CSC 360 Summer 2023
Assignment 4

Programming Environment

For this assignment, your code must work in the JupyterHub environment (`https://jhub.csc.uvic.ca`). Even if you already have access to your own Unix system, we recommend you work as much as possible with the JupyterHub environment. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration.

Individual Work

This assignment is to be completed by each individual student (i.e. no group work). You are encouraged to discuss aspects of the problem with your classmates, however **sharing of code is strictly forbidden**. If you are unsure about what is permitted or have other questions regarding academic integrity, please ask the instructor. Code-similarity tools will be run on submitted programs. Any fragments of code found or generated on the web and is used in your solution must be properly cited (citation in the form of a comment in your code).

Assignment Objectives

Write three C programs implementing operations on a 360fs file system image (sample images will be provided for you). The operations are as follows:

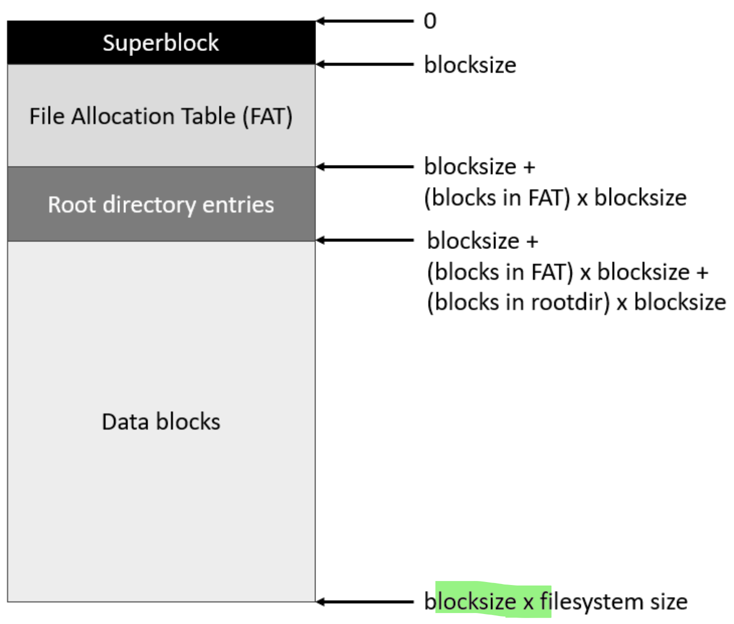
1. printing information about a disk image (`stat360fs.c`)
2. printing a directory listing of files in a disk image (`ls360fs.c`)
3. outputting the contents of a file in disk image to standard output (`cat360fs.c`)

As an optional part of the assignment (not worth marks), you can write the fourth operation which copies a file on your local machine to a disk image (`stor360fs.c`).

The File System 360fs

Before describing your tasks, you first need to know more about the file system for this assignment. It has four major components as shown in the diagram below. The arrows indicate byte offsets from the start of the file system. This file system only has a root directory, so you don't have to worry about implementing subdirectory files.

The size of the file system disk image (in bytes) is the product of its blocksize (always some number of bytes having a power of 2) and the file system size (always expressed as some number of blocks, where this number may not be a power of 2). For example, if the blocksize is 512 bytes and the number of blocks is 5120, then the disk-image size of the file system is 2,621,440 bytes.


Superblock
File Allocation Table (FAT)
Root directory entries
Data blocks
0
blocksize
blocksize + (blocks in FAT) x blocksize
blocksize + (blocks in FAT) x blocksize + (blocks in rootdir) x blocksize
blocksize x filesystem size

The superblock in the first area is reserved for critical file system metadata. The following table shows the layout of this block as well as the value of each field in the provided disk image `disk02X.img`. Everything else from the last entry in the superblock to the end of the first disk block is filled with zeroes.

Description	**Size**	**Value in `disk02X.img`**
file system identifier	8 bytes	360fs
blocksize	2 bytes	0x0100
file system size (in blocks)	4 bytes	0x00000bb8
block where FAT starts	4 bytes	0x00000001
number of blocks in FAT	4 bytes	0x0000002f
block where root directory starts	4 bytes	0x00000030
number of blocks in root directory	4 bytes	0x00000010

The File Allocation Table (FAT) is stored in the second section of the disk image and always starts at the second block. Before describing the FAT, we must first describe the directory entries contained in the root directory section.

Each directory entry is 64 bytes in size, and the maximum number of these entries within the root directory is fixed. For 360fs, this number is 64. Each file that exists in the file system will have its own directory entry. The layout of every directory entry is as follows:

Description	**Size**
status	1 byte
starting block	4 bytes
number of blocks in file	4 bytes
file size (in bytes)	4 bytes
create time	7 bytes
modify time	7 bytes
filename	31 bytes
unused (set to 0xff)	6 bytes

Each field of a directory entry is described as follows:

- **status:** Bit mask describing the status of the file. Only three bits are used. Bit 0 (i.e. least significant bit) is false if the entry is available, true otherwise. Bit 1 is set to true if the entry is for a normal file. Bit 2 is set to true if the entry is for a directory (but you don't have to worry about implementing subdirectories). Therefore, bits 1 and 2 cannot both be true (an entry is either a normal file or a directory but not both).
- **starting block:** The block in the disk image that is the first block of the file corresponding to the directory entry.
- **number of blocks in file:** Total number of blocks in the file.
- **file size (in bytes):** Size of the file in bytes. Note that file size must be less than or equal to the number of blocks in the file system multiplied by the file system block size.
- **create time / modify time:** Date and time when the file was created / modified. The date / time stored in a directory entry assumes having the form YYYYMMDDHHMMSS where two bytes are used to store the year and one byte each to store other values (month, day, hour, minute, second).
- **filename:** a null-terminated string (the largest file name is 30 chars). The characters accepted in filenames are alphabetic (upper and lower case), digits (0-9) and the underscore character.

We can now return to the File Allocation Table. The concept of a FAT has been around for nearly forty years and has some similarities to an array-based implementation of a linked list (the FAT itself is treated as an array of 4-byte integers). In order to find out what blocks belong to a file:

1. The directory entry for the file is located and the starting block is read from that entry. Let S be the file's starting block.
2. Block S in the Data Block section of the file system is then read.
3. To find the next block in the file, we look at the value in entry S of the FAT. Let T be the value in this entry (i.e. $T = \text{FAT}[S]$). If T's value does not indicate end-of-file, then T is the next block in the file, so we set S to T and go back to step 2. Otherwise, we stop. We would not expect T's value to indicate the block as available or as reserved.

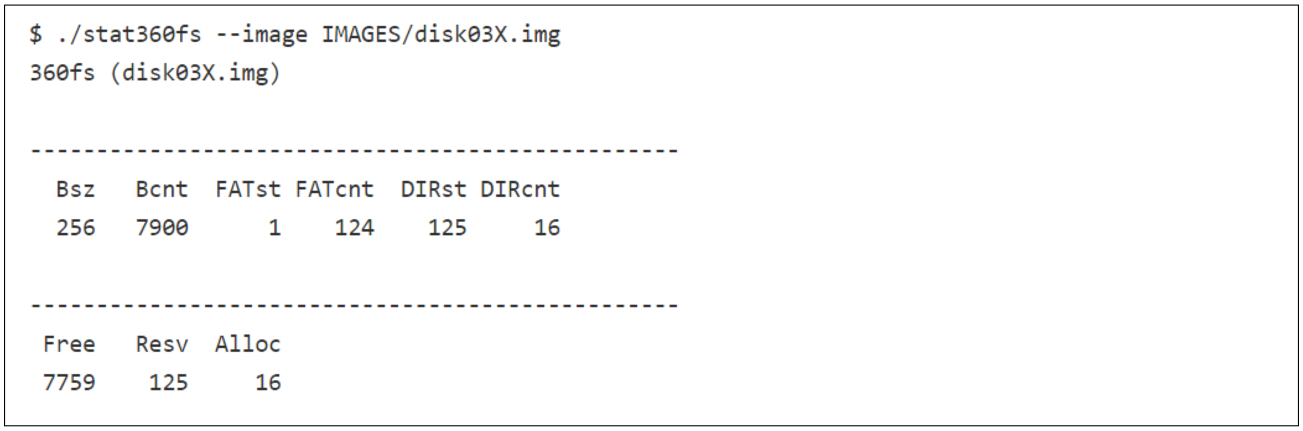
FAT entries are four bytes long (32 bits). Therefore, if the file system blocks size is 256 bytes, each block in the FAT will contain 64 entries. If the blocksize is 512 bytes, then each block will contain 128 FAT entries. There are as many FAT entries as there are blocks in the entire file system. FAT entries may contain the following values indicating the status of its corresponding file system block:

Value	**Meaning**
0x00000000	This file system block is available
0x00000001	This file system block is reserved (part of superblock or FAT)
0x00000002 - 0xFFFFFF00	This file system block is allocated to some file
0xFFFFFFFF	This is the last block in a file

The final section of the disk is made up of the data blocks for files, and we would expect this to be the largest section of the disk image.

Task 1: stat360fs.c

Write a program `stat360fs.c` which displays information about a 360fs file system image. Among other things, your code must read in the superblock of the disk image and use that information to read entries in the FAT. The following is the program's output for the provided `disk03X.img`. Please follow this format for the output.



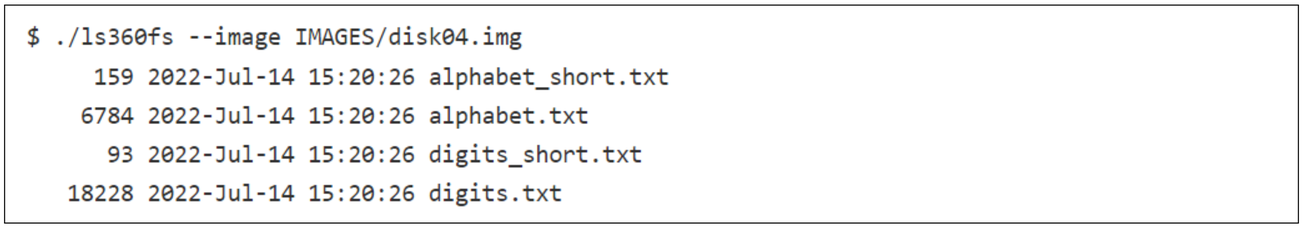
```
$ ./stat360fs --image IMAGES/disk03X.img
360fs (disk03X.img)

-------------------------------------------------
  Bsz   Bcnt  FATst FATcnt  DIRst DIRcnt
  256   7900      1    124    125     16

-------------------------------------------------
 Free   Resv  Alloc
 7759    125     16
```

Task 2: ls360fs.c

Write a program `ls360fs.c` that displays the root directory listing for a 360fs file system image. The following is an example of the program's output for `disk04.img` in the distributed files.

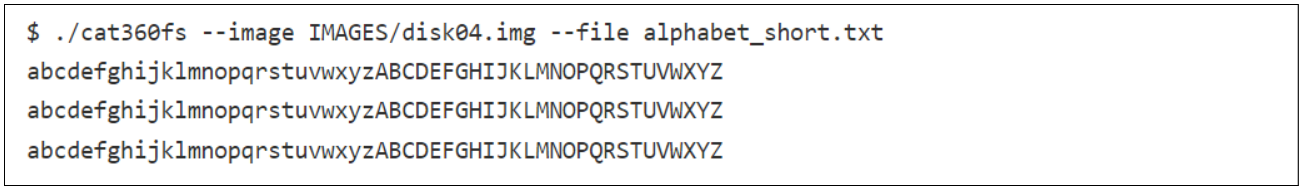


```
$ ./ls360fs --image IMAGES/disk04.img
     159 2022-Jul-14 15:20:26 alphabet_short.txt
    6784 2022-Jul-14 15:20:26 alphabet.txt
      93 2022-Jul-14 15:20:26 digits_short.txt
   18228 2022-Jul-14 15:20:26 digits.txt
```

On each line, eight digits are used for the file size (in bytes). Although the order of lines in your output may vary from what is expected, each expected line must appear in the output and no others. Please use the line format as shown.

Task 3: cat360fs.c

Write a program `cat360fs.c` that copies a file from a disk image to the standard output stream of the console. If the specified file is not found in the root directory, print a "file not found" message on a single line and exit the program.



```
$ ./cat360fs --image IMAGES/disk04.img --file alphabet_short.txt
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

(Optional) Task 4: `stor360fs.c`

Write a program **`stor360fs.c`** that copies a file from your local system to the root directory of a specified disk image. A message is printed if the local system file cannot be found, if the file already exists on the disk image, or if there is not enough room to store the file on the disk image. The following shows a text file named "foo.txt" containing the text "Foo!" onto a disk image.

```
$ ./stor360fs --image disk03.img --file foo.txt --source foo.txt
$ ./ls360fs --image disk03.img
     159 2022-Jul-16 19:58:56 alphabet_short.txt
      93 2022-Jul-16 19:58:56 digits_short.txt
       5 2022-Jul-17 13:20:32 foo.txt
$ ./stor360fs --image disk03.img --file foo.txt --source foo.txt
file already exists in the image
$ ./cat360fs --image disk03.img --file foo.txt | diff ./foo.txt -
$ # no output from diff -- that's good as it means the files are identical
```

Implementation Detail Hints

A file called `function_and_utilities.txt` is provided for you which contains some code that you might find useful in your implementation.

Byte ordering

Some content of superblock and directory entries are **integers** (either two-byte or four-byte). However, the order in which an integer's bytes are written to a file depends on the **endianness** of the host computer.

When reading / writing bytes on `jhub.csc.uvic.ca`, which uses **little-endian**, bytes in a short (two-byte integers) and bytes in an integer (four-bytes) are in reverse order. However, we are used to reading bytes from left to right (**big-endian**) and integers in the file system are stored in big-endian.

To ensure this, we can use the `<arpa/inet.h>` functions `htons()`, `htonl()`, `ntohs()`, and `ntohs()` as needed to convert data to and from the disk image (examples in `function_and_utilities.txt`). Data should be in host computer order for calculations but in disk order for storage. All disk images provided for you in the assignment have been stored in big-endian order.

Packed Structs

One of the files provided is `disk.h`. Among other things in this header file, such as a large number of `#define` directives, there are two types: `superblock_entry_t` and `directory_entry_t`. These can be used to read superblocks and directory entries from the disk image.

At the end of each type declaration, there is a strange compiler directive. It is needed in order to override the compiler's normal layout of fields within a struct. Normally, the compiler ensures fields are always aligned on a word boundary (one word = four bytes). There are good reasons for this, having to do with the way hardware retrieves data from RAM. For our disk image though, such alignment causes problems as we want fields to appear precisely where we put them in the structure. In order to ensure this precise layout, we use the `packed` directive. Now we can directly read and write instances of `superblock_entry_t` and `directory_entry_t` using C file routines.

File Routines

You will make heavy use of the library functions `fopen()`, `fseek()`, `fread()`, and `fwrite()`. The the provided text file `function_and_utilities.txt` includes several use cases that you may find useful.

Submission

Submit the three files `stat360fs.c`, `ls360fs.c`, and `cat360fs.c` containing your solutions for Task 1, Task 2, and Task 3 respectively.