CS494/CS594 -- Lab 2: Implementing a File Allocation Table

- CS494/CS594
- Fall, 2021
- James S. Plank
- This file: **http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/labs/Lab-2-FAT/**
- Lab Directory: **/home/jplank/cs494/labs/Lab-2-FAT**

In this lab, you are going to manage emulated disks with a File Allocation Table. It's a fun lab, so buckle up!

Xxd, and if you want to use VI for binary

In general, the program **xxd** can be really handy for looking at, and setting binary values. "man xxd".

As for VI: This info came from Connor Minton, a CS360 student in 2015:

- Open the file with the "-b" option, as in "vim -b foo.dat"
- Type the command ":%!xxd"
- Edit the hex part (editing the ascii representation will not affect the output)
- When you're done, type the command ":%!xxd -r"
- Save and exit
- Here is a link that shows an example: http://makezine.com/2008/08/09/edit-binary-files-in-vi/

Jdisk: A disk emulator

Before I describe the actual lab, I'll discribe the disk emulator that you will use.

I have written a library called "jdisk." Its interface is in **jdisk.h**, and the code is in **jdisk.c**. Here's the header file.

```
#ifndef _JDISK_
#define _JDISK_

#define JDISK_SECTOR_SIZE (1024)
#define JDISK_DELAY (1)

void *jdisk_create(char *fn, unsigned long size);
void *jdisk_attach(char *fn);
int jdisk_unattach(void *jd);

int jdisk_read(void *jd, unsigned int lba, void *buf);
int jdisk_write(void *jd, unsigned int lba, void *buf);

unsigned long jdisk_size(void *jd);
long jdisk_reads(void *jd);
long jdisk_writes(void *jd);

#endif
```

It exports a disk-like interface on top of standard unix files. To use the library, you create a jdisk with **jdisk_create()**. The disk will be composed of sectors whose sizes are JDISK_SECTOR_SIZE (1K), and the disk's size will be the specified size (which must be a multiple of JDISK_SECTOR_SIZE). The jdisk is then held in the file specified by the file name. When **jdisk_create()** returns, the file will be created and it will have **size** bytes, whose contents are unspecified. It returns a **void *** to you, which is your handle on the jdisk.

You can "attach" to an existing jdisk file with **jdisk_attach()**. (There is no structure to a jdisk file -- its size simply has to be a multiple of JDISK_SECTOR_SIZE, so you can attach to any file of the proper size).

jdisk_unattach() closes the file and frees the memory associated with a jdisk. You don't have to call **jdisk_unattach()** -- when a process dies, the jdisk will be fine.

All I/O from and to the jdisk must be done on whole sectors. So, for example, **jdisk_read()** takes a sector number ("LBA" stands for "logical block address") and a pointer to JDISK_SECTOR_SIZE bytes, and it reads the sector from the jdisk. Similarly **jdisk_write()** writes the bytes to disk. Sectors are numbered consecutively from 0. The reads and the writes are performed immediately via **lseek()** and **read()/write()** system calls.

The last three procedures return information -- the size of the jdisk (in bytes), and how many reads/writes have occured since calling **jdisk_create()** or **jdisk_attach()**.

I have written a program to let you mess with jdisks, called **jdisk_test.c**. You call it one of three ways:

```
usage: jdisk_test CREATE disk-file size
       jdisk_test W disk-file string|hex       seek_ptr string/hex
```

```
jdisk_test R disk-file string|hex|bytes seek_ptr nbytes
```

The first lets you create a jdisk. The other two allow you to read from and write to the jdisk. You *don't* have to work on sector boundaries with **jdisk_test** -- it does all of that magic for you. So, for example, the following will create a jdisk with ten sectors, and then do some writing and reading. When you write, you write bytes, and there is no null character, unless you put it there:

```
UNIX> ./jdisk_test CREATE test.jdisk 10240
UNIX> ./jdisk_test W test.jdisk string 0 James-Plank
UNIX> ./jdisk_test R test.jdisk string 0 11
James-Plank
UNIX> xxd -g 1 test.jdisk | head -n 2
0000000: 4a 61 6d 65 73 2d 50 6c 61 6e 6b 00 00 00 00 00  James-Plank.....
0000010: 00 00 00 00 00 00 00 80 07 40 00 00 00 00 00 00  .........@......
UNIX> ./jdisk_test W test.jdisk string 5 ABCDEFGHIJKLMNOP
UNIX> ./jdisk_test R test.jdisk string 0 11
JamesABCDEF
UNIX> ./jdisk_test R test.jdisk string 2 11
mesABCDEFGH
UNIX> ./jdisk_test R test.jdisk hex 2 11
 6d 65 73 41     42 43 44 45     46 47 48
UNIX> xxd -g 1 test.jdisk | head -n 2
0000000: 4a 61 6d 65 73 41 42 43 44 45 46 47 48 49 4a 4b  JamesABCDEFGHIJK
0000010: 4c 4d 4e 4f 50 00 00 80 07 40 00 00 00 00 00 00  LMNOP....@......
UNIX>
```

You don't have to read from or write to the beginning, and you can read and write across sector boundaries. If you write hex, it simply takes a stream of two-digit hex characters.

```
UNIX> ./jdisk_test W test.jdisk hex 1020 6d657341424344454647 48
UNIX> ./jdisk_test R test.jdisk hex 1023 3
 41 42 43
UNIX> ./jdisk_test R test.jdisk string 1023 3
ABC
UNIX> xxd -s 1020 -l 16 -g 1 test.jdisk
00003fc: 6d 65 73 41 42 43 44 45 46 47 48 00 00 00 00 00  mesABCDEFGH.....
UNIX>
UNIX>
UNIX>
```

As you can see, you can also use **xxd** to see what's going on with the disk. Sometimes it's easier than **jdisk_test**, and sometimes it's not. You'll get used to it.

If you specify the "bytes" flag when you are reading with **jdisk_test**, it will emit the raw bytes. I'll show you a use of that below.

If you read from an uninitialized part of a disk, the bytes can have any values. They don't have to be zeros. You should compile and play with **jdisk_test** (there's a makefile in the lab directory).

Jdisks can be as big as 2^{32} sectors. Thus, LBA's can be any unsigned int.

What is a File Allocation Table (FAT)

A File Allocation Table is a way of implementing a file system on a disk. Any textbook on Operating Systems will have a description, but here is mine, which will match the lab. With a FAT, you partition the disk sectors into two sets of sectors:

- The data sectors.
- The FAT.

With this organization a file is represented very simply -- by its starting sector. This is regardless of the file's size.

Now, every data sector has a corresponding *link* in the FAT. If a sector belongs to a file, then the link will hold the identity of the next sector in the file. If the sector is the last sector in the file, then the link will specify that. If a sector doesn't belong to a file, then it will be on a "free list," and its link field will specify the next sector on the free list.

So, if you want to, for example, read a file that starts at sector *i*, you go ahead and read sector *i* and then read its link. That will tell you the next sector to read, and then you'll look at that sector's link field to continue, and so on.

File allocation tables are nice, because the tables are small, compared to the data sectors, and it typically doesn't cost much to have most of the table in memory.

The Exact Format of Our Disks

In this lab, our disks have a very specific format. When there are *D* data sectors on the disk, the FAT will be composed of *D+1* entries. Each entry will be an **unsigned short**. The first entry identifies the first sector on the free list. The link field for that entry will be the next sector on the free list, and so on.

The way links identify sectors is as follows. Suppose the first data sector is sector *S*. If a link's value is *L*, then the link is pointing to sector *S+L-1*.

Converesly, data sector *X* corresponds to link *X-S+1*.

The FAT occupies the first *S* sectors of the disk. The value of *S* is the smallest number that can hold *D+1* **shorts**.

The values of *D* and *S* are uniquely determined by the size of the disk. Specifically, *D* is the largest possible value such that *D+S* is less than or equal to the size of the disk. Since *S* is a function of *D*, this definition is sufficient.

Let me give some concrete numbers to help. In this lab, we are going to affix the sector size to be 1024 bytes. Suppose that my disk is 5000 sectors. Then *D* will equal 4990 and *S* will be ten. This is because *D+1* links take (4990+1)*2 = 9982 bytes, which requires 10 sectors. If I tried to set *D* to 4991, then I'd still need 10 sectors for the FAT, and I wouldn't have enough sectors for both the FAT and the data.

Now, suppose that sector *X* is on the free list. Then the link for that sector is entry *X-S+1* in the FAT. If the value of that link is 0, then sector *X* is the last sector on the free list. Otherwise, the link points to the next sector on the free list.

Now, suppose now that sector *X* belongs to a file. Again, the link for that sector is entry *X-S+1* in the FAT. If the value of that link is any value except 0 or *X-S+1*, then the value points to the next sector in the file. If the value is 0 or *X-S+1*, then the sector is the last sector in the file. If the value of the link is 0, then the entire sector belongs to the file. Otherwise, only part of the sector belongs to the file, and you need to look at the last byte of the sector to determine how many bytes belong to the file:

- If the last byte is 0xff, then 1023 bytes of the sector belong to the file.
- Otherwise, the last byte contains the high order byte of the number of bytes that belong to the file, and then penultimate byte of the sector contains the low order byte.

For example, if the last byte is 0x2 and the penultimate byte is 0x91, then the number of bytes that belong to the file is 0x291, which is equal to 657. The unused bytes are simply unused -- they don't belong to any file.

I will have some more concrete examples of the last sector in a file below.

You are to write the program **FATRW**. You call it in one of two ways:

```
FATRW diskfile import input-file
FATRW diskfile export starting-block output-file
```

Diskfile should be a jdisk file that has been formatted as a FAT disk (more on that below). When you call "**FATRW diskfile import intput-file**", then your program's goal is to store the file **input-file** on the disk. Your program will determine the file's size, and then determine whether the disk has enough free blocks to store the file. If not, it prints an error. If it does have enough free blocks, then it goes ahead and stores the file, modifying the FAT appropriately. It prints the starting block of the file, then it prints the results of **jdisk_reads()** and **jdisk_writes()**. See below for an example.

If you call "**FATRW** diskfile **export** starting-block output-file", then your program finds the file starting at block *starting-block*, and writes it to *output-file*. Again, it prints the results of **jdisk_reads()** and **jdisk_writes()**.

You can use the program **FAT** in the lab directory to format jdisk files. Here's an example, where I create a jdisk with 10 sectors, and format it as a FAT:

```
UNIX> ./jdisk_test CREATE t1.jdisk 10240
UNIX> ./FAT
usage: FAT diskfile format
       FAT diskfile import input-file
       FAT diskfile export starting-block output-file
       FAT diskfile report
UNIX> ./FAT t1.jdisk format
Size(sectors): 10  FAT-Sectors: 1   Data-Sectors: 9
UNIX>
```

The FAT will be the first 20 bytes of sector 0. This is because *D* equals 9, and the FAT is composed of *D+1* entries, which are **unsigned shorts**. *S* equals one. Let's take a look with **jdisk_test** and **xxd**. Here is where **xxd** is totally superior, because it lets you group two bytes at a time and print them in little endian:

```
UNIX> ./jdisk_test R t1.jdisk hex 0 20
 01 00 07 00     09 00 08 00     02 00 04 00     05 00 03 00
 06 00 00 00
UNIX> xxd -g 2 -e -l 20 t1.jdisk
00000000: 0001 0007 0009 0008 0002 0004 0005 0003  ................
00000010: 0006 0000                                ....
UNIX>
```

Because the first short is 1, the first entry in the free list has a value of 1, which corresponds to disk block *S+1-1* = 1. Therefore, the first free block is block 1. The link for block one is entry *1-S+1* = 1. That link has a value of 0x0007, which corresponds to disk block *S+7-1* = 7. So the next free block is block 7. Its link field is 0x0003, so the next free block is 3. The rest of the free blocks are in this order: 8, 6, 5, 4, 2, 9. The link field of block 9 is 0, so that's the end of the free list.

You'll note that all blocks are free. That's because I just formatted the disk. Now, I'm going to import some files. Each of these files is named for how big it is in bytes.

```
UNIX> ls -l File-*
-rw-r--r--  1 jplank staff    10 Jan 27 16:07 File-0010.txt
-rw-r--r--  1 jplank staff   591 Jan 27 16:04 File-0591.txt
```

```
-rw-r--r--  1 jplank staff  1023 Jan 27 16:04 File-1023.txt
-rw-r--r--  1 jplank staff  1024 Jan 27 16:04 File-1024.txt
-rw-r--r--  1 jplank staff  2047 Jan 27 16:06 File-2047.txt
-rw-r--r--  1 jplank staff  2048 Jan 27 16:05 File-2048.txt
-rw-r--r--  1 jplank staff  3029 Jan 27 16:01 File-3029.txt
UNIX> ./FATRW t1.jdisk import File-0010.txt
New file starts at sector 1
Reads: 1
Writes: 2
UNIX> ./FATRW t1.jdisk import File-2048.txt
New file starts at sector 7
Reads: 1
Writes: 3
UNIX> ./FATRW t1.jdisk import File-2047.txt
New file starts at sector 8
Reads: 1
Writes: 3
UNIX> ./FATRW t1.jdisk import File-3029.txt
New file starts at sector 5
Reads: 1
Writes: 4
UNIX> ./FATRW t1.jdisk import File-3029.txt
Not enough free sectors (1) for File-3029.txt, which needs 3.
UNIX>
```

Let's take a look at the FAT table:

```
UNIX> ./jdisk_test R t1.jdisk hex 0 20
 09 00 01 00     02 00 00 00     02 00 04 00     06 00 03 00
 06 00 00 00
UNIX> xxd -g 2 -e -l 20 t1.jdisk
00000000: 0009 0001 0002 0000 0002 0004 0006 0003  ................
00000010: 0006 0000                                ....
UNIX>
```

The free list starts with sector 9, whose link field is 0, so that's the only sector on the free list. When we imported the 10-byte file, it went to sector one. We can confirm that one pretty easily:

```
UNIX> cat File-0010.txt
Jim Plank
UNIX> ./jdisk_test R t1.jdisk string 1024 10
Jim Plank

UNIX> xxd -s 1024 -l 10 -g 1 t1.jdisk
00000400: 4a 69 6d 20 50 6c 61 6e 6b 0a                    Jim Plank.
UNIX>
```

(The last character is a newline -- ASCII for 0xa. **xxd** prints a . for whitespace, so don't let that confuse you).

The link field for block 1 has a value of 1, which means that:

- It is the last block of the file.
- The number of bytes in the block that belong to the file need to be determined by looking at the last byte of the sector, and maybe the byte before that. Let's look at those bytes:

```
UNIX> ./jdisk_test R t1.jdisk hex 2046 2
 0a 00
UNIX> xxd -s 2046 -l 2 -g 1 t1.jdisk
00007fe: 0a 00                                             ..
UNIX>
UNIX>
```

The last byte is not 0xff, so we build the number of bytes as 0x00 * 256 + 0xa = 10. That's right. Yay! The file **File-2048.txt** takes exactly two sectors. It starts at sector 7, whose link field is 3, and thus continues with sector 3, whose link field is 0. That means that sector 3 is the last sector, and all of it belongs to the file. Let's verify:

```
UNIX> echo '7 1024 * p' | dc
7168
UNIX> ./jdisk_test R t1.jdisk bytes 7168 1024 > tmp.txt
UNIX> echo '3 1024 * p' | dc
3072
UNIX> ./jdisk_test R t1.jdisk bytes 3072 1024 >> tmp.txt
UNIX> diff tmp.txt File-2048.txt
UNIX> ./FATRW t1.jdisk export 7 tmp.txt
Reads: 3
Writes: 0
UNIX> diff tmp.txt File-2048.txt
UNIX>
```

The file **File-2047.txt** takes two sectors, but only 1023 bytes of the second sector. It starts at sector 8, and its link is 6, so its second sector is 6. Sector 6's link is 6, so it is indeed the last sector, and we need to look at the sector's last byte to determine the number of bytes that belong to the file:

```
UNIX> echo '7 1024 * p' | dc
7168
UNIX> ./jdisk_test R t1.jdisk hex 7167 1
 ff
```

```
UNIX>
```

That tells us that 1023 bytes belong to the file -- that's correct!

The last file consumes three sectors. It begins with sector 5, so a look at the FAT confirms that the sectors are 5, 4 and 2. We need to look at the last two bytes of sector 2 to determine how many bytes belong to the file:

```
UNIX> echo '3 1024 * p' | dc
3072
UNIX> ./jdisk_test R t1.jdisk hex 3070 2
 d5 03
UNIX> echo '13 16 * 5 + p' | dc
213
UNIX> echo '3 256 * 213 + p' | dc
981
UNIX> echo 981 2048 + p | dc
3029
UNIX>
```

The answer is 981, and 981 + 2048 = 3029, which is the size of **File-3029.txt**. Let's confirm that the bytes are the same:

```
UNIX> echo '5 1024 * p' | dc
5120
UNIX> ./jdisk_test R t1.jdisk bytes 5120 1024 > tmp.txt
UNIX> echo '4 1024 * p' | dc
4096
UNIX> ./jdisk_test R t1.jdisk bytes 4096 1024 >> tmp.txt
UNIX> ./jdisk_test R t1.jdisk bytes 2048 981 >> tmp.txt
UNIX> diff tmp.txt File-3029.txt
UNIX>
```

Booyah!

Finally, we can use FATRW to export the files and compare them with the originals:

```
UNIX> ./FATRW t1.jdisk export 1 tmp.txt
Reads: 2
Writes: 0
UNIX> diff tmp.txt File-0010.txt
UNIX> ./FATRW t1.jdisk export 8 tmp.txt
Reads: 3
Writes: 0
UNIX> diff tmp.txt File-2047.txt
UNIX> ./FATRW t1.jdisk export 5 tmp.txt
Reads: 4
Writes: 0
UNIX> diff tmp.txt File-3029.txt
UNIX>
```

Be careful -- Link numbers do not always equal sector numbers as in that example.

Let's try a larger disk to illustrate:

```
UNIX> ./jdisk_test CREATE t4.jdisk 5120000
UNIX> FAT t4.jdisk format
Size(sectors): 5000  FAT-Sectors: 10   Data-Sectors: 4990
UNIX> xxd -l 64 -g 2 -e t4.jdisk
00000000: 0001 0002 0760 0469 03c9 03c5 0007 0009  ....`.i.........
00000010: 0b1d 000c 0964 0f4b 04c1 0265 06fd 0c23  ....d.K...e...#.
00000020: 0be9 122b 0cc9 103d 0cf3 0954 0d8b 04cd  ..+...=...T.....
00000030: 00ff 0b3c 06f9 0d00 02b6 0b94 08c1 04d0  ..<.............
UNIX>
```

So, we've created a much larger disk, and there are 10 sectors of FAT. Therefore, $S = 10$ and $D = 4990$. You'll also notice that the program **FAT** doesn't set up its links according to any pattern. That is intentional. The first link on the free list has a value of 1, which corresponds to sector $S+1-1 = 10$. Now, suppose we import a file with 3029 bytes:

```
UNIX> ./FATRW t4.jdisk import File-3029.txt
New file starts at sector 10
Reads: 2
Writes: 5
UNIX> xxd -l 64 -g 2 -e t4.jdisk
00000000: 0006 0002 0760 0469 03c9 03c5 0007 0009  ....`.i.........
00000010: 0b1d 000c 0964 0f4b 04c1 0265 06fd 0c23  ....d.K...e...#.
00000020: 0be9 122b 0cc9 103d 0cf3 0954 0d8b 04cd  ..+...=...T.....
00000030: 00ff 0b3c 06f9 0d00 02b6 0b94 08c1 04d0  ..<.............
UNIX>
```

The start of the free list is now 6 -- block $S+6-1 = 15$. The newly created file starts at block ten, which corresponds to entry $10-S+1 = 1$ in the FAT. That entry has a value of 2, so sector 11 is the next one in the file. Sector 11 corresponds to FAT entry 2, which has a value of 0x760, which equals 1888 in decimal. To look at entry 1888 in the FAT, we need to look at the two bytes starting at byte 1888x2 = 3776:

```
UNIX> ./jdisk_test R t4.jdisk hex 3776 2
 60 07
```

```
UNIX>
```

That entry equals its index, so this is the last block in the file, and we can use the last two bytes in the sector to show how many bytes belong in the file. The sector is sector *S+1888-1* = 1987. Let's look at its last two bytes:

```
UNIX> echo '1898 1024 * p' | dc
1943552
UNIX> ./jdisk_test R t4.jdisk hex 1943550 2
 d5 03
UNIX>
```

That corresponds to the number 0x3d5 in hex which equals 981. Armed with this information, we can recreate the file:

```
UNIX> ./jdisk_test R t4.jdisk bytes 10240 1024 > tmp.txt
UNIX> echo 10240 1024 + p | dc
11264
UNIX> ./jdisk_test R t4.jdisk bytes 11264 1024 >> tmp.txt
UNIX> echo 1897 1024 * p' | dc
Unmatched '.
UNIX> echo '1897 1024 * p' | dc
1942528
UNIX> ./jdisk_test R t4.jdisk bytes 1942528 981  >> tmp.txt
UNIX> diff tmp.txt File-3029.txt
UNIX>
```

Or we can just use **FATRW**:

```
UNIX> ./FATRW t4.jdisk export 10 tmp.txt
Reads: 5
Writes: 0
UNIX> diff tmp.txt File-3029.txt
UNIX>
```

Details of your program

Your program needs to create files and read files within this structure. The output format needs to be just like mine, calling **jdisk_reads()** and **jdisk_writes()** at the end to report the disk usage. Your reads and writes must be minimal. In other words:

- Don't read sectors that you don't need to read.
- Don't read sectors twice.
- Don't write sectors twice.
- If you don't change a sector, don't write it.

In my implementation, I have a variable for a FAT which contains *S* pointers, all of which are initialized to NULL. When I need to read a link, I determine which of these pointers corresponds to the sector holding the link, and I check to see if the pointer is NULL. If so, I call **jdisk_read()** to read the sector, and set my pointer to equal the sector. That way, I only have to call **jdisk_read()** once per FAT sector.

Whenever I write a link, I first make sure that the proper sector has been read (using the same pointer as above). I then check to see if I am actually changing the link when I write it. If I'm not, I do nothing (and I won't write the sectors). Otherwise, I write the link, and set a flag denoting that the sector has been modified. At the end of the operation (this only applies to "import"), I write all of the sectors that have been modified using **jdisk_write()**. In that way, I only write a FAT sector once.

To make all of this easier, I wrote three procedures:

- **unsigned short Read_Link(int link)**.
- **void Write_Link(int link, unsigned short val)**.
- **void Flush_Links()** (checks the flags and calls **jdisk_write()**).

For reading and writing data sectors, I simply have one buffer of 1024 bytes. I use that one buffer for every read and write of data sectors. I do reading and writing one sector at a time. I use **fread()** and **fwrite()** to do file I/O.

You can use either C or C++ for this. If you use C++, you may *not* use the STL (so, no vectors, no maps, no lists, no sets). I did mine in C++, but is was a very C-like subset of C++.

Report and Digest

The **FAT** program also has a "report" function which tells you some information about the jdisk:

```
UNIX> ./FAT t1.jdisk report
Total Sectors: 10.  Data Sectors: 9.  Fat Sectors: 1.
Nodes in the free list: 1

BEGIN FILE INFORMATION
File Starting Block 1.  (Link 00001)
File Starting Block 5.  (Link 00005)
File Starting Block 7.  (Link 00007)
File Starting Block 8.  (Link 00008)
UNIX> ./FAT t4.jdisk report
```

```
Total Sectors: 5000.  Data Sectors: 4990.  Fat Sectors: 10.
Nodes in the free list: 4987

BEGIN FILE INFORMATION
File Starting Block 10.  (Link 00001)
UNIX> ./FAT t7.jdisk report
Total Sectors: 514.  Data Sectors: 512.  Fat Sectors: 2.
Nodes in the free list: 354

BEGIN FILE INFORMATION
File Starting Block 2.  (Link 00001)
File Starting Block 10.  (Link 00009)
File Starting Block 72.  (Link 00047)
File Starting Block 107.  (Link 0006a)
File Starting Block 130.  (Link 00081)
File Starting Block 153.  (Link 00098)
File Starting Block 174.  (Link 000ad)
File Starting Block 180.  (Link 000b3)
File Starting Block 206.  (Link 000cd)
File Starting Block 233.  (Link 000e8)
File Starting Block 234.  (Link 000e9)
File Starting Block 243.  (Link 000f2)
File Starting Block 250.  (Link 000f9)
File Starting Block 339.  (Link 00152)
File Starting Block 399.  (Link 0018e)
File Starting Block 400.  (Link 0018f)
File Starting Block 450.  (Link 001c1)
UNIX>
```

The shell script **Digest-Disk.sh** uses this functionality to grab each file and calculate its MD5 hash.

```
UNIX> sh Digest-Disk.sh
usage: sh Digest-Disk.sh FAT-File Directory-Holding-My-FAT-Program
UNIX> sh Digest-Disk.sh t1.jdisk .
```

When you run it, it creates two files -- **tmp-digest-output.txt** and **tmp-digest-short.txt**. The first of these has detail about all of the files. The second has the initial information, plus the sorted MD5 hashes. If two jdisks have the same size and hold the same files, then their output of **tmp-digest-short.txt** should be identical.

```
UNIX> cat tmp-digest-output.txt
Total Sectors: 10.  Data Sectors: 9.  Fat Sectors: 1.
Nodes in the free list: 1

BEGIN FILE INFORMATION
File Starting Block 1.  (Link 00001)
File Starting Block 5.  (Link 00005)
File Starting Block 7.  (Link 00007)
File Starting Block 8.  (Link 00008)
File from block 1 has a hash of 1bf31cac7815710ad161f90bdc63a285
File from block 5 has a hash of f571501ba07d3a25b9ac9cc4d1b566ef
File from block 7 has a hash of f177cb81362c455b297006db7dd73af0
File from block 8 has a hash of 0735029c6e27cdc776d92df994cf033d
END FILE INFORMATION

SORTED-HASHES
0735029c6e27cdc776d92df994cf033d
1bf31cac7815710ad161f90bdc63a285
f177cb81362c455b297006db7dd73af0
f571501ba07d3a25b9ac9cc4d1b566ef
UNIX> cat tmp-digest-short.txt
Total Sectors: 10.  Data Sectors: 9.  Fat Sectors: 1.
Nodes in the free list: 1


SORTED-HASHES
0735029c6e27cdc776d92df994cf033d
1bf31cac7815710ad161f90bdc63a285
f177cb81362c455b297006db7dd73af0
f571501ba07d3a25b9ac9cc4d1b566ef
UNIX>
```

To illustrate further, **t8.jdisk** is a jdisk that is the same size as **t1.jdisk**, and contains the same files, imported in a different order. Thus, it is different from **t1.jdisk**, but its **tmp-digest-short.txt** file is the same:

```
UNIX> sh Digest-Disk.sh t1.jdisk .
UNIX> mv tmp-digest-short.txt tmp-t1.txt
UNIX> sh Digest-Disk.sh t8.jdisk .
UNIX> mv tmp-digest-short.txt tmp-t8.txt
UNIX> diff tmp-t1.txt tmp-t8.txt
UNIX>
```

Two grading programs

Finally, I have two shell scripts that you can use to test your programs. The first, which is easier, is **Export-Grader.sh**. You run it from your own directory that holds your own **FATRW** program:

```
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Export-Grader.sh
usage: sh Export-Grader.sh Grading-Directory Disk-File Starting-Block
UNIX>
```

It will copy the jdisk file to the current directory, and then run my version of **FATRW** on it, exporting the file. Next it copies the jdisk file again, and runs the version of **FATRW** that's in the current directory. It double-checks that the exported files, and the standard outputs of the two programs are identical.

For example, we know that starting block 5 of **t1.jdisk** holds the file **File-3029.txt**. So, if I specify that disk and starting block, then **Export-Grader.sh** will go ahead and use that for testing:

```
UNIX> ls
FATRW
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Export-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t1.jdisk 5
Making sure the tmp files are all deleted.
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t1.jdisk to tmp-grader.jdisk
Running: /home/jplank/cs494/labs/Lab-2-FAT/FATRW tmp-grader.jdisk export 5 tmp-grader-file.txt
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t1.jdisk to tmp-your.jdisk
Running: ./FATRW tmp-yours.jdisk export 5 tmp-yours-file.txt
Double-checking that the exported files are the same.
Double-checking that the stdout files are the same.
All is good -- Success!
UNIX> ls
FATRW               tmp-grader-stderr.txt  tmp-yours.jdisk
tmp-grader-file.txt  tmp-grader-stdout.txt  tmp-yours-stderr.txt
tmp-grader.jdisk     tmp-yours-file.txt     tmp-yours-stdout.txt
UNIX> rm tmp*
UNIX>
```

The program **Import-Grader.sh** is more complex, but it double-checks importing files. Again, it copies the jdisk and imports the given file from the given directory. It does this with my version of **FATRW** and the one in the current directory. It calls **Digest-Disk.sh** on both jdisks, and compares the output of the **tmp-digest-short.txt** files. They must be identical. It then compares the new jdisk file to the old one, and determines how many sectors have changed. That number must equal the number of **jdisk_write()**'s that you report on line three of your standard output (plus or minus one -- I was in a kind mood):

```
UNIX> ls
FATRW
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh
usage: sh Import-Grader.sh Grading-Directory Disk-File Import-File
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t4.jdisk File-3029.txt
Making sure the tmp files are all deleted.
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t4.jdisk to tmp-grader.jdisk
Running: /home/jplank/cs494/labs/Lab-2-FAT/FATRW tmp-grader.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/File-3029.txt
Running: sh /home/jplank/cs494/labs/Lab-2-FAT/Digest-Disk.sh tmp-grader.jdisk /home/jplank/cs494/labs/Lab-2-FAT
The file information is in tmp-grader-short.txt
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t4.jdisk to tmp-your.jdisk
Running: ./FATRW tmp-yours.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/File-3029.txt
Running: sh /home/jplank/cs494/labs/Lab-2-FAT/Digest-Disk.sh tmp-yours.jdisk /home/jplank/cs494/labs/Lab-2-FAT
The file information is in tmp-yours-short.txt
Checking to see how many sectors you changed.
Your number of writes (+- 1) matched the number of changed sectors.
Your digest file and the correct digest files match.  Success!
UNIX> ls
FATRW          tmp-grader-short.txt   tmp.txt      tmp-yours-stderr.txt
tmp-digest-output.txt  tmp-grader-stderr.txt  tmp-yours.jdisk    tmp-yours-stdout.txt
tmp-grader.jdisk       tmp-grader-stdout.txt  tmp-yours-short.txt
UNIX> rm tmp*
UNIX>
```

.... And Gradescripts

The directory **/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts** has 100 gradescripts that test importing, and 63 that test exporting. Simply run them as shell scripts. There is a **gradeall** that tests them all. It takes a little while, mostly because of **t3.jdisk**.

```
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-001.sh
Success!
UNIX> cat /home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-001.sh
sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t9.jdisk t8.jdisk | tail -n 1 | awk '{ print $NF }'
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t9.jdisk t8.jdisk
Making sure the tmp files are all deleted.
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t9.jdisk to tmp-grader.jdisk
Running: /home/jplank/cs494/labs/Lab-2-FAT/FATRW tmp-grader.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/t8.jdisk
/home/jplank/cs494/labs/Lab-2-FAT/FATRW exited with an error.  See tmp-grader-stderr.txt
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t9.jdisk to tmp-your.jdisk
Running: ./FATRW tmp-yours.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/t8.jdisk
Both your program and the correct one exited with an error.  Success!
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-052.sh
Success!
UNIX> cat /home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-052.sh
sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t6.jdisk t5.jdisk | tail -n 1 | awk '{ print $NF }'
UNIX> sh /home/jplank/cs494/labs/Lab-2-FAT/Import-Grader.sh /home/jplank/cs494/labs/Lab-2-FAT t6.jdisk t5.jdisk
Making sure the tmp files are all deleted.
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t6.jdisk to tmp-grader.jdisk
Running: /home/jplank/cs494/labs/Lab-2-FAT/FATRW tmp-grader.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/t5.jdisk
Running: sh /home/jplank/cs494/labs/Lab-2-FAT/Digest-Disk.sh tmp-grader.jdisk /home/jplank/cs494/labs/Lab-2-FAT
The file information is in tmp-grader-short.txt
```

```
Copying: /home/jplank/cs494/labs/Lab-2-FAT/t6.jdisk to tmp-your.jdisk
Running: ./FATRW tmp-yours.jdisk import /home/jplank/cs494/labs/Lab-2-FAT/t5.jdisk
Running: sh /home/jplank/cs494/labs/Lab-2-FAT/Digest-Disk.sh tmp-yours.jdisk /home/jplank/cs494/labs/Lab-2-FAT
The file information is in tmp-yours-short.txt
Checking to see how many sectors you changed.
Your number of writes (+- 1) matched the number of changed sectors.
Your digest file and the correct digest files match.  Success!
UNIX> /home/jplank/cs494/labs/Lab-2-FAT/gradeall
/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-001.sh Success!
/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-002.sh Success!
/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-003.sh Success!
/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-004.sh Success!
/home/jplank/cs494/labs/Lab-2-FAT/Gradescripts/Gradescript-005.sh Success!
.....
```

Another walk-through

Please see morehelp.html for a detailed walkthrough of how your program can read the file starting at sector 10 of t4.jdisk.