File System Design in pi-OS

Introduction

File related operations from user's view

Open, read, write, close, create, delete
void * open(char * name, int permit)
int close(void * pFile)
int seek(void * pFile, int offset)
int read(void * pFile, char * buffer, int size)
int write(void * pFile, char * buffer, int size)
Explanation:

1) Permit can take 0 or 1. 0 means READ ONLY, 1 means READ & WRITE
2) Name is the file name. Don't support directory. All files will be in SD card top folder
3) pFile is the File handler. It will be interrupted as inode pointer in VFS.
4) If the permit is 0 and file doesn't exist, return open error code 0; if permit is 1 and file doesn't exist, VFS will create one in top folder. If someone else has already open the file as READ ONLY, the following user could open the same file as READ ONLY. If someone else has already open the file as READ & WRITE, the following user could open the same file as READ ONLY or READ & WRITE.
5) Seek can move current file offset to a value.
6) All buffer data will be interrupted as string, which ends with '\0'
7) Only support small file size, less than 512 bytes

Block device

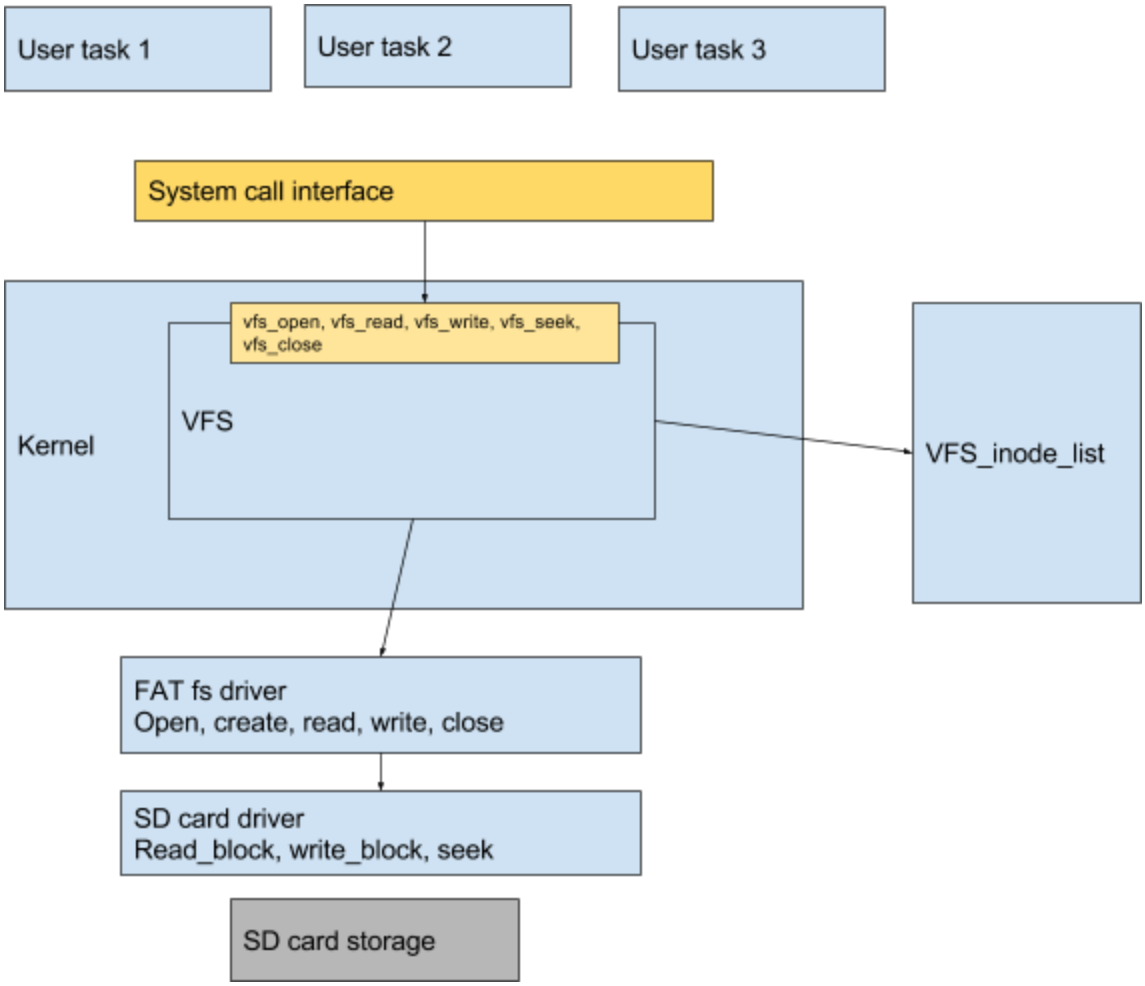
SD Card.

File System Type

Types: FAT fs, since SD card supports FAT.

VFS

File system in pi-OS



mission

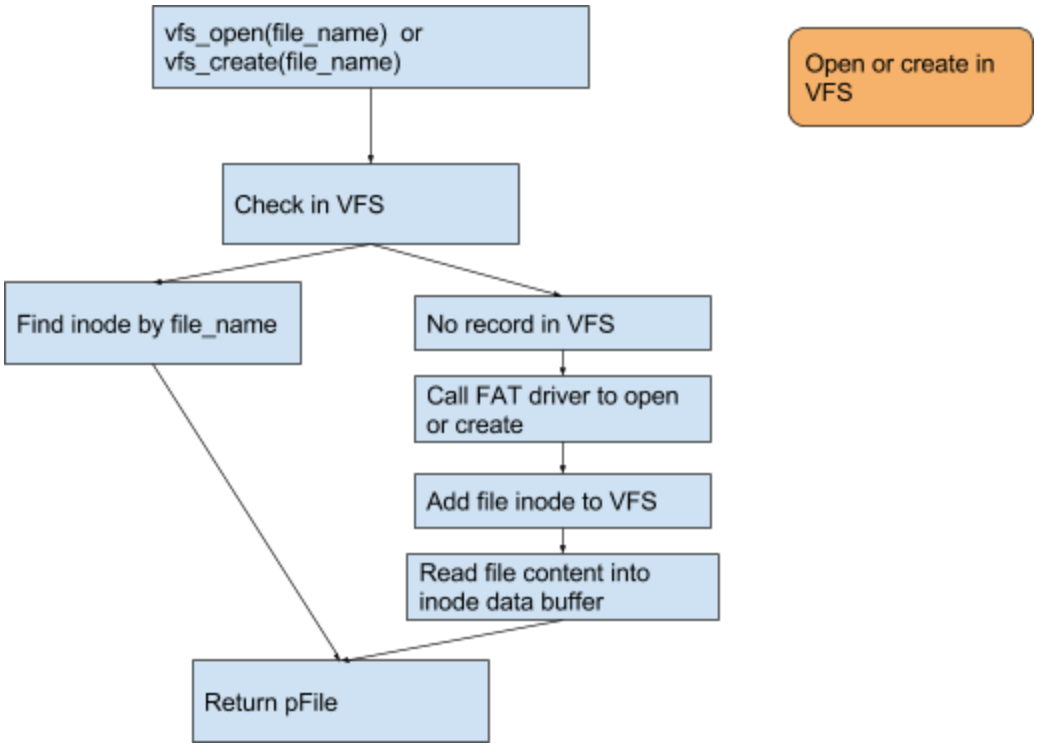
1 Implement VFS in pi-OS

Inode structure:

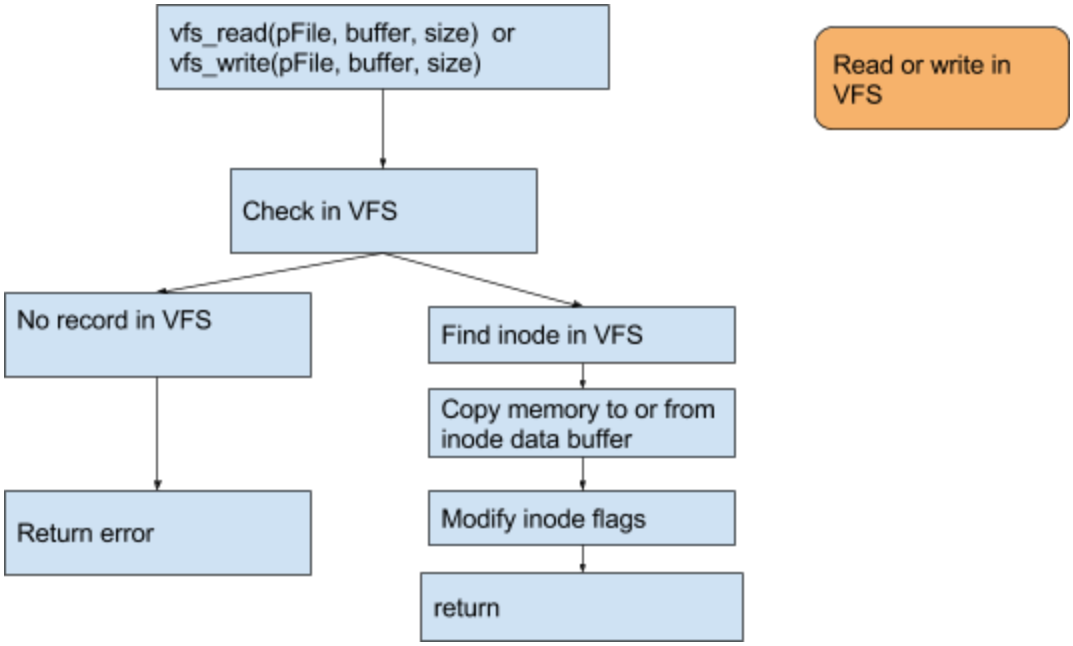
```
40⊖typedef struct inode{
41      unsigned magic;
42      unsigned file_size;
43      char file_name[FS_TITLE_LEN];
44      unsigned hFile;
45      int file_offset;
46      int use_count;
47      int dirty_flag;
48      unsigned permit;
49      char data[INODE_DATA_BUFFER_LEN];
50      inode * next;
51  };
52
53  static inode * inode_list[INODELISTLEN];
54  unsigned inode_count=0;
55
```

Inode_list can hold 256 inode pointers. Each opened file corresponds to one inode. VFS should store inode pointer in inode_list. When opening a file, VFS should read file content (if exists) into inode->data buffer, and also fill in other fields as well. Magic must be 0x0447. **hFile is the fileHandler for FAT driver, not the same as inode handler.** File_offset record current file cursor position, read & write will start from this offset instead of the beginning or end of file. User_count record the number of users of this file, VFS would close this file only user_count reaches 0. Dirty_flag is set when user writes something, and it will be checked when VFS closes the file.

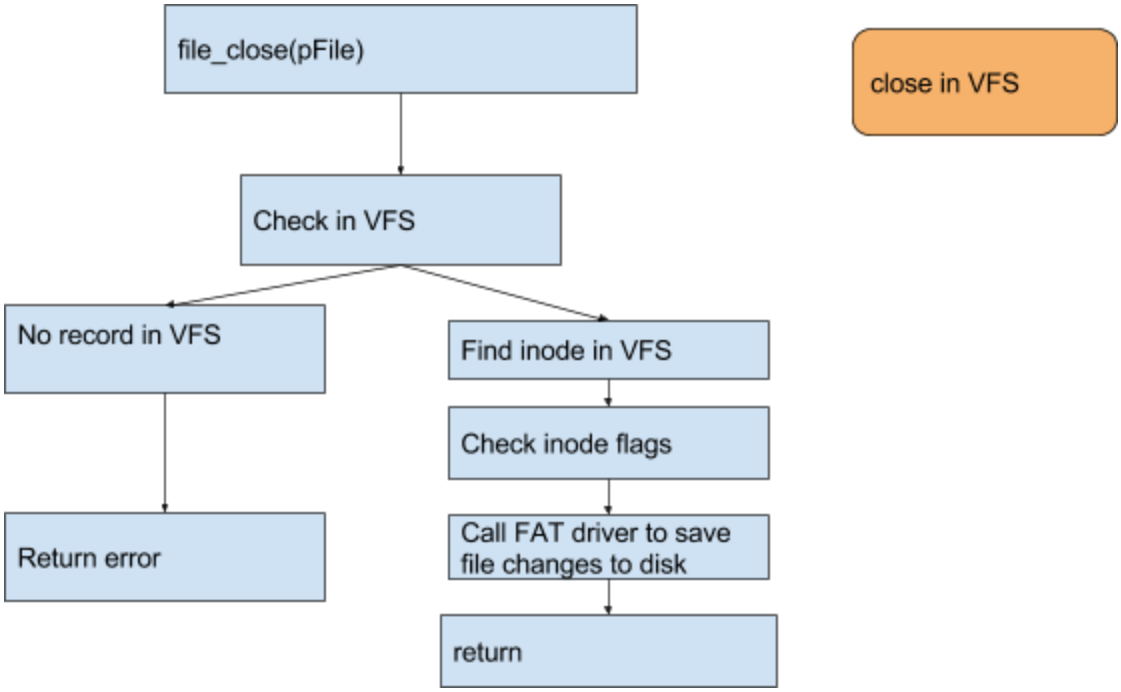
VFS open and create:

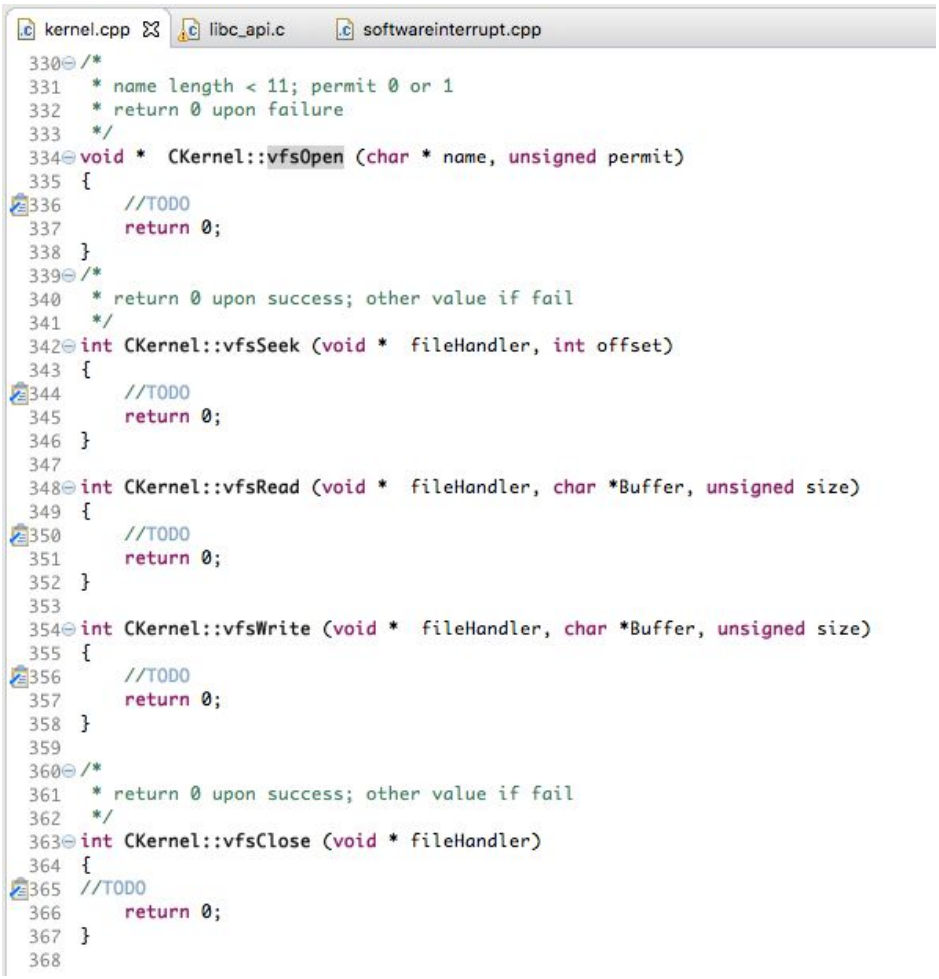


VFS read and write:



VFS close:


file_close(pFile)
close in VFS
Check in VFS
No record in VFS
Find inode in VFS
Check inode flags
Return error
Call FAT driver to save file changes to disk
return



Find FAT fs API usage in CKernel::FileRun (void):

unsigned hFile = m_FileSystem.FileCreate (FILENAME);
*m_FileSystem.FileWrite (hFile, (const char *) Msg, Msg.GetLength ())*
hFile = m_FileSystem.FileOpen (FILENAME);
m_FileSystem.FileRead (hFile, Buffer, sizeof Buffer)
m_FileSystem.FileClose (hFile)

2 Add file-operation system calls

Expected behavior 1:

```
#define BLOCKLEN 512
char file01[] = "test01.txt";
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);

    temp = write(pFile, buffer, 60);
    printf("Write %d bytes to %s", temp, file01);

    temp = write(pFile, buffer, 60);
    printf("Write %d bytes to %s", temp, file01);

    memset(buffer, 0, BLOCKLEN);
    temp = read(pFile, buffer, 100);
    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);

    close(pFile);
```

```
142     //sleep(2);
143     memset(buffer, 0, BLOCKLEN);
144     pFile = open(file01, 1);
145     temp = read(pFile, buffer, 100);
146     printf("Read %d bytes, content:%s", temp, buffer);
147
148     temp = write(pFile, buffer, 60);
149     printf("Write %d bytes to %s", temp, file01);
150
151     seek(pFile, 0);
152
153     memset(buffer, 0, BLOCKLEN);
154     temp = read(pFile, buffer, 100);
155     printf("Read %d bytes from %s, content:%s", temp, file01, buffer);
156     close(pFile);
157
158     printf("TASK4:  stops .*****");
159     exit(1);
160     printf("TASK4:  this message should not appear");
161 }
```

Output:

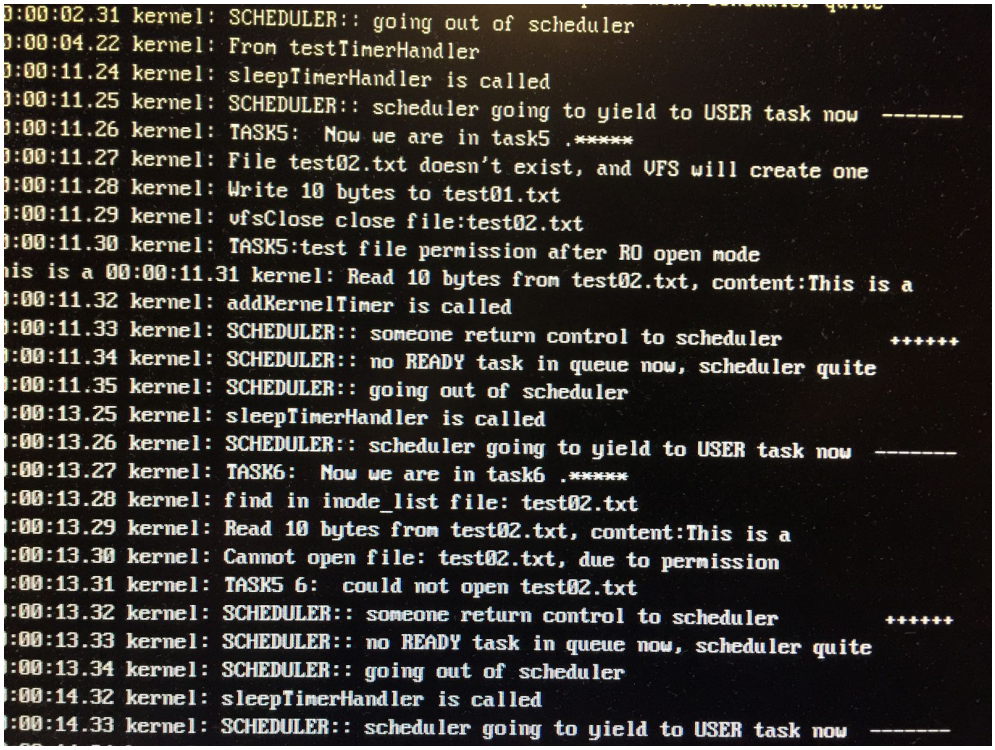
```
00:00:01.15 kernel: tag 003
00:00:01.20 kernel: vfsClose close file:circaa.txt
00:00:01.20 kernel: tag 004
00:00:01.20 kernel: Now print the task queue
00:00:01.20 kernel: Task_ID=0, queue_NUM = 0, task=2260580, priority=1
00:00:01.21 kernel: Task_ID=1, queue_NUM = 1, task=2260444, priority=10
00:00:01.21 kernel: Task_ID=103, queue_NUM = 2, task=2259900, priority=30
00:00:01.21 kernel: Task_ID=104, queue_NUM = 3, task=2259764, priority=30
00:00:01.21 kernel: Task_ID=105, queue_NUM = 4, task=2259628, priority=30
00:00:01.22 kernel: flag 2222
00:00:01.22 kernel: KERNEL:: Some one calls the kernel, or the queue just starts over
00:00:01.22 kernel: addKernelTimer is called
00:00:01.22 kernel: SCHEDULER:: scheduler going to yield to USER task now  -------
00:00:01.23 kernel: Task is entering now
00:00:01.23 kernel: TASK4:  Now we are in task4 .******
00:00:01.23 kernel: addKernelTimer is called
00:00:01.23 kernel: SCHEDULER:: someone return control to scheduler         ++++++
00:00:01.24 kernel: SCHEDULER:: scheduler going to yield to USER task now  -------
00:00:01.24 kernel: Task is entering now
00:00:01.24 kernel: addKernelTimer is called
00:00:01.24 kernel: SCHEDULER:: someone return control to scheduler         ++++++
00:00:01.25 kernel: SCHEDULER:: scheduler going to yield to USER task now  -------
00:00:01.25 kernel: Task is entering now
00:00:01.25 kernel: addKernelTimer is called
00:00:01.25 kernel: SCHEDULER:: someone return control to scheduler         ++++++
00:00:01.25 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:01.26 kernel: SCHEDULER:: going out of scheduler
00:00:02.23 kernel: sleepTimerHandler is called
00:00:02.23 kernel: SCHEDULER:: scheduler going to yield to USER task now  -------
00:00:02.23 kernel: File test01.txt doesn't exist, and VFS will create one
00:00:02.23 kernel: Read 0 bytes from test01.txt, content:
00:00:02.24 kernel: Write 10 bytes to test01.txt
00:00:02.24 kernel: Read 5 bytes from test01.txt, content:task4
00:00:02.24 kernel: Read 10 bytes from test01.txt, content:task4 cont
00:00:02.24 kernel: Write 10 bytes to test01.txt
00:00:02.24 kernel: Write 10 bytes to test01.txt
00:00:02.25 kernel: Read 0 bytes from test01.txt, content:
00:00:02.25 kernel: Read 25 bytes from test01.txt, content:task4task4 conttask4 cont
00:00:02.25 kernel: vfsClose close file:test01.txt
task4task4 conttask4 cont00:00:02.26 kernel: Read 25 bytes, content:task4task4 conttask4 cont
00:00:02.26 kernel: Write 25 bytes to test01.txt
00:00:02.26 kernel: Read 50 bytes from test01.txt, content:task4task4 conttask4 conttask4task4 conttask4 cont
00:00:02.27 kernel: vfsClose close file:test01.txt
00:00:02.28 kernel: TASK4:  stops .******
00:00:02.29 kernel: SCHEDULER:: someone return control to scheduler         ++++++
00:00:02.30 kernel: SCHEDULER:: no READY task in queue now, scheduler quite
00:00:02.31 kernel: SCHEDULER:: going out of scheduler
00:00:04.22 kernel: From testTimerHandler
```

Expected behavior 2:

```
163 char file02[] = "test02.txt";
164 char file03[] = "test03.txt";
165 void task5_run()
166 {
167     sleep(10);
168     printf("TASK5:  Now we are in task5 .*****");
169     char buffer[BLOCKLEN];
170     int temp;
171     memset(buffer, 0, BLOCKLEN);
172     char string01[300] = "This is a test";
173 
174     void * pFile = open(file02, 1);
175     seek(pFile, 0);
176     strncpy(buffer, string01,10);
177     temp = write(pFile, buffer, 60);
178     printf("Write %d bytes to %s", temp, file01);
179     close(pFile);
180 
181     printf("TASK5:test file permission after RO open mode");
182     pFile = open(file02, 0);
183     seek(pFile, 0);
184     memset(buffer, 0, BLOCKLEN);
185     temp = read(pFile, buffer, 100);
186     printf("Read %d bytes from %s, content:%s", temp, file02, buffer);
187 
188     sleep(3);
189     close(pFile);
190 
191     exit(1);
192     printf("TASK5:  this message should not appear");
193 }
```

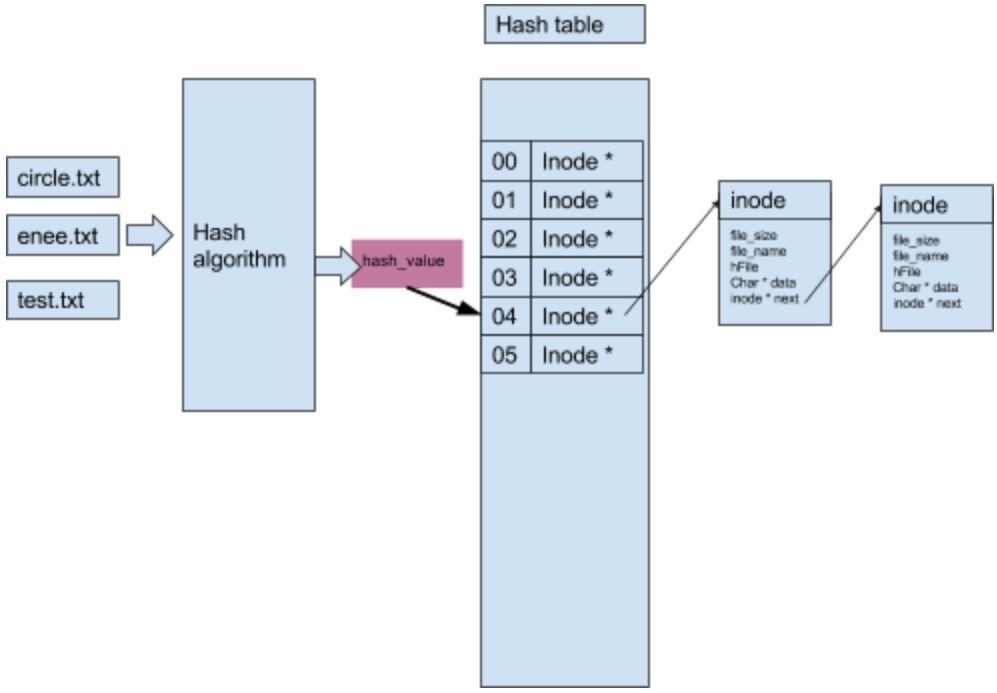
```
196 void task6_run()
197 {
198     sleep(12);
199     printf("TASK6:  Now we are in task6 .*****");
200     char buffer[BLOCKLEN];
201     int temp;
202     memset(buffer, 0, BLOCKLEN);
203     char string01[300] = "This is a test";
204 
205     void * pFile = open(file02, 0);
206     if(0 != pFile){
207         seek(pFile, 0);
208         temp = read(pFile, buffer, 100);
209         printf("Read %d bytes from %s, content:%s", temp, file02, buffer);
210         close(pFile);
211     }else{
212         printf("TASK5 6:  could not open %s", file02);
213     }
214 
215     pFile = open(file02, 1);
216     if(0 != pFile){
217         seek(pFile, 0);
218         temp = read(pFile, buffer, 100);
219         printf("Read %d bytes from %s, content:%s", temp, file02, buffer);
220         close(pFile);
221     }else{
222         printf("TASK5 6:  could not open %s", file02);
223     }
224     exit(1);
225     printf("TASK6:  this message should not appear");
226 }
```

Output:



3 Implement hash algorithm for VFS inode table

(This is used to check files in VFS. No new functional API for user task, but this will improve VFS performance)



Existing codes

FAT FS driver and API:
Find the usage of FAT fs river in CKernel::FileRun (void).
Limitations:
If file exists, open will only return READ ONLY mode file. So if you want to write something to an existing file, you need to read it first, then delete it, then create it, and put original content and added content in the file, then close.

Goal

Understanding file systems, FAT file system
Understanding block device and driver, SD card
Understanding and Implementing VFS in pi-OS

References:
https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
https://en.wikipedia.org/wiki/File_Allocation_Table
https://en.wikipedia.org/wiki/Virtual_file_system