

## Projects 5 and 6 – A File System Design and Implementation

Assigned: March 29, 2004

Due: April 18, 23:59pm)

**Overview.** For this assignment, you are to implement a rudimentary file system. We will be using the layered approach in this file system. You are given the functionality of the physical disk drive (the layer below you) in `physical.h` and `physical.c`. The description of the functionality you must provide is found in `filesys.h`, and a skeleton implementation in `filesys.c`. You are also given a testing program `testfs.c` which conducts a number of tests on your file system, which we will also use to grade your final submission. Also provided is a Makefile which you can use to build your file system implementation.

There are also Java implementations of each of the above files, should you choose to complete the assignment in Java. You are encouraged to use C, since C is more suited for this type of work, but you can choose either.

You can compile and run your program on any Unix system that you have available. Possibilities are 0311ece001-012, Acmeey, Yamsrv, Bigzilla, or most CoC Unix systems.

**Getting Started.** Start by choosing an appropriate cluster size. We don't want to track the use of each individual sector on the disk, since this would require substantial overhead. Too large of a cluster size will waste disk space.

Next, define a C Structure (or Java class) to use for a "Directory" entry, to be stored on disk. Since we are assuming a fixed maximum file name length of 8 characters (see `MAX_FILE_NAME` in `filesys.h`), this can be compact, probably not more than 16 bytes (although the Java implementation takes a bit more since characters are 16 bits in Java). Also note we are assuming a fixed maximum number of files that can be stored on the file system (see `MAX_FILES` in `filesys.h`). This allows a fixed allocation of sectors to store the directory entries on disk.

Next choose a way to mark clusters as reserved (or alternately mark clusters as available), and choose a way to link clusters together. A good and simple way to do this is to use the *File Allocation Table (FAT)* method, where a single table is used to track both sector usage and sector linkage. The *FAT* table is described in the Silberschatz/Galvin textbook in section 12.14.2.

With those two decisions made, you can then determine apriori how much disk space to reserve for the Directory table and the Allocation table. Then, code the "FormatLow" and "FormatHigh" routines, which should simply zero out all disk sectors, and initialize the Directory Table and Cluster Allocation Table appropriately. You can test the format routine by running "testfs 1 2", which says to run test number 1 (FormatLow) followed by test number 2 (FormatHigh).

At this point, you can code the "Mount" procedure. This will be called every time we run "testfs", and allows you to do any initialization you might need (such as reading the Directory and Allocation tables into memory). Then code the "UnMount" procedure, which will be called in "testfs" when all tests are completed. This allows you to write any updates to the Directory and Allocation table back to the disk.

Next, you need to decide what information you need in an "Open File Table" entry, to keep track of open files. At a minimum, you need a pointer to the Directory entry, a pointer to a memory buffer for data (probably one cluster in size), a current offset into the buffer, and the current file position. Again we are assuming a fixed maximum number of open files (`MAX_FILES_OPEN = 10`), so you can just use a static array for this.

Next, code the `OpenWrite` routine, which needs to create a new Directory entry for the file specified and creates a corresponding Open File Table entry. Then implement the `Delete` routine, which deletes an existing file. For `Delete`, be sure to mark any clusters that were assigned to that file as "available". Note that `OpenWrite` should automatically delete any previously existing file by the specified name.

Then, code the "Write" routine. Write should write the data first to a memory buffer (writing that buffer to disk only when it fills up or on a subsequent "flush" call).

Next should be `flush`, which causes the memory buffer to be written out to disk. Flush should NOT adjust the current file position.

The remaining routines can be coded in any order. The requirements for all routines are clearly documented in "filesys.h" and "filesys.c". You should probably save "Seek" and "OpenExtend" until the last, as they are the most complicated.

**Testing your Program.** You can test your program using the "testfs.c" program I provided. It accepts one or more numeric command line arguments which specify which tests to run, and in which order. Presently there are 16 tests available in testfs, numbered from 1 to 16. As you are debugging, feel free to add debug statements into testfs.c. But for final testing we will use the original testfs.c and will likely test your final submission as follows:

```
testfs 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

```
testfs 7
```

```
testfs 8
```

The second two will insure "persistence" (you correctly updated the Directory and Allocation table on the disk).

**A Debugging Hint.** You probably realize that filesys.c does not actually write directly to the hard disk at the sector number requested. Instead, we simulate this behavior by just using a normal Unix file, called `physdisk.dat`. It is extremely useful during debugging to inspect the contents of this file, using the Unix "octal dump" utility `od`, as follows:

```
od -x physdisk.dat
```

This shows the contents of the file in hexadecimal.

```
P5/mysolution>od -x physdisk.dat
0000000 ffff ffff ffff ffff 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0007740 0000 0000 cab0 0804 0000 0000 0000 0000
0007760 0000 0000 0000 0000 0000 0000 0003 0000
0010000 6966 656c 0031 0000 0000 0000 0003 0000
0010020 0000 0000 0000 0000 0000 0000 ffff 0000
*
0014000 0000 0000 0000 0000 0000 0000 0000 0000
*
17640000
```

enzo

The above example shows that the FAT table (at offset 0) has five reserved entries. The directory table starts at location 10000 (octal) and has a name of `file1`. On Intel systems that use little endian format, the 16 bit hex values are shown in byte reversed order, so what is actually shown for the file name is `if e1 1` (the word `file1` with each pair of two bytes swapped).

Using this dump utility can help determine what the program is doing for the various tests.

**Grading Criteria:** Tests 1 – 2 : 4 points each.  
Tests 3 – 6 : 2 points each.  
Tests 7 – 12 : 10 points each.  
Tests 13 - 16 : 6 points each.  
Total points 100