Hints for Programming Assignment 2

Bijan Soleymani

14th March 2007

1 Objective

The goal of this assignment is to implement a simple filesystem. This filesystem supports a limited number of files, limited lengths filenames and a single directory. Free space is managed through a File Allocation Table (FAT).

This filesystem should be implemented on top of the disk emulation code from programming assignment 1. This means that all read and write operations must use: read_blocks and write_blocks. Note that this means you can't directly use functions like fread, fwrite or fseek.

2 Data Structures

There are four main data structures: the (root) directory table, the File Allocation Table (FAT), the free block list and the file descriptor table.

The first three are stored on the disk. The last is only stored in memory. A copy of the structures on disk, is also kept in memory. They should be periodically written back to the disk.

2.1 Directory Table

The directory table contains (at least) the following information for each file:

- Filename: the name of the file. This can be a fixed length string (e.g. char name[16])
- Size: the size of the file in bytes.
- Date: the last modification time of the file. Can be stored as an int. Do something like: date = time(NULL). (don't forget to #include<time.h>)

- FAT_index: index of the FAT entry that contains points to the first block of the file.

You can make a structure of the form:

```
struct directory_entry {
  char name[16];
  int size;
  int date;
  int FAT_index;
};
```

And you can make the table be an array of the structures:

```
struct directory_entry directory[MAX_FILES];
```

Where MAX_FILES is a reasonable value for the maximum number of files.

2.2 FAT

Each FAT entry contains the index of a block and that of the next FAT entry for the current file. It can be represented as:

```
int FAT[NUM_BLOCKS][2];
```

Where FAT[n][0] contains the block corresponding to FAT entry n, and FAT[n][1] contains the index of the next entry (or EOF if there are no more blocks in the file). So to go through all the blocks in a file:

```
temp = directory[i].FAT_index;
while(temp!=EOF){
  block = FAT[temp][0];
  temp = FAT[temp][1];
  // Insert code to read or write block
}
```

2.3 Free Block List

The free block list is a simple array that indicates whether each block is used or not. So something like:

char free_list[NUM_BLOCKS];

And something like this to find a free block:

```
i=0;
while(free_list[i]!=FREE){
  i++;
}
```

Don't forget to mark the blocks that contain the Directory table, FAT and free block list as used. If you forget to do this then sfs_write will think they are free blocks and overwrite them.

2.4 File Descriptor Table

The file descriptor table contains the following information for each open file:

- read_ptr: the current read position in the file (in bytes). Should be set to 0 when a file is opened.
- write_ptr: the current write position in the file (in bytes). Should be set to the size of the file, when a file is opened (0 if the file is newly created).
- directory_index: index of the directory entry for this file

The data structure will look something like this:

```
struct fdt_entry {
  int read_ptr;
  int write_ptrl
  int directory_index
};
struct fdt_entry FDT[MAX_FILES]
```

3 Function Outlines

3.1 void mksfs(int fresh)

Similar to init disk (in fact it can call init_disk), but also initializes all the data structures. If fresh is 0, then read all the data structures from the file. Else create empty data structures and overwrite the ones in the file.

3.2 void sfs_ls()

Print out the contents of the directory table in user friendly form (output should look a bit like "ls -l" in Linux).

3.3 int sfs_open(char *name)

Open a file for reading and writing, and creates the file if it doesn't exist.

If the file exists already, all this does is add an entry in the file descriptor table.

If the file doesn't exist then it finds a free block, a free FAT entry and a free directory entry. It then sets the file size to 0, the modification date to the current date, and sets FAT_index to the index of the free FAT entry. Then it sets the FAT entry to point to the free block, and sets the next to EOF (since the file only has one block at this time). Finally the free_block list is updated to indicate that the block chosen is no longer free.

The function returns the index of the newly created entry in the file descriptor table.

3.4 void sfs_close(int fileID)

This function just clears the entry in the file descriptor table.

3.5 void sfs_write(int fileID, char *buf, int length)

Write length bytes to the disk. Note that you have to do this using the block-oriented functions read_blocks and write_blocks. This means that if the writing occurs in the middle of a block, you'll have to read the block, add the data after the preexisting data, and rewrite the block to the disk.

3.6 void sfs_read(int fileID, char *buf, int length)

Read length bytes from the disk. Note that you have to do this using the block-oriented function read_blocks. This means that if you are reading from the middle of a block, you'll have to read the block, and copy the desired data to buf.

3.7 int sfs_remove(char *file)

Deletes the file. This consists of freeing the blocks used by the file. Clearing the FAT entries used by the file. And finally clearing the directory entry for the file.