Project 2 RSFS: Implementing A Ridiculously Simple In-Memory File System

(Total: 60 points + bonus)

1. Introduction

In this project, you will develop a highly simplified in-memory file system, named **RSFS** – Ridiculously Simple File System. Note: this project will be developed in a regular **Linux** system with regular **C or C++** language.

To facilitate your development, the major data structures (i.e., inodes, bitmaps, data blocks, and directory entries) and their implementations have been provided. Your main task is to implement the file system API based on them. Four API functions, including RSFS_init(), RSFS_create(), RSFS_delete() and RSFS_stat(), have also been provided as examples. Sample testing codes and the expected outputs are provided as well.

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

We will evaluate your system in two levels:

- As the basic level of evaluation, which is **mandatory**, you are expected to implement the API functions open, append, read, seek, write, and close, under the condition that each file is accessed by only one single thread at a time.
- As the **advanced** level of evaluation, which is **mandatory for pairs** but is optional (resulting in **bonus** credits) for individuals, your API functions should work correctly when a file is concurrently requested for access by multiple threads. These threads can be readers (who want to open the file in the Ready-Only, i.e., RSFS_RDONLY, mode) or writers (who want to open the file in the Read-Write, i.e., RSFS_RDWR, mode).

Please 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 need to be developed based on the understanding of the bigger picture.

Evaluation Level	Task	Points (for Pair)	Points (for Individual)
Mandatory (the functions should work correctly when no file is concurrently accessed)	RSFS_open()	8	8+2
	RSFS_append()	10	10+2
	RSFS_read()	10	10+2
	RSFS_fseek()	4	4+1
	RSFS_write()	10	10+2
	RSFS_close()	4	4+1
	Documentation	4	4
Advanced	The above functions should work correctly when a file is accessed by multiple readers concurrently, or by a write exclusively, at a time.	10 (mandatory)	10 (bonus)

2. Implementation Requirements and Guides

As a prerequisite of this project, you should be familiar with the file system interface and the file system implementation covered in L21 and L22 of the lectures.

2.1 Provided code package

The data structures, their implementations, the sample API code, and the testing 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 development, type “make” to compile the package and get executable application named “app”; type “make clean” to clean up all but the source code if needed.
- def.h: This file contains the definitions of global constants, main data structures and API functions.
- dir.c: This file contains the implementation of search, insertion and deletion of directory entries in the root directory. Note: for simplicity, only one data block is allocated to the root directory, which allows “BLOCK_SIZE/sizeof(struct dir_entry)” files to exist at a time.
- inode.c: This file contains the declaration of inodes, inode bitmap, and their guarding mutexes, as well as the functions for allocating and freeing inodes.
- open_file_table.c: This file contains the declaration of open_file_table, which is an array of open_file_entries, and its guarding mutex. It also contains the functions for allocating and freeing open file entries.
- data_block.c: This file contains the declaration of data_blocks, data bitmap and its guarding mutex, as well as the functions for allocating and freeing data blocks.
- api.c: This is the major file that contains the implementation of basic functions provided by a typical file system.
- application.c: This file contains the application (testing) code that calls the API to create, open, append, read, fseek, write, close and delete files. It tests the case of a single thread or multiple concurrent threads accessing the files.
- sample_output.txt: This file contains the sample outputs of running 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 stored in disks and main memory. In this project, however, all these data structures are implemented in the main memory for simplicity (and thus it is considered 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 (which is 32 by default). 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 each block (1: used, 0: not used). Also, to assure mutually-exclusive access of data_bitmap, lock 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

Each inode is defined as "struct inode" with the following fields:

- int block[NUM_POINTER]; //an array of block-numbers of the data blocks assigned to this file
- int length; //length of this file in the unit of byte

Here, array "block" tracks up to NUM_POINTERS data blocks; that is, it implements a direct indexing approach. The field "length" keeps track of the length of a file in the unit of byte.

The whole space for storing up to NUM_INODES 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.

Mutexes inodes_mutex and inode_bitmap_mutex are used to guard mutually-exclusive access to each inode and the inode_bitmap array, 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., name of the file) and inode_number (i.e., inode of the file). To save space and simplify implementation, a name is just one character and an inode_number is a byte (also declared as type “char”).

The directory entries of the root directory are stored in a single data block. When an entry is empty, its “name” field is 0 (i.e., char ‘\0’); otherwise, the “name” field records the file name, and in this case the “inode_number” field stores the inode number of the file.

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.

Also note that, in this project, files directly belong to the root directory; that is, no hierarchical directory structure is expected to be implemented.

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)
- “int inode_number” - the inode number of this opened file
- “int access_flag” - it takes the value of **RSFS_RDONLY** or **RSFS_RDWR** indicating that the file is opened for read-only (thus the opener is a reader) or read-write (thus the opener is a writer).
- “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, int inode_number)
- 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. It returns 0 when initialization succeeds or –1 otherwise.

Though this function is already provided, you should read it to get familiar with the provided file system data structures.

2.3.2 RSFS_create(char file_name) - create an empty file with the given name

The **provided** function works as follows:

- Search the root directory to find the directory entry matching the provided file name. If such entry exists, the function returns with –1; otherwise, the procedure continues in the following.
- Call allocate_inode() to get a free and initialized inode.
- Call insert_dir() to construct and insert a new directory entry (with a certain file name and inode number) to the root directory.

Though this function is already provided, you should read it to get familiar with the provided file system data structures.

2.3.3 RSFS_open(char file_name, int access_flag) - open a file of the given file name with the given access flag (i.e., RSFS_RDONLY or RSFS_RDWR).

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

- Test the sanity of provided arguments
- Find the directory entry matching the provided file name.
- From the directory entry, find the corresponding inode for the file.
- Find an un-used open file entry, and have it initialized.
- 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_append(int fd, void *buf, int size) - append “size” bytes of data from the buffer pointed to by “buf”, to the file with file descriptor “fd”.

The comments on file api.c include the suggested steps for implementation. Note that, the comments do not provide the detail on how to append the content in buf to the data blocks of the file, from the current end of the file. You need to figure it out.

2.3.5 RSFS_fseek(int fd, int offset) - change the position of the file with file descriptor "fd" to "offset". Note that the change is made only if offset is within the range of the file (i.e., 0, ..., file_length-1). This function should return the new position of the file.

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

2.3.6 RSFS_read(int fd, void *buf, int size) - read up to "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 suggested steps.

2.3.7 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 (say, x) of the file. The original content of the file, from position x, is removed.

For example, if the file with descriptor "fd" originally has content: "charliecharliecharlie". Calling RSFS_write(fd, 4, "hello") would result in the new content of "charhello".

No detailed steps are provided. You need to figure them out.

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

According to the **suggested steps provided by the comments in api.c**, the function should check the sanity of the arguments, and free the open file entry.

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

In this already-implemented function, if there exists a file with the provided file name, the function should free the data blocks, inode and directory entry of the file for the provided file name.

2.3.10 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 (Advanced)

In this part, 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 can be opened again with only RSFS_RDONLY for any times, but it should not be opened with RSFS_RDWR.
- After a file has been opened with RSFS_RDWR and before it is closed, the file should not be opened with either RSFS_RDONLY or RSFS_RDWR.
- When a thread requests to open a file but the request is not allowed for now (due to the above cases), the thread should wait till it is allowed to open the file.

As a hint, you need to make some additions to the provided data structures (e.g., the inodes) 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 may 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.

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 check 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.