in the following.

- Call insert_dir() to construct and insert a new directory entry with the given file name.
- Call allocate_inode() to get a free and initialized inode
- Recode the inode number to the directory entry.

2.3.3 RSFS_open(char *file_name, int access_flag) - open a file of the given file name with the given access flag.

This to-be-implemented function should accomplish the following:

- Test the sanity of provided arguments
- Find the directory entry that matches the provided file name.
- Find an un-used open file entry to use, and have it initialized with the afore-obtained directory entry and the provided access_flag.
- Return the index of the open file entry in the open file table, as the file descriptor (fd).

The comments on file api.c include the suggested steps for implementation.

2.3.4 RSFS_write(int fd, void *buf, int size) - write size bytes of data from the buffer pointed to by buf, to the file with file descriptor fd, from the current position of the file.

According to the more detailed suggested steps in the file api.c, this to-be-implemented function should accomplish the following:

- Perform sanity test of the provided arguments.
- Get the open file entry of fd, and then get the corresponding directory entry and inode.
- Based on the inode and the position maintained in the open file entry, data are copied from the buffer to the file's data block(s) from its current position. During the course, new data blocks may be allocated for the file to contain the data.
- The number of bytes actually written to the file should be returned.

2.3.5 RSFS_read(int fd, void *buf, int size) - read size bytes of data from the file with file descriptor fd, starting at its current position, to the buffer pointed to by buf. Less than size bytes may be read if the file has less than size bytes from its current position to its end.

The comments on file api.c contain the detailed suggested steps.

2.3.6 RSFS_fseek(int fd, int offset, int whence) - change the position of the file with file descriptor fd, based on arguments offset and whence:

- If whence == RSFS_SEEK_SET, the position is set to offset.
- If whence == RSFS_SEEK_CUR, the position is set to current position plus offset.
- If whence == RSFS_SEEK_END, the position is set to END position plus offset.

The new position should be within the file.

This function should return the new position of the file.

2.3.7 RSFS_close(int fd) - close the file with file descriptor fd.

As provided by the comments in api.c, the function should check the sanity of the arguments, and free the open file entry.

2.3.8 RSFS_delete(char *file_name) - delete the file with provided file name

As provided by the comments in api.c, if there exists a file with the provided file name, the function should free the data block, inode and directory entry of the file with the provided file name.

2.3.9 RSFS_stat() - display the current state of the file system

This provided function can be used for debugging. It displays the list of files in the system, including each one's name, length, and inode number. It also displays the current usage of data blocks, inodes and open file entries.

2.4 API functions (Optional)

To earn bonus credits, you are expected to enhance the above API functions, to support concurrent reading and mutually exclusive writing of each file. Specifically:

- After a file has been opened with RSFS_RDONLY and before it is closed, the file should be allowed to be opened with RSFS_RDONLY again for any times, but it should not be allowed to be opened with RSFS_RDWR.
- After a file has been opened with RSFS_RDWR and before it is closed, the file should not be allowed to be opened with either RSFS_RDONLY or RSFS_RDWR.

As a hint, you may need to make some additions to the provided data structures as well as make additions/changes to the above API functions of RSFS_open() and RSFS_close(). Reviewing the solutions to the readers/writers problem discussed in class will help.

2.5 Test code

File application.c provides sample code to test the mandatory functionality, in test_basic(), and the optional functionality, in test_concurrency(), which is commented off for now. You are encouraged to develop more tests on your own.

3. Documentation

Documentation is required for the project. Every location that you add/modify code must have a comment explaining the purpose of the change.

Include a README file with your name(s), a brief description, and a list of files added or modified in the project.

4. Submission

Make sure your code compiles (with “make”) on pyrite, even you may develop your code in other environments. We will look at the code for partial credit. Document anything that is incomplete in the README file.

Submit a zip file of the RSFS directory. On the linux command line, the zip file can be created using:

```
$zip –r project-2.zip RSFS
```

Submit project-2.zip.