

Project 4: File System in pi-OS

ENEE 447: Operating Systems — Spring 2017

Assigned Date: Thursday, April 6th.

Due Date: Sunday, April 16th. @11:59 p.m.

1. Abstract

In project 3, we learned about software interrupt (SWI). We implemented system calls in which trigger software interrupt to perform privileged operation in Supervisor Mode. In project 4, we will learn about file system in pi-OS and build system calls to support file I/O operations. We will use **File Allocation Table (FAT)** file system in this project. On top of the provided FAT file system interface, we will implement **virtual file system (VFS)**, which allow client applications to access files in different types of concrete file systems in a uniform way.

2. Introduction

File System

A file is an array of bytes stored on storage device which ranges of bytes can be read/written. File system consists of **many** files and it is the methods and data structures that an operating system uses to keep track of all those files on a disk or partition. Different operating system uses different file system base on different need.

Files need **names** so programs can choose the right one. Therefore, beside the actual data of the file, it is also important to store the **name** of the file along with its data. The file system achieves this by having a structure called inode.

iNode Data Structure

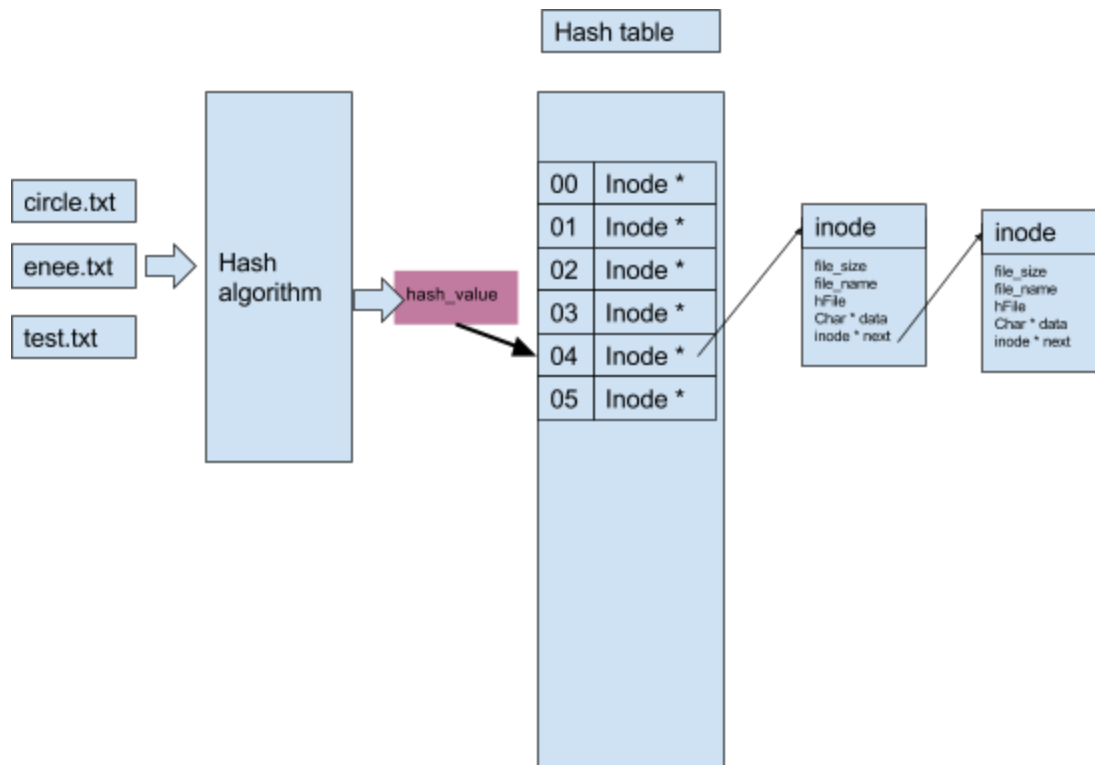
```
typedef struct inode{
    unsigned magic;
    unsigned file_size;
    char file_name[FS_TITLE_LEN];
    unsigned hFile;
    int file_offset;
    int use_count;
    int dirty_flag;
    unsigned permit;
    char data[INODE_DATA_BUFFER_LEN];
    inode * next;
};
```

Inode data structure is used to store metadata and the actual content of the file. The inode data structure is defined in *lib/kernel.cpp*. **magic** stores the magic number 0x0447, which indicates the files store on files system of pi-OS. **hFile** (unique index for file), an integer, is used when we need to communicate with FAT file system driver.

Now, we know that files are represented as inodes in file system. We need a data structure to store all the files (inodes). One trivial approach will simply use a linear array to store all the inodes. This approach may suffer from performance overhead when number of file in system become large. The alternative approach is using a hash function. The second approach results in a better performance if a good hash algorithm is used.

The default value for the **inode_list_len** is 512 in pi-OS file system.

Below is a graphical representation of the second approach (hash algorithm) discussed above. Feel free to use any of the approaches discussed above.



Now, we have a way to uniquely represent a file in inode data structure. When a program is given a name of the file, it is able to find a corresponding inode structure of that given file. The inode structure stores the metadata, as well as the actual data of that particular file.

lib_api File Operations (APIs)

File APIs	arg1	arg2	arg3	arg4 (SWI num)	return value
open	char * name	int permit		5	void *
read	void * pFile	char * buffer	int size	3	byte read
wirte	void * pFile	char * buffer	int size	4	byte write
close	void * pFile			6	0 - success -1 - fail
seek	void * pFile	int offset		7	0 - success -1 - fail

In order for user program to interact with files on storage device, we need to provide system calls APIs for user to execute privileged operations. In this part, you need to implement five systems calls summarized on the chart above.

A standard flow of accessing file involve the following steps:

- **open** (filea_name) returns *pFile
- **read** (pFile, buffer, size_to_read)
- **close** (pFile)

Similarly, a standard flow of modifying file involve the following steps:

- **open** (filea_name) returns *pFile
- **write** (pFile, buffer, size_to_read)
- **close** (pFile)

The **close()** operation is necessary even OS will eventually help to close all opened file upon exit of the program:

1. Prevent run out of file handles available for your process (new open()s will failed)
2. Clear Buffered data

File Allocation Table (FAT)

FAT is a type of file systems which is widely used for SD card or USB storage. In this project, we will be using the SD card as our main storage device for files. Our SD card has a default FAT file system format. The FAT file system format has its advantage and disadvantages.

Advantages:

- Best compatibility for cross-platforms
- Portable, easy to manage.
- non-journaling file system (non-journaling file system extends life of flash drive)

Disadvantages:

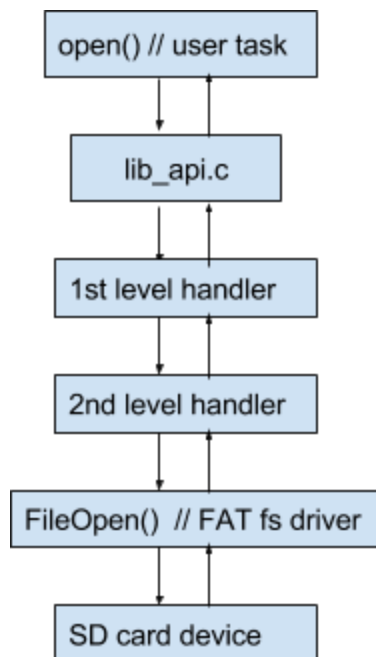
- Single file max size limited to 4GB
- Single partition max size limited to 32GB

To access or make change to files in FAT file system, we need to communicate with the FAT file system driver interface. The FAT file system driver interfaces are stored in ***lib/fs/fat/fatfs.cpp***.

FAT file system Driver Interface (APIs)

Interface	arg1	arg2	arg3	return value
FileOpen	const char *pTitle			File handle number 0 - error
FileRead	unsigned hFile	const void *pBuffer	unsigned ulBytes	byte read
FileWrite	unsigned hFile	const void *pBuffer	unsigned ulBytes	byte write
FileClose	unsigned hFile			1 - success 0 - error

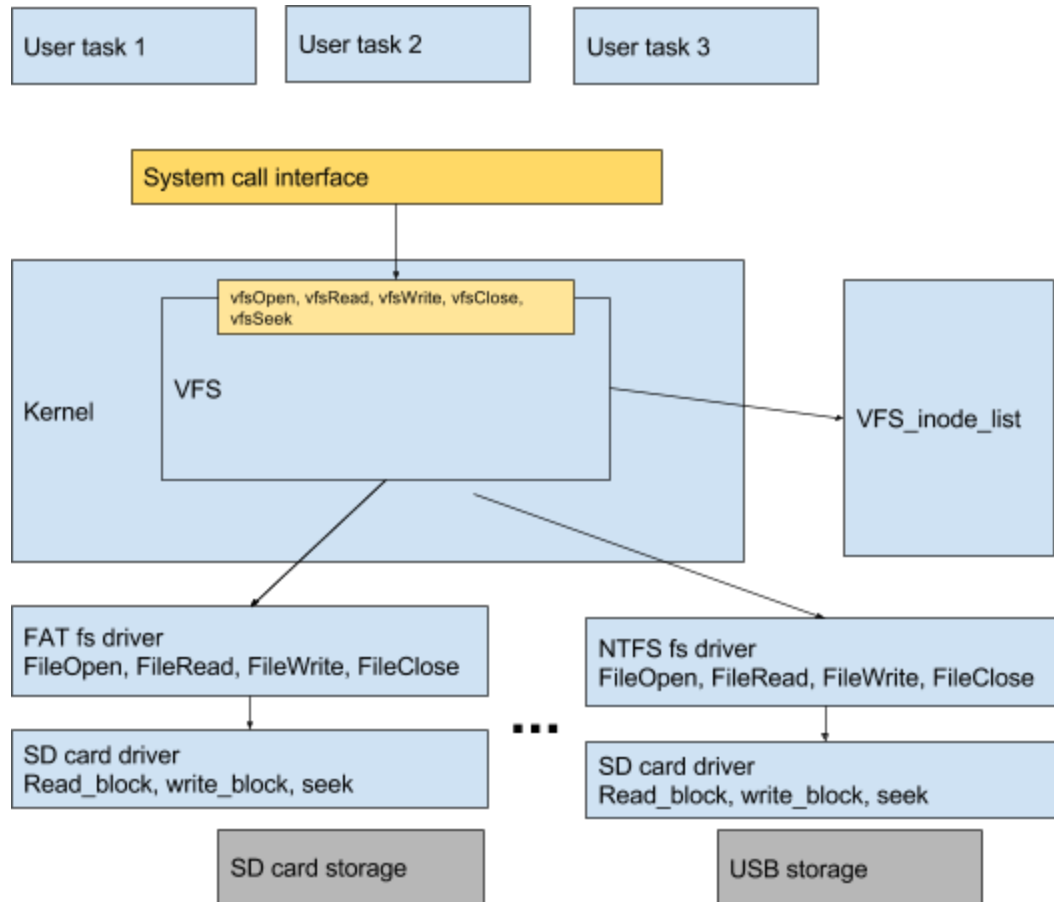
Kernel is possible to directly call FAT interfaces (APIs) to access or modify a file stored on FAT file system.



However, the direct file manipulation has its limitation (and we should NOT do it in this project). What if we want to read a file, which is stored on a partition other than FAT file system, for example, NTFS files system?

We want the system call APIs for files remains **unchanged**, users could access or manipulated with the file without the knowledge of the type of file system it is stored, whether it is stored on FAT or NTFS. Here we introduce **Virtual File System (VFS)**.

Virtual File System (VFS)



Instead of letting user determines what type of file system the file is stored on, the Virtual File System will determine that for users and call the corresponding file system driver interface. All operating systems we are using today have some level of VFS supported. **Linux** VFS allows user to **read and write** content on partition/disk on FAT/FAT32/NTFS/HFS/HFS+ file system. **Mac OS** VFS only support **read** operation on partition/disk with FAT/FAT32/NTFS, but no native support on **write**.

VFS Interface (APIs)

Interface	arg1	arg2	arg3	return value
vfsOpen	char * name	unsigned permit		void * fileHandler
vfsRead	void * fileHandler	char *Buffer	unsigned size	byte read
vfsWrite	void * fileHandler	char *Buffer	unsigned size	byte write
vfsClose	void * fileHandler			0 - success
vfsSeek	void * fileHandler	int offset		0 - success

Read/Write Buffer

Storage devices (hard disk, usb devices, SD cards) are secondary storage, which means direct modification will be slow comparing to access or modification on primary storage. Therefore, any operations on a file should be performed in primary storage (RAM) to prevent performance overhead.

In other words, all access or modification of a file should be performed in primary storage(RAM) which means it loads into a buffer. Write data in buffer back to secondary storage upon return (when close() a file).

3. Your mission

The initial project defines the behavior of file operations in task4, task 5 and task6, your implementation should behave like example output provided.

```

void task4_run()
{
    char buffer[BLOCKLEN];
    int temp;
    memset(buffer, 0, BLOCKLEN);
    char string01[300] = "task4 controls";
    printf("TASK4: Now we are in task4 .*****");
    sleep(1);
    void * pFile = open(file01, 1);

    temp = read(pFile, buffer, 100);
    printf("Read %d bytes from %s, content:%s", temp, file01, buffer);

    memset(buffer, 0, BLOCKLEN);
    strncpy(buffer, string01,10);
    temp = write(pFile, buffer, 60);
    printf("Write %d bytes to %s", temp, file01);

    seek(pFile, 0);

    memset(buffer, 0, BLOCKLEN);
    temp = read(pFile, buffer, 5);
    printf("Read %d bytes from %s, content:%s", temp, file01, buffer);

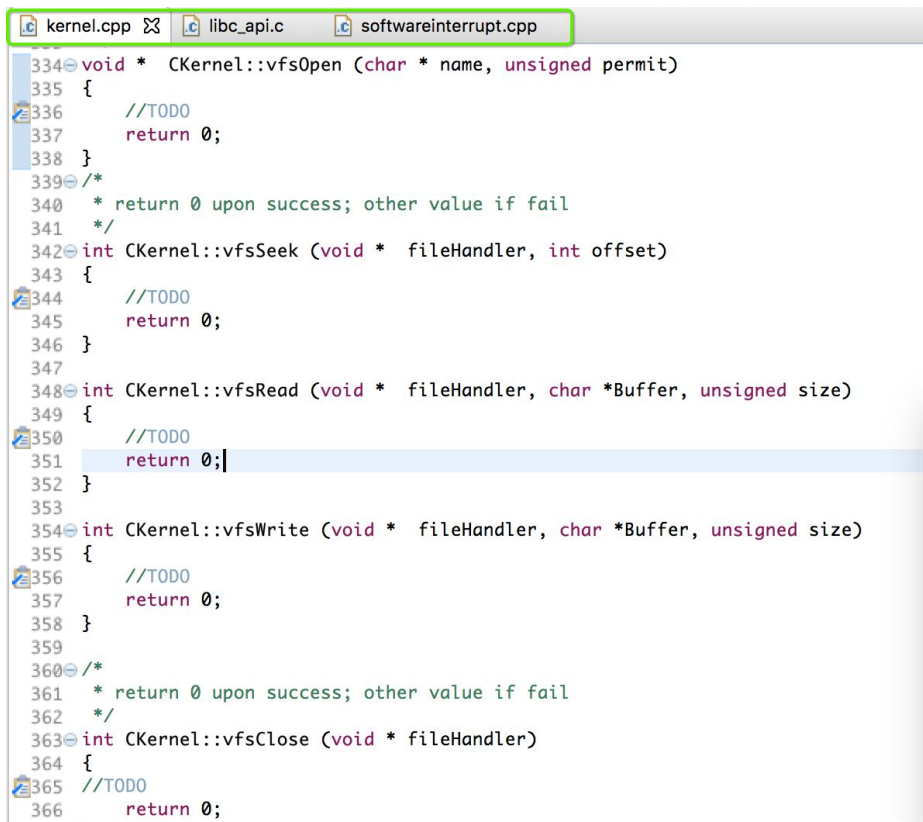
    seek(pFile, 0);

    memset(buffer, 0, BLOCKLEN);
    temp = read(pFile, buffer, 100);
    printf("Read %d bytes from %s, content:%s", temp, file01, buffer);

    seek(pFile, 5);
}

```

You need to understand FAT file system layer, and implement the VFS layer and add system calls. Follow designing document provided to you. Three missions are list in design document.

A screenshot of a code editor with three tabs: 'kernel.cpp', 'libc_api.c', and 'softwareinterrupt.cpp'. The 'kernel.cpp' tab is active, showing C++ code for kernel functions. The code includes comments and return statements, with line numbers 334 through 366 visible. The functions shown are CKernel::vfsOpen, CKernel::vfsSeek, CKernel::vfsRead, CKernel::vfsWrite, and CKernel::vfsClose. Most functions are currently implemented with 'return 0;' and '//TODO' comments.

```
334 void * CKernel::vfsOpen (char * name, unsigned permit)
335 {
336     //TODO
337     return 0;
338 }
339 /*
340  * return 0 upon success; other value if fail
341  */
342 int CKernel::vfsSeek (void * fileHandler, int offset)
343 {
344     //TODO
345     return 0;
346 }
347
348 int CKernel::vfsRead (void * fileHandler, char *Buffer, unsigned size)
349 {
350     //TODO
351     return 0;
352 }
353
354 int CKernel::vfsWrite (void * fileHandler, char *Buffer, unsigned size)
355 {
356     //TODO
357     return 0;
358 }
359
360 /*
361  * return 0 upon success; other value if fail
362  */
363 int CKernel::vfsClose (void * fileHandler)
364 {
365     //TODO
366     return 0;
```

User task should **not** access file directly through FAT file system driver. **You need to implement VFS layer and all file operations should go through VFS layer.** Make sure your final submission look like screenshots within *File System Design in pi-OS.pdf*.

4. Implementation guide:

Please refer to *File System Design in pi-OS.pdf* for implementation details.

5. Submission

No extension, any late submission (after **Sunday, April 16th. @11:59pm**) will result in penalty. If you submit one day late, 10% will be deducted from your project score. If you submit two days late, 20% will be deducted, etc.

What is inside your zip submission:

- Design/Documentation Report in **PDF format**. (include **screenshot** for working system call implementation with task 4, task 5 & task 6)
- All Project files, make sure it is compliable, any compile error automatically 10% off.

Demo:

- Make sure your code is working before submission, TA will download your code and compile there, if your code did **not compile**, and result of **demo not matching** our output. at least **10% project will be deducted**.
- Any reschedule for demo should be completed within the **same week** of demo.
- You need to answer questions during the demonstration. You can find example questions for project 4 on Piazza before demonstration. Failure of the demonstration would get you 15% deduction of score, and you would need to show again...

6. Honor Code

Never show your code to your classmates. Copying code and idea is considering cheating and will be reported to Student Honor Council.

The University of Maryland, College Park has a nationally recognized Code of Academic Integrity. This Code sets standards for academic integrity at Maryland for all undergraduate and graduate students. As a student you are responsible for upholding these standards for this course. It is very important for you to be aware of the consequences of cheating, fabrication, facilitation, and plagiarism. For more information on the Code of Academic Integrity or the Student Honor Council, please visit <http://www.shc.umd.edu>

7. Reference:

What happens if I don't call fclose() in a C program?

<http://stackoverflow.com/questions/8175827/what-happens-if-i-dont-call-fclose-in-a-c-program>

File-System APIs

<http://pages.cs.wisc.edu/~harter/537/lec-18.pdf>