Project 2 RSFS: Implementing A Ridiculously Simple In-Memory 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. The RSFS system will be developed on top of **Linux, not xv6**.

To facilitate your development, the major data structures and the code for most 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**, you are expected to implement the functions of open, append, read, seek, close and delete.
- As the **advanced** level of evaluation, which is **mandatory for pairs** but **optional for individuals**, your system is also expected to implement the write and cut functions.

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 should be developed based on understanding the bigger picture.

Evaluation Level	Task	Points (for Pair)	Points (for Individual)
Basic (Mandatory for both individuals and pairs)	RSFS_open()	10	10+4
	RSFS_append()	15	15+6
	RSFS_read()	10	10+4
	RSFS_fseek()	10	10+4
	RSFS_close()	5	5+1
	RSFS_delete()	5	5+1
	Documentation	5	5
Advanced (Mandatory for pairs; optional for individuals)	RSFS_write()	10	10 (bonus)
	RSFS_cut()	10	10 (bonus)

2. Implementation Requirement and 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 runs a Linux-like 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 (which is an array of open_file_entries) 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. **You will add your code to this file.**
- application.c: application (testing) code that calls the API to create, open, append, read, fseek, write, cut, close and delete files. **You can add more test cod to this file.**
- sample_output.txt: sample outputs of running the provided application (testing) code.

2.2 Main data structures

Recall from our in-class discussion of file system implementation (Lecture 22), 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.

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 bytes 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 a block (1: used, 0: not used). 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(), which returns the block number allocated
- void free_data_block(int block_num), which frees the block with the given block number

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]; //array of block-numbers of the data blocks assigned to this file
- int length; //length of this file in byte
- int num_current_reader; //number of reader threads/processes that concurrently open this file
- pthread_mutex_t rw_mutex; //mutex to ensure either a single writer or multiple readers open this file
- pthread_mutex_t read_mutex;//mutex to ensure mutually exclusive access of num_current_reader

Here, array block tracks a fixed number (specified as NUM_POINTER) of data block numbers; hence we implement only direct index. The field length tracks the length of a file in the unit of byte. The fields of num_current_reader, rw_mutex and read_mutex are used to regulate concurrent reading and exclusive writing; **recall the solution of reader/writer problem discussed in class**.

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.

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 (i.e., the index of this file's inode in array inodes).

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 assure 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.

Also note that, in this project, files directly belong to the root directory; that is, no hierarchical directory structure is 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 (i.e., 1) or not (i.e., 0). 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 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, 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 (Basic)

In this part, 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 manipulation of 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 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.

Though this function is already provided, you should read it to get familiar with the manipulation of 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 (which can be RSFS_RDONLY or RSFS_RDWR).

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.
- From the directory entry, find the corresponding inode entry for the file.
- Make sure the file is either opened as RSFS_RDONLY by one/multiple readers only or as RSFS_RDWR by one writer only. Specifically (recall the solution to reader/writer problem!):
    - If the file was already opened with flag RSFS_RDWR by a writer, the caller is blocked until the file is closed by the writer.
    - If the file was already opened with flag RSFS_RDONLY by one/multiple readers:
        - If the current caller wants to open with RSFS_RDONLY, the file can be open
        - If the current caller wants to open with RSFS_RDWR, the caller is blocked until the file is closed by all of the readers
- Find an un-used open file entry to use, 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 at its end.

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 should review the file implementation discussed in Lecture L22 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. 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 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_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.

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

According to the **suggested steps provided by the comments in api.c**, 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.

2.3.9 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 (Advanced)

In this part, you are expected to implement the following functions.

2.4.1 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 are expected to figure them out.

2.4.2 RSFS_cut(int fd, int size) - cut up to size bytes of data from the file with file descriptor fd, starting from the current position of the file.

For example, if the file with descriptor fd originally has content: “charliecharliecharlie”. Calling RSFS_cut(fd, 4, 3) would result in the new content of “charcharliecharlie”.

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

2.5 Test code

File application.c provides sample code to test all the above-described functionality. You are also 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, if you are not able to complete the whole project w.

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.