CS494/CS594 -- Lab 1: Implementing a File Allocation Table

- CS494/CS594
- Fall, 2020
- James S. Plank
- This file: http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/labs/Lab-1-FAT/
- Lab Directory: **/home/jplank/cs494/labs/Lab-1-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 Winton, 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 6d6573414243444546474 8
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

ti.

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 sector