CS3026 Assessment

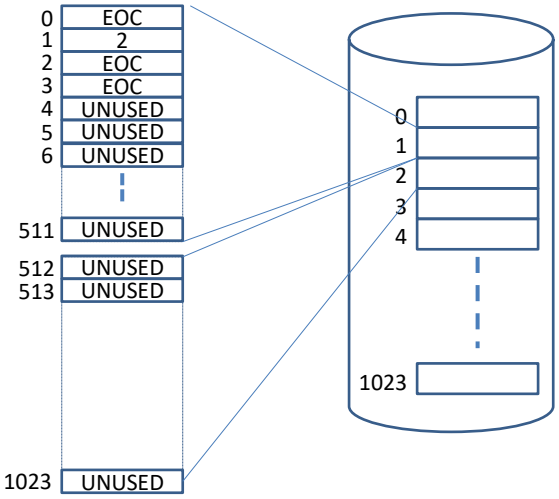
In this assessment, you will implement a simple file system that allows you to manage files and directories in a virtual in-memory disk. The file system is to be based on simplified concepts of a File Allocation Table (FAT). Your implementation will allow the creation of files and directories within this virtual hard disk and the performance of simple read and write operations on files.

Your task is to implement interface functions for creating files and directories and for reading and writing operations. For this assessment, the virtual disk will be simulated by an array of memory blocks, where each block is a fixed array of bytes. Each block has a block number (from 0 to MAXBLOCKS-1). The allocation of a new block to a file is recorded in the FAT. The FAT is a table (an array of integers) of size MAXBLOCKS that acts as a kind of block directory for the complete disk content: it contains an entry for each block and records whether this block is allocated to a file or unused. The FAT itself is also stored on this virtual disk at a particular location.

Files may occupy one or more disk blocks and this allocation is non-contiguous – any block of the disk may be allocated to a file. A disk usually becomes more fragmented over time, the more files are deleted and added. When a file is opened for read, we want to be able to read the blocks of a file from start to end in the correct order. The FAT records the correct sequence of blocks comprising a file in the form of a "block chain": the number stored in one array entry in the FAT may be some index of another array entry – the next block number in the chain – or ENDOFCHAIN (this block is the last block of the file), or UNUSED (block on disk is free). We assume: ENDOFCHAIN == 0 and UNUSED == -1. When reading a file, we first have to look into the FAT to read one block after the other into memory, following such a block chain in the FAT.

For this assessment, we assume that the virtual disk consists of an array of 1024 blocks (MAXBLOCKS), where each block is an array of 1024 bytes (BLOCKSIZE). A block can be either (a) file data (an array of 1024 bytes), (b) directory data (a set of directory entries, as much as can fit within 1024 bytes), or (c) the FAT itself containing information about used and unused blocks. As there are 1024 disk blocks, the FAT has to have 1024 entries. For this assessment, we assume that a FAT entry is a short integer (2 bytes).

Two files are provided for this assessment: You will find the files filesys.c and filesys.h containing data structures and basic functions you may use in your implementation. The two basic functions that are directly interacting with the virtual disk are writeblock() and readblock(). All other functions (those you have to implement) are using these two functions for reading from and writing to the virtual disk.



The Layout of the FAT

The virtual disk has the following layout:

- block 0 is reserved and can contain any information about the whole file system on the disk (e.g. volume name etc.); block 0 is left free, because the number 0 has a special meaning in the FAT (it is ENDOFCHAIN == 0). However, you can put arbitrary information into these first 1024 bytes on your virtual disk, such as the name of the disk etc.
- block 1 and 2 will be occupied by the FAT (we need 2 blocks, because each entry is a short integer, occupying 2 bytes of disk space, and with 1024 entries in the FAT it needs 2048 bytes, which are 2 blocks of disk space
- block 3 is the root directory: a directory block has special structure, containing a list of directory entries
- The rest of the virtual disk, blocks 4 – 1023, are either data or directory blocks.

Your task is to extend filesys.c with additional interface functions (as outlined below). Also, implement a test program called shell.c that calls functions you have implemented. The files filesys.h and filesys.c are provided to give you the C structures needed for the implementation. You can also create your own C structures to complete the assessment. Don't hesitate to extend or change structures in filesys.h, if you see a need for that in order to support your implementation (you may have to add additional parameters to the file descriptor MyFILE to record additional information, e.g. about the location of a file in the file system etc.).

The complete public interface of the file system for this virtual disk is the following (for each assessment step, you have to implement some of them):

void format()

- creates the initial structure on the virtual disk, writing the FAT and the root directory into the virtual disk

MyFILE * myfopen (const char * filename, const char * mode) ;

- Opens a file on the virtual disk and manages a buffer for it of size BLOCKSIZE, mode may be either “r” for readonly or “w” for read/write/append (default “w”)

void myfclose (MyFILE * stream)

- closes the file, writes out any blocks not written to disk

int myfgetc (MyFILE * stream)

- Returns the next byte of the open file, or EOF (EOF == -1)

void myfputc (int b, MyFILE * stream)

- Writes a byte to the file. Depending on the write policy, either writes the disk block containing the written byte to disk, or waits until block is full

void mymkdir (const char * path)

- this function will create a new directory, using path, e.g. mymkdir (“/first/second/third”) creates directory “third” in parent dir “second”, which is a subdir of directory “first”, and “first is a sub directory of the root directory

void myrmdir (const char * path)

- this function removes an existing directory, using path, e.g. myrmdir (“/first/second/third”) removes directory “third” in parent dir “second”, which is a subdir of directory “first”, and “first is a sub directory of the root directory

void mychdir (const char * path)

- this function will change into an existing directory, using path, e.g. mkdir (“/first/second/third”) creates directory “third” in parent dir “second”, which is a subdir of directory “first”, and “first is a sub directory of the root directory

void myremove (const char * path)

- this function removes an existing file, using path, e.g. myremove (“/first/second/third/testfile.txt”)

char ** mylistdir (const char * path)

- this function lists the content of a directory and returns a list of strings, where the last element is NULL

Assessment Requirements

CGS D3-D1

Task

Implement the function **format()** to create a structure for the virtual disk. Format has to create the FAT and the root directory. Write a test program containing the main() function, and call it "shell.c".

Your test program shell.c should perform the following three actions:

- call format() to format the virtual disk
- transfer the following text into block 0: "CS3026 Operating Systems Assessment"
- write the virtual disk to a file (call it "virtualdiskD3_D1").

Include any header files required, such as filesys.h, in your test program shell.c (do NOT include any .c files!!).

Use the unix command "hexdump" to see what the file contains: hexdump –C virtualdiskD3_D1.

In the downloaded file CS3026_Assessment.zip, you will find the file "virtualdisk" that shows a layout of the virtual disk as acceptable for CGS D3-D1. You must demonstrate that your implementation produces the same or a similar layout (your implementation may differ), with the FAT and entries within the FAT recognizable. It is expected that the hexdump only shows the information written back into the virtual disk and no other clutter. This requires that the virtual disk is properly initialized when formatted().

Additional Information:

Please also read section "**How to start your Project**" for additional information and how to implement the "format()" function.

Submission for CGS D3-D1

In order to achieve at least a D3, your submission must include a make file for building your solution and your solution also must correctly include the required header files. Please describe your implementation in detail in your report: for each statement in function format(), provide an explanation in your report about its purpose. Provide detailed explanations how to run the submission. Provide an explanation what the result of such an execution is: include a hexdump of the virtual disk into the report and provide explanations for it.

Submit a test program, called shell.c, as well as filesys.c, filesys.h, a file virtualdiskD3_D1 and a Makefile that allows your program to be compiled (put files into a directory CGS_D3_D1).

CGS C3-C1

Task

Implement the following interface functions:

- myfopen(),
- myfputc(),
- myfgetc() and
- myfclose().

It is assumed that there is only a root directory and that all files are created there.

Extend your test program shell.c with the following steps:

- create a file “testfile.txt” in your virtual disk: call myfopen (“testfile.txt”, “w”) to open this file
- write a text of size 4kb (4096 bytes) to this file, using the function myfputc():
    - in shell.c, create a char array of 4 * BLOCKSIZE, fill it with text and then write it to the virtual file with myfputc()
- close the file with myfclose()
- write the complete virtual disk to a file “virtualdiskC3_C1”
- test myfgetc():
    - open the file again on your virtual harddisk
    - read out its content with myfgetc() (you may read until the function returns EOF) and, at the same time, print it to the screen
    - write the content to a real file on your real hard disk and call it “testfileC3_C1_copy.txt”

In order to create a recognizable pattern in your hexdump for “testfile.txt”, you may loop through the alphabet over and over again, until the array of size 4*BLOCKSIZE is filled (remember how a string literal “ABCDEFGHIJKLMNOPQRSTUVWXWZ” can be indexed).

Use the unix command hexdump to check the content of your virtual disk:

- hexdump –C virtualdiskC3_C1

Redirect the output of your shell program into a file “traceC3_C1.txt”

- ./shell > traceC3_C1.txt

Submission

For a CGS C3, submit shell.c, filesys.c, filesys.h, the files virtualdiskC3_C1, testfileC3_C1_copy.txt, traceC3_C1.txt and a Makefile that allows your program to be compiled. Put files into a separate directory CGS_C3_C1. In order to achieve a CGS C1, the virtual disk should not show any clutter, only the information you write into it, it has to show the block chain in the FAT for the created file and the content of the file, and in your report, you have to provide detailed comments about the implementation of the required functions (explain it by walking the reader through the implemented statements and provide comments for each of them).

Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

Explanation

When a file is created or opened, a file descriptor has to be created (see filesys.h):

```
typedef struct filedescriptor {
   char        mode[3] ;
   fatentry_t  blockno ;       // block no
   int         pos ;           // byte within a block
   diskblock_t buffer ;
} MyFILE ;
```

This file descriptor can hold one disk block with “diskblock_t buffer”. Read and write operations access this buffer. The attribute “pos” points to the byte in this buffer that is read or written. If read operations go beyond the current buffer content, the next buffer according to the block chain in the FAT has to be loaded. If the buffer becomes full due to write operations, the current buffer has to be written out to disk and a new disk block has to be allocated to the open file – this is done by simply finding the next entry in the FAT with value UNUSED and extending the block chain in the FAT. When the file is closed with fclose(), its length has to be written into the directory entry of this file (situated in the root directory, block 3).

Find the FAT (at 0x400) and the block for the root directory in the hexdump (it starts at 0xc00):

```
00000400  00 00 02 00 00 00 00 00  05 00 06 00 07 00 08 00  |................|
00000410  00 00 ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000420  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000c10  00 00 00 00 00 00 00 00  00 00 00 00 04 00 74 65  |..............te|
00000c20  73 74 66 69 6c 65 2e 74  78 74 00 00 00 00 00 00  |stfile.txt......|
00000c30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

As illustrated, the file 'testfile.txt' was opened for write on the virtual disk, which created it in the root directory (block no 3, starting at 0xc00), and the content of file 'testfile.txt' starts at block no 4. There should be a block chain visible in the FAT table: at 0x400, entry 04 contains 05 00, entry 05 contains 06 00 etc. (please note that the order of bytes is reversed, you see 05 00, and not 00 05). At location 0xc00, the block of the root directory starts. The first two bytes, set to ’01 00’, indicate that this is a dirblock.

Please note: text strings always have a ‘\0’ at the end. C functions such as strcpy() etc. will scan a text string as long as they haven’t found the ‘\0’. For example, the parameter ‘mode’ of function myfopen() is a character array of 3 elements, because it can hold a mode string that can be one or two characters long and has ‘\0’ as its last character.

Indicating End-of-File: the last block of a file may not be filled completely. How do we know where the file ends? You have to store the length of the file in the directory entry. When using **myfgetc** (), it has to be calculated whether all bytes of the files have been read by checking the amount of chars read against the file size stored in the directory entry. If the last byte of the file has been read, then at the next call of

fgetc(), the function has to return EOF (EOF may already be a macro contained in one of the system header files you include in your program). Therefore, when you close your file with myfclose(), you have to update the directory entry with the new file size (number of bytes added to the file).

CGS B3-B1

Task

Add a directory hierarchy to your virtualdisk that allows the creation of subdirectories. Implement the following interface functions:

- mymkdir(char * path) that creates a new directory
- char ** mylistdir (char * path) that lists the content of a directory

Extend your test program shell.c with the following test steps:

- create a directory "/myfirstdir/myseconddir/mythirddir" in the virtual disk
- call mylistdir("/myfirstdir/myseconddir"): print out the list of strings returned by this function
- write out virtual disk to "virtualdiskB3_B1_a"
- create a file "/myfirstdir/myseconddir/testfile.txt" in the virtual disk
- call mylistdir("/myfirstdir/myseconddir"): print out the list of strings returned by this function
- write out virtual disk to "virtualdiskB3_B1_b"

Redirect the output of your shell program into a file "traceB3_B1.txt"

- ./shell > traceB3_B1.txt

Submission

For a CGS B3, submit shell.c, filesys.c, filesys.h, the files virtualdiskB3_B1_a, virtualdiskB3_B1_b, traceB3_B1.txt and a Makefile that allows your program to be compiled. Put files into a separate directory CGS_B3_B1. For a CGS B1, the virtual disk should not show any clutter, only the information you write into it, and in your report, you have to provide detailed comments about the implementation of the required functions (explain it by walking the reader through the most important implemented statements and and explain their purpose). Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

Explanation

A directory can be specified absolute or relative to another directory:

- absolute: "/mydirectory"
- relative: "mydirectory"

A directory may be specified with a path:

- absolute: "/firstlevel/secondlevel/mydirectory"
- relative: "somelevel/somelevelbeneath/mydirectory"

In order to create the directory “mydirectory”, all the subdirectories specified in the path must exist. If you call mymkdir (“/firstlevel/secondlevel/mydirectory”) in your test program shell.c, then the directory hierarchy consisting of Root->firstlevel->secondlevel must exist, before you can create “mydirectory” in the parent directory “secondlevel”. If these directories don’t exist, they have to be created.

Use strtok_r() from the C standard library to tokenize a path string (look up its usage). If you experience a segmentation fault during running the program, remember that pointers have to point to allocated memory and that string literals are allocated in the segment ‘.rodata’ and cannot be manipulated.

CGS A5-A1

Task

Implement the following interface functions:

- mychdir(char * path), using the global variable “currentDir” as specified in filesys.c: a change into a directory will change the variable “currentDir”
- myremove(char * path) removes a file; the path can be absolute or relative
- myrmdir(char * path) removes a directory, if it is empty; the path can be absolute or relative

Change how directories are created:

- add two default entries (as we are used to under Unix etc.):
    - the directory entry “.” points to the directory itself
    - the directory entry “..” points to the parent directory
- allow the creation of a directory relative to the current directory

Change how files are created:

- the function myfopen() can be called using an absolute or relative path in the filename, if the directories specified in the path do not exist, then they have to be created

Demonstrate with your test program shell.c that creating and deleting files and directories works and that results are visible in the hexdumps of the virtual disk. Save intermediate results. You may follow the following steps:

- create a directory “/firstdir/seconddir” in the virtual disk
- call myfopen(“/firstdir/seconddir/testfile1.txt”)
- you may write something into the file
- close the file
- call mylistdir(“/firstdir/seconddir”): print out the list of strings returned by this function
- change to directory “/firstdir/seconddir”
- call mylistdir(“/firstdir/seconddir/”) or mylistdir(“.”) to list the current dir, print out the list of strings returned by this function
- call myfopen(“testfile2.txt, “w”)
- you may write something into the file
- close the file
- create directory “thirddir”

- call myfopen(“thirddir/testfile3.txt, “w”)
- you may write something into the file
- close the file
- write out virtual disk to “virtualdiskA5_A1_a”
- call myremove(“testfile1.txt”)
- call myremove(“testfile2.txt”)
- write out virtual disk to “virtualdiskA5_A1_b”
- call mychdir (thirddir”)
- call myremove(“testfile3.txt”)
- write out virtual disk to “virtualdiskA5_A1_c”
- call mychdir(“/firstdir/seconddir”) or mychdir(“..”)
- call myremdir(“thirddir”)
- call mychdir(“/firstdir”)
- call myrmdir (“seconddir”)
- call mychdir(“/”) or mychdir(“..”)
- call myrmdir(“firstdir”)
- write out virtual disk to “virtualdiskA5_A1_d”

Redirect the output of your shell program into a file “traceA5_A1.txt”

- ./shell > traceA5_A1.txt

Optional work for extra points:

- Try to write a copy function that allows you to copy files from your real hard disk into your virtual disk and vice versa.
- Try to implement a copy and a move function that relocates files within your virtual disk.
- Saveguard the manipulation of the FAT table in a multithreaded application. Introduce a lock variable and store it in block 0 (you can introduce an extra struct for block 0 that contains, among other things, a volume name and this lock variable). The lock variable indicates either a LOCKED or UNLOCKED state of the virtual disk. Use mutexes to change the lock in a thread. Run tests by implementing a multithreaded shell.c.

Submission

For a CGS A5, submit shell.c, filesys.c, filesys.h, the files virtualdiskA5_A1_a .. d, traceA5_A1.txt and a Makefile that allows your program to be compiled. For a CGS A4 .. A2, your solutions have to be of high quality with attempts at providing solutions to extra work as outlined above. For CGS A1, provide additional functionality of your own choosing.

Put files into a separate directory CGS_A5_A1. The virtual disk should not show any clutter, only the information you write into it and in your report, you have to provide detailed comments about the implementation of the required functions (explain it by walking the reader through the most important implemented statements and explain their purpose). Parts missing in your solution may reduce the CGS mark. Try to get with your implementation as far as possible.

Explanation

Look into fileys.h. A directory entry direntry_t uses a char array of 256 bytes for the file or directory name. Very few files may have such a long name. You may reduce the length of this array two 128 or 64 to allow more directory entries. Try to implement directory entries with variable size. Also note: a block

of type dirblock_t contains an array of directory entries, if one of them is “deleted”, it has to be marked as “UNUSED”: look up the type definition for direntry_t – it contains the attribute “unused” that has to be set accordingly. Unused directory entries have to be reused when new files are created. The pointer nextEntry has to be changed to a counter, indicating whether the dirblock is full, and the array of directory entries has to be scanned for an entry set to “UNUSED”.

Submission Procedure

You are required to submit in electronic as form:

- one zip file called **`cs3026_assessment_<your_username>.zip`**

Submit your work, using the following method:

- In MyAberdeen, go to “Assessment”, and do the following:
  - Go to the area after the text explaining the submission procedure
  - click on “CS3026 Assessment (submit here)”
  - click on “Browse My Computer” (this is underneath “2. Assignment Submission”)
  - find your zip file and complete this upload
- As a backup, send an email to m.j.kollingbaum@abdn.ac.uk with the following **exact** subject line:
  - “CS3026 Submission Assessment”
  - Attach to this email your zip file containing your submission.

Please use your University email address to submit your assessment.

All submissions should be documented, and this documentation must include your name and userid at the top of each file. Please submit (a) the complete source code (plus any necessary make files, text files, configuration files etc to compile and run your submission) and (b) a report describing your submission and how to operate your application. Make a zip file containing all this information and send it to the email address above.

You can submit any time before that date. The last electronic submission counts. Standard lateness penalties for submissions apply.

How to start your Project

Two files are provided for this assessment: You will find the files filesys.c and filesys.h containing data structures and basic functions you may use in your implementation:

- The file filesys.h contains specifications of data structures you will use to implement the interface functions for this file system
- The file filesys.c contains the function writeblock() already implemented for you. Implement a function readblock() accordingly. It also contains a function writedisk() that allows you to save the virtual disk to a file on your real harddisk (such a file has to be submitted).

The two basic functions that are directly interacting with the virtual disk are writeblock() and readblock(). All other functions (those you have to implement) are using these to read/write from/to the virtual disk.

The File Allocation Table FAT is declared in filesys.c as an array of fat entries (short integer, 2 bytes):

```
fatentry_t   FAT [MAXBLOCKS] ;
```

The FAT is managed as a local array and held in memory, just like real File Allocation Tables. However, it also has to be stored on the virtual disk. For this, we have to write the content of the FAT out to its reserved disk blocks and update this information when changes to the FAT occur. Changes will occur when blocks are allocated to files: files are newly created or need more disk space.

Remember that the virtual disk has a particular layout: block 0 is reserved and can contain any information about the whole file system on the disk (e.g. volume name etc.), block 1 and 2 reserved for information copied from the FAT, and block 3 is the root directory: a directory block has special structure, containing a list of directory entries. The rest of the virtual disk, blocks 4 – 1023, are either data or directory blocks.

The virtual disk is declared as an array of disk blocks (find this declaration in filesys.h):

```
extern diskblock_t virtualDisk [ MAXBLOCKS ] ;
```

Why is it declared as “extern” in filesys.h? If you look into filesys.c, you will find a definition without the “extern”. Why is that? How do the “extern” declaration and the definition work together?

Look into filesys.h and try to understand the data structure **diskblock_t**: a disk block is declared as a union:

```
typedef union block {
   datablock_t data ;
   dirblock_t  dir  ;
   fatblock_t  fat  ;
} diskblock_t ;
```

As it is declared as a Union, it can be three things at the same time: a data block, a directory block or a fat block.

The union helps us to access a block in different ways. If we regard it as a datablock, we just see an array of bytes:

```
typedef Byte datablock_t [ BLOCKSIZE ] ;
```

If we regard it as a directory block, we see a more elaborate structure:

```
typedef struct dirblock {
   int isdir ;
   int nextEntry ;
   direntry_t entrylist [ DIRENTRYCOUNT ] ;
} dirblock_t ;
```

If we regard it as a block for storing FAT entries, we see an array of FAT entries:

```
typedef fatentry_t fatblock_t [ FATENTRYCOUNT ] ;
```

A block can now be accessed in different ways. For example, if we want to initialize a directory block to '\0', we can first regard it as a pure disk block, which is an array of BLOCKSIZE bytes and set all elements to 0:

```
diskblock_t block;
for ( int i = 0; i < BLOCKSIZE; i++) block.data[i] = '\0' ;
```

The same block can then be used as, for example, a directory block (where everything is set to '\0'):

```
dirblock_t blockAsDir = block.dir;
```

Implementing the format() Function

For the implementation of the format() function, you have to do three things:

- initialize block 0:
  - use the variable "diskblock_t block", defined as a local variable in format():
    - If we want to initialize a disk block (set all its bytes to '\0'), we may approach this by treating the block as a datablock:

```
diskblock_t block;
for ( int i = 0; i < BLOCKSIZE; i++) block.data[i] = '\0' ;
```

  - use strcpy(block.data, "CS3026 blabla etc") to copy the recommended text into block 0
  - use writeblock(&block, 0) to write this block to the virtual disk

- initialize the FAT and write it to the virtual disk, starting at block 1
  - set all entries of the FAT to UNUSED
  - initialize the FAT with its own block chain
    - if the FAT needs two blocks on the disk, then it will occupy block 1 and 2, therefore there is always a chain "block1->block2" for the FAT itself:
    - this is expressed by the following assignments
      FAT[1] = 2 ;

FAT[2] = ENDOFCHAIN ;

- If you decide to create a larger disk with more blocks, then you need a larger FAT, and the chain for the FAT is longer.
- Implement a function copyFAT() that copies the content of the FAT into one or more blocks, then write these blocks to the virtual disk

- Create one directory block for the root directory
  - A directory block has an array of directory entries (which can be files and directories) and a "nextEntry" pointer that points to the next free list element for a directory entry

```
typedef struct dirblock {
   int isdir ;
   int nextEntry ;
   direntry_t entrylist [ DIRENTRYCOUNT ] ;
} dirblock_t ;
```

  - Initialize "isDir" to 1 (this is a directory block)
  - Initialize the "nextEntry' index to 0 (first element in the entrylist)
  - Write this block to the virtual disk
  - Initialize the block chain for the root directory:
    - This is expressed by the following assignment:
      FAT[3] = ENDOFCHAIN
    - As each change of the FAT has to be copied to the virtual disk, you may have done this already above in combination with initializing the block chain of the FAT itself
  - Set "rootDirIndex" to the block number of the root directory

We assume the following definitions (see filesys.h): ENDOFCHAIN set to 0, UNUSED set to -1, and EOF set to -1.

A hexdump of your virtualdiskD3_D1 may look like the following:

```
00000000  43 53 33 30 30 38 20 4f  70 65 72 61 74 69 6e 67  |CS3026 Operating|
00000010  20 53 79 73 74 65 6d 73  20 41 73 73 65 73 73 6d  | Systems Assessm|
00000020  65 6e 74 20 32 30 31 32  00 00 00 00 00 00 00 00  |ent 2014........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 02 00 00 00 00 00  ff ff ff ff ff ff ff ff  |................|
00000410  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000c00  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00100000
```

Block 0 occupies addresses 0x0 until 0x3ff (decimal 1023). Block 1 (0x400 – 0x7ff, decimal 1024 - 2047) and block 2 (0x800 – 0xbff, decimal 2048 - 3071) are occupied by the FAT. Block 3 (0xc00 – 0xfff, decimal 3072 - 4095) is occupied by the root directory.

You can see the FAT at 0x400:

- the first '00 00' is the value in FAT entry for block 0 (FAT[0]), set to ENDOFCHAIN

- the next '02 00' is the value in FAT entry for block 1 (FAT[1]), set to 2 (this is the index of FAT[2])
    - note that we use Intel processors, therefore the sequence of bytes for an integer 2 is '02 00' and not '00 02'
- the next '00 00' is the value in FAT entry for block 2 (FAT[2]), set to ENDOFCHAIN (the end of the block chain for the FAT table)
- the next '00 00' is the value in FAT entry for block 3 (FAT[3]), set to ENDOFCHAIN (the root directory)
- the rest remains UNUSED, which is -1 or 0xff

You can see the block for the root directory at 0xc00

- the first '01 00' is the value stored in "isDir" of **`dirblock_t`**.
- the rest of the block is initialized to '\0'

The File Allocation Table

The FAT shows which blocks belong to a file or a directory. If a file grows, it may need more space and additional blocks have to be allocated to the file. The allocation of blocks is done via FAT itself – a block chain has to be created. The same is true for a directory – if it has more entries than fit on a single block, additional blocks have to be allocated.

In filesys.c, the file allocation table is is declared as:

```
fatentry_t FAT [MAXBLOCKS]
```

The type fatentry_t is declared as a short integer (see filesys.h), therefore the array FAT is an array of short integers (2 bytes). The file allocation has to be stored in the virtual disk, we therefore need a special disk block structure for the FAT. The basic idea is to copy the FAT entry-by-entry into a diskblock, before the disk block is stored in the virtual disk. The structure of a disk block for the FAT will therefore be an array of FAT entries. In filesys.h, a declaration for this block structure exists:

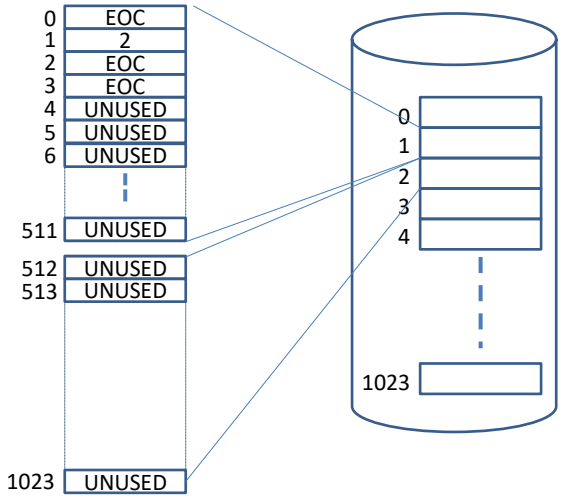
```
typedef fatentry_t fatblock_t [FATENTRYCOUNT]
```

We define a disk block for holding FAT entries. A diskblock of size 1024 bytes can hold only part of the FAT: if each FAT entry needs 2 bytes, then the whole FAT is of size 2048, therefore we need two blocks to store the whole FAT

When a file is opened, the file descriptor MyFILE points to a buffer of BLOCKSIZE bytes. The functions myfputc() and myfgetc() operate on this buffer:

- in case of write operations: when this buffer becomes full, it has to be written (copied) to the virtual disk and a new block has to be allocated (an UNUSED block as indicated in the FAT)
- in case of read operations: the first block of the file has to be loaded (copied) from the virtual disk when opened; each read pushes a position pointer to the end of the buffer, when it becomes BUFFERSIZE then the next block from the virtual disk has to be loaded; for this the chain in the FAT has to be followed to find the next block of the file

The integer array FAT has to be stored on the disk as well: it will occupy a number of blocks. Usually, the first couple of disk blocks are reserved for the FAT. The FAT has to be loaded, when the virtual disk is "mounted" and written back to the disk, when the disk is "dismounted". As the FAT stores block chains for all chained blocks, it will contain a small block chain for itself as well.



Read and Write of Virtual Disk

Please use filesys.h and filesys.c as a starting point. In filesys.h, the virtual disk is an array of blocks the size of BLOCKSIZE. There are only two functions that access this virtual disk directly:

- **`writeblock (diskblock_t * block, int blockIndex)`**: write a complete block to the disk
- **`readblock (diskblock_t * block, int blockIndex)`** : reads a complete block from this virtual disk

For moving blocks from / to the virtual disk, two methods can be used:

- byte-wise copying with a for loop:

```
for( bytePos = 0; bytePos < BLOCKSIZE; bytePos++ )
          virtualdisk[blockIndex].data[bytePos] = block->data[bytePos];
```

- use the system call memmove() :

```
memmove ( virtualdisk[blockIndex].data, block->data, BLOCKSIZE ) ;
```