

## CS2106: Operating Systems

### Lab 5 – USFAT File System

#### Important:

- The deadline of submission on IVLE is **16<sup>th</sup> November 5pm**
- The total weightage is 1% [Demo-Ex1] + 5%:
  - Exercise 2: 2%
  - Exercise 2: 3%

#### Note:

- This lab is **platform sensitive**. Please use only Linux on Intel.

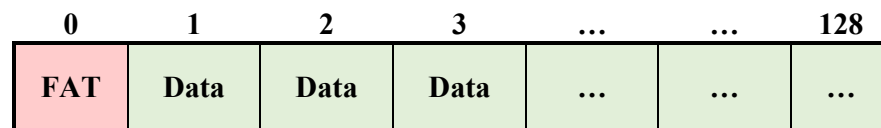
## Section 1. USFAT File System Overview

USFAT (pronounced as /'ʌŋkl/ /su:/ /ɪz/ **FAT**) is a **fictional** file system invented just for CS2106 ☺! It draws inspirations from the basic FAT based file allocation scheme and the MS-DOS FAT16 file system. Your tasks in this lab is to understand and provide functionalities that interact with the underlying USFAT file system.

### 1.1 USFAT File System Layout

Absolute Sector

# →



Data Sector #

→

0      1      2      ...      ...      127

Note that a "logical block" is the same size as a "sector" in USFAT and we use the two terms interchangeably.

There are a total of 129 sectors (with absolute index 0 to 128) and each sector is 256 bytes. So, a typical USFAT file system is  $129 * 256 = 33,024$  bytes in size. The FAT table **occupies one sector** and is located at sector 0. All remaining sectors (128) are used for data storage.

Each FAT entry is **2 bytes** in size, i.e. the FAT contains  $256 / 2 = 128$  entries. Note that the FAT index refers only to blocks in the file data region. For ease of reference (and coding), we will use **data sector number** to indicate sectors in the data region. For example, the status of data sector 2 can be found in  $FAT[2]$ , but the actual storage on the "hard disk" is at sector 3. So, pay attention to whether you are using data sector number of the absolute sector number during coding to avoid "off-by-one" errors.

Each FAT entry can contain one of the following values:

Values	Meaning
<b>0xFFFA</b>	The sector is <b>free</b> .
<b>0xFFF7</b>	The sector is <b>bad</b> (i.e. not working, don't store any content here).
<b>0xFFFF</b>	The sector is the <b>END</b> of a linked sector chain.
<b>0x0000 to 0x007F</b>	The sector leads to the indicated sector as part of a linked sector chain.

Here's a sample FAT printout:

Offset:	+00	+01	+02	+03	+04	+05	+06	+07
0x0000:	FREE	FREE	FREE	FREE	FREE	END*	FREE	003f
0x0008:	0009	000a	0056	FREE	FREE	FREE	FREE	0010
0x0010:	0011	0012	0013	0014	0008	FREE	FREE	FREE
0x0018:	FREE	FREE	FREE	FREE	END*	FREE	FREE	0020
0x0020:	006e	FREE	FREE	FREE	FREE	FREE	FREE	FREE
0x0028:	FREE	FREE	FREE	FREE	FREE	FREE	FREE	FREE
0x0030:	FREE	FREE	FREE	FREE	FREE	FREE	FREE	FREE
0x0038:	FREE	FREE	FREE	FREE	0055	003e	001c	0040
0x0040:	0041	0042	000f	FREE	FREE	FREE	0047	0048
0x0048:	0049	004a	004b	004c	END*	FREE	FREE	FREE
0x0050:	FREE	FREE	FREE	FREE	FREE	005c	0057	0058
0x0058:	0059	005a	005b	003c	005d	005e	005f	0060
0x0060:	0062	END*	0063	0064	0065	0066	0067	0068
0x0068:	0069	006a	006b	006c	006d	001f	006f	0070
0x0070:	0071	0072	0073	0074	END*	FREE	FREE	FREE
0x0078:	FREE	FREE	FREE	007c	003d	FREE	FREE	FREE

From the FAT printout above, we can see that the data sector 0x0000 is free; the data sectors 0x004a→0x004b→0x004c (end) is **part** of a linked sector chain, etc.

## 1.2 Directory (Folder) and File under USFAT

Under USFAT, directory and file both use the file data sectors to store information. For a directory, the data sector stores **directory entries**, which contains information about **files under** that directory. For a file, the data sector stores the **actual file content**.

For simplicity, the USFAT media provided in this lab has the following limitations:

- There is only one directory, the **Root Directory**. It is located at **data sector 5**.
- Directory uses **only 1** sector for its directory entries, which place an upper limit on the number of files it can store.
- Your code only need to work with these limitations in place.
- [Note: These limits are imposed to simplify the exercises, the design of the USFAT is much more general / flexible.]

Each of the directory entry in a directory's data sector occupies **32 bytes** and has the following layout:

Offset	0	...	10	11	12	...	25	26	27	28	...	31		
Usage	Name				Attr	<not used>				Start Sector	File Size			

The name uses the old "8+3" format, where the file name is 8 characters long and the extension takes up 3 characters, e.g., a file with name "sample.cc" is stored as:

0	1	2	3	4	5	6	7	8	9	10
—	—	s	a	m	p	l	e	c	c	—

Note that the filename is right aligned to the "." while the extension is left aligned. The "." itself is **not stored**. We use '—' represents a space, i.e. ' '.

The attribute is a single byte (8 bits):

Bit	7	6	5	4	3	2	1	0
Usage				Is directory?		Is System?	Is Hidden?	Is Readable?

For our exercises, you can assume that **all files have an attribute 0x01**, i.e. readable, not hidden, not a system file and not a directory.

Since each directory entry is 32 bytes and the directory in USFAT can utilize only 1 sector for directory entries, this gives us 256 bytes / 32 bytes = 8 files under a directory **at most**.

### 1.3 USFAT "Media", Library Calls and Utility Program

There are a number of “disk image” files provided for this lab, e.g. *4files.img*, *empty.img*, etc. Each of the file represents a complete USFAT file system. You can imagine they represent simulated storage media like a hard disk, etc.

A **large number** of library calls are provided for you to focus on “high level” file system functionalities. In the common/ directory, take a look at the **USFAT.h** header files which defines all important system parameters and the available library calls. Essentially, “low level” functionalities that deals with reading / write information from / to the media, e.g. sector / FAT reading / writing, etc are available for use.

In addition, a “debug inspector” program, known as USFATI (USFAT Insepector) is also available so that you can view the raw content on a USFAT media easily. Instructions to setup the inspector etc is given in Section 2.

## Section 2. Exercises for USFAT

### 2.1 Directory structure of the skeleton code

There is one additional folder *common/* with the following files:

Filename	Purpose
USFAT.h	USFAT header file with all key definitions and declarations.
USFAT_Util.c	Implementation of all USFAT library functions.
USFAT_Insepct.c	The debug inspector utility program. Compiles into the “USFATI” executable.
Various *.img	Backup copies of all USFAT disk images. <b>In exercise 3, your program will modify the USFAT disk image</b> , so if you ever need to “reset” the disk images, copy the backup over.
makefile	For compiling the USFATI debug inspector as mentioned above. reset.sh: A simple script file to copy the backup images to the exercise directories.

#### Preparation:

1. Go into the common/ folder and type “make” to produce the USFATI executable.
2. Enable the “reset.sh” script file by “chmod 700 reset.sh”
3. Execute the “reset.sh” script file “./reset.sh”, this copy a fresh set of disk images to the exercise directories. Use this step whenever you need to reset your disk images.

## 2.2 Exercise 2

**Main task:** Display the file content of a file under the root directory.

The main function is already written for you. The main function will repeatedly print the directory content of the root directory (i.e. similar to a “ls”), then prompt the user for a file to display (i.e. similar to a “cat” / “less” command). Your task is to implement the function “`read_file( FAT_RUNTIME* rt, char filename[])`” which returns:

- **0** if the file with `filename` cannot be found under the root directory.
- **1** if the operation is successful.

This function attempts to locate the directory entry for the file *filename*, then read **all** data sectors of this print and print them to the screen. **Note: use the `print_as_text()` function when you need to print out the content of a file data sector.** This ensure your output format is exactly the same as ours to facilitate checking.

Several key criteria:

- Entire content of the file should be shown (duh!). This requires you to follow the “*linked sector chain*” by traversing in the FAT.....
- Note that the last sector may not be full! You need to print out **only the valid content**. (hint: use file size.....).
- You are allowed to define as many helper functions as you need.
- You can add / change the parameter(s) of the `read_file()` function if needed.
- The main function should not be changed except the function call to `read_file()` can be modified with new parameters if you change them.

Sample Output (using [4files.img](#), user input in **bold**, file content in **red**):

Filename	Attr	Start	Size
fat.txt	01 <file>	[0x0067]	1563
mystery.abc	01 <file>	[0x003a]	1092
hello.c	01 <file>	[0x007e]	74
rain.txt	01 <file>	[0x0042]	12194

Read File ("DONE" to quit) > **hello.c**

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");

    return 0;
}
```

Note that only 74 bytes of "hello.c" are valid out of 256 bytes in the sector.

Filename	Attr	Start	Size
----------	------	-------	------

```

-----
fat.txt 01 <file> [0x0067] 1563
mystery.abc 01 <file> [0x003a] 1092
hello.c 01 <file> [0x007e] 74
rain.txt 01 <file> [0x0042] 12194
Read File ("DONE" to quit) > hi.txt
"hi.txt" not found!
  Filename      Attr      Start      Size
-----
fat.txt 01 <file> [0x0067] 1563
mystery.abc 01 <file> [0x003a] 1092
hello.c 01 <file> [0x007e] 74
rain.txt 01 <file> [0x0042] 12194
Read File ("DONE" to quit) > mystery.abc
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
..... <Some file content omitted to save space> .....
    }

    fclose(fat_rt.media_f);

    return 0;
}

  Filename      Attr      Start      Size
-----
fat.txt 01 <file> [0x0067] 1563
mystery.abc 01 <file> [0x003a] 1092
hello.c 01 <file> [0x007e] 74
rain.txt 01 <file> [0x0042] 12194
Read File ("DONE" to quit) > DONE

```

There is no  
"hi.txt" in the root

The entire  
"mystery.abc" is  
printed.

To aid your checking, the original files "fat.txt", "mystery.abc", "hello.c" and "rain.txt" (as well as a few other text files) are included in the exercise folder.

## 2.2 Exercise 3

**Main task:** Import a normal file into the USFAT file system.

Similar to exercise 2, the main function is already coded for you. You only need to provide the implementation of the `import_file()` function. This function returns:

- **-1:** if there is any error. Full error lists is given below. OR
- **Non-negative value:** Actual number of bytes copied over.

To facilitate explanation, let us assume we make the following call using the given function prototype:

```
import_file( &runtime, "example.txt", 25 );
```

The function should perform the following checks:

- Ensure the normal file “example.txt” can be opened. You can assume the filename given by user follows the “8+3” filename restriction.
- Ensure the root directory of the USFAT media **does not** have another file with the same filename as “example.txt” as filename should be unique under a directory.
- Ensure the root directory is **not full**, i.e. has less than 8 files currently.
- If any of the above fails, the function returns “-1”.
- Once checks are all cleared, the function will now attempt to copy the “example.txt” into the data sectors.
- The function will **try** to use the first sector as specified (data sector 25) in this example. If the sector is free, copying can start there. Otherwise, you should check subsequent data sectors (e.g. 26, 27, 28 ..... ) and wraps around if needed. If there are no free data sector, the function terminates and return -1.
- Once copying starts, data sector chain needs to be constructed if you need more than one data sector. The logic for getting the next sector is the same: look for the free data sector in the subsequent indices and wrap around if needed. **Remember to modify the FAT entries accordingly as you move along.**
- Copy stops when i) the input file e.g. “example.txt” has been copied fully OR ii) there are no more free data sectors. **Remember to “terminate” your sector chain by setting the END flag in the FAT.**
- The function will also add a new directory entry with the right information into the root directory.
- **Don’t forget to flush the FAT into the actual USFAT media.**
- Finally, the function returns the total number of bytes copied to the caller.
- Note that this exercise **changes the disk images**. Use the “reset.sh” to restore the images if needed.

## Hints and Tips:

- Browse the library calls. You have **many** helpful functions there to keep your pain in check....
- Define useful helper functions to reduce code spaghetti...
- The sample solution is about 120+lines (with newlines, debug code, comments included).
- You can use your `ex2.c` to check whether the files are imported properly.
- Use the utility program `USFATI` to monitor the changes of sectors.

**Sample Output** (using empty.img, user input in **bold**, notable key info in **red**).

The empty.img is an empty USFAT media with **only the root directory taking up data sector 5** at the beginning.

Filename	Attr	Start	Size
-----			
Import File ("DONE" to quit) > <b>hi.c</b>			
Start sector (in Hex) > 0x0			
<b>Import "hi.c" to [0x0000] Data Sector...FAILED!</b>			
-----			
Import File ("DONE" to quit) > <b>hello.c</b>			
Start sector (in Hex) > 0x5			
<b>Import "hello.c" to [0x0005] Data Sector...Written 74 bytes.</b>			
-----			
hello.c	01 <file>	[0x0006]	74
-----			
Import File ("DONE" to quit) > <b>hello.c</b>			
Start sector (in Hex) > 0x4A			
<b>Import "hello.c" to [0x004a] Data Sector...FAILED!</b>			
-----			
hello.c	01 <file>	[0x0006]	74
-----			
Import File ("DONE" to quit) > <b>fat.txt</b>			
Start sector (in Hex) > 0x2f			
<b>Import "fat.txt" to [0x002f] Data Sector...Written 1563 bytes.</b>			
-----			
hello.c	01 <file>	[0x0006]	74
fat.txt	01 <file>	[0x002f]	1563
-----			
Import File ("DONE" to quit) > <b>mystery.abc</b>			
Start sector (in Hex) > 0x0			

"hi.c" doesn't exist.

Data sector 5 is occupied, so next available sector (i.e. 6) is used.

There is already a "hello.c" → import failed.



```

Import "mystery.abc" to [0x0000] Data Sector...Written 1092
bytes.
  Filename      Attr      Start      Size
  -----
  hello.c       01 <file> [0x0006]      74
  fat.txt       01 <file> [0x002f]    1563
  mystery.abc   01 <file> [0x0000]    1092

Import File ("DONE" to quit) > alice.txt
Start sector (in Hex) > 0x4A
Import "alice.txt" to [0x004a] Data Sector...Written 30720
bytes.
  Filename      Attr      Start      Size
  -----
  hello.c       01 <file> [0x0006]      74
  fat.txt       01 <file> [0x002f]    1563
  mystery.abc   01 <file> [0x0000]    1092
  alice.txt     01 <file> [0x004a]   29184

Import File ("DONE" to quit) > DONE

```

Both "fat.txt" and "mystery.abc" are imported fully. You can verify their file size.

The USFAT disk is almost full at this point and can only stores 29,184 bytes out of the full 177,428 bytes for "alice.txt"

The FAT table (use USFATI to inspect) should look like the following afterwards:

Offset:	+00	+01	+02	+03	+04	+05	+06	+07
0x0000:	0001	0002	0003	0004	END*	END*	END*	0008
0x0008:	0009	000a	000b	000c	000d	000e	000f	0010
0x0010:	0011	0012	0013	0014	0015	0016	0017	0018
0x0018:	0019	001a	001b	001c	001d	001e	001f	0020
0x0020:	0021	0022	0023	0024	0025	0026	0027	0028
0x0028:	0029	002a	002b	002c	002d	002e	0036	0030
0x0030:	0031	0032	0033	0034	0035	END*	0037	0038
0x0038:	0039	003a	003b	003c	003d	003e	003f	0040
0x0040:	0041	0042	0043	0044	0045	0046	0047	0048
0x0048:	0049	END*	004b	004c	004d	004e	004f	0050
0x0050:	0051	0052	0053	0054	0055	0056	0057	0058
0x0058:	0059	005a	005b	005c	005d	005e	005f	0060
0x0060:	0061	0062	0063	0064	0065	0066	0067	0068
0x0068:	0069	006a	006b	006c	006d	006e	006f	0070
0x0070:	0071	0072	0073	0074	0075	0076	0077	0078
0x0078:	0079	007a	007b	007c	007d	007e	007f	0007

Several notable observations:

- "Hello.txt" is in sector 6, where the FAT entry is indicated with the END flag as it occupies only 1 sector.
- "Mystery.abc" starts from sector 0, follow the linked sector list to understand the requirement better (use adjacent if possible, otherwise search forward for free sector).

### 2.3 And beyond....

As exercise 3 is “a bit” challenging, I have decided to drop the bonus questions. ☹ However, I hope you have the curiosity (and time) to explore further. Several things you can try (in increasing insanity order):

1. Expand the directory to use multiple sectors. This removes the 8 files per directory limitation. Your code in ex2 can help.
2. Implement subdirectory. (Rather straightforward actually).
3. With (2), extend ex2 and ex3 to support subdirectory, i.e. read file with full path “/dir1/dir2/example.txt”, import file for deeper directory structure etc.

### Section 3. Submission

Zip the following files as A/E0123456.zip (use your **NUSNET user id!**):

- a. **ex2.c** (Remember to remove all debug messages)
- b. **ex3.c** (Remember to remove all debug messages)

Upload the zip file to the "Student Submission→Lab 5" workbin folder on IVLE. Note the deadline for the submission is **16<sup>th</sup> November, 5pm**.

Again, please ensure you follow the instructions carefully (**output format**, how to zip the files etc). **Deviations will be penalized.**

### Reference:

1. "Design of the FAT file system" (Very good read – Much deeper than you'll need) [https://en.wikipedia.org/wiki/Design\\_of\\_the\\_FAT\\_file\\_system](https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system)
2. "Alice in the wonderland." (IP free!), by Lewis Carroll, Project Gutenberg version.
3. "There Will Come Soft Rains", by Ray Bradbury

Note: The above are the sample files stored in the various USFAT disk images. (2) and (3) are good reads and definitely worth your time. You are welcome! ☺