Machine Problem 7: Vanilla File System

Introduction

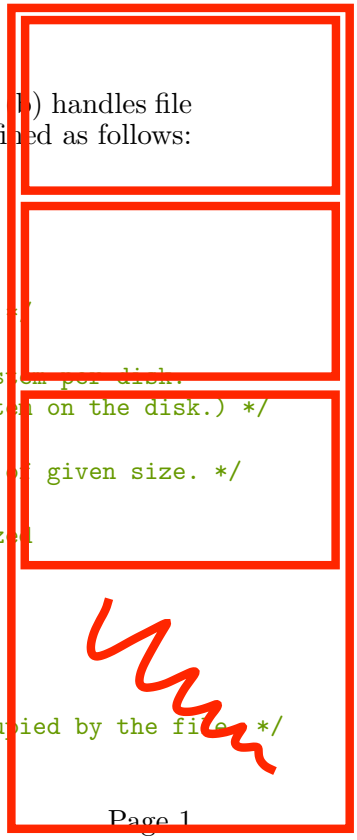
In this machine problem you will implement a **simple file system**. Files support **sequential access only**, and the file name space is very simple (files are identified by unsigned integers; no multilevel directories are supported). The file system is to be implemented in classes `File` and `FileSystem`, respectively. Class `File` implements sequential read/write operations on an individual file:

```
class File {
     /* -- your file data structures here ... */
public:
    File(/* you may need arguments here */);
    /* Constructor for the file handle. Set the 'current
       position' to be at the beginning of the file. */
    int Read(unsigned int _n, char * _buf);
    /* Read _n characters from the file starting at the current location and
       copy them in _buf.  Return the number of characters read.
       Do not read beyond the end of the file. */
    void Write(unsigned int _n, char * _buf);
    /* Write _n characters to the file starting at the current location, if we
       run past the end of file, we increase the size of the file as needed. */
    void Reset();
    /* Set the 'current position' at the beginning of the file. */
    void Rewrite();
    /* Erase the content of the file. Return any freed blocks.
       Note: This function does not delete the file! It just erases its content. */
    bool EoF();
    /* Is the current location for the file at the end of the file? */
};
```

Notice that we don't do much error checking. This is not industrial-strength code!

Class `FileSystem` (a) controls the mapping from the file name space to files and (b) handles file allocation, free-block management on the disk, and other issues. Its interface is defined as follows:

```
class FileSystem {
   /* -- your file system data structures here ... */
public:
   FileSystem();
   /* Just initializes local data structures. Does not connect to disk yet. */
   bool Mount(SimpleDisk * _disk);
   /* Associates this file system with a disk. Limit to at most one file system per disk.
      Returns true if operation successful (i.e. there is indeed a file system on the disk.) */
   static bool Format(SimpleDisk * _disk, unsigned int _size);
   /* Wipes any file system from the disk and installs an empty file system of given size. */
   File * LookupFile(int _file_id);
   /* Find file with given id in file system. If found, return the initialized
      file object. Otherwise, return null. */
   bool CreateFile(int _file_id);
   /* Create file with given id in the file system. If file exists already,
      abort and return false. Otherwise, return true. */
   bool DeleteFile(int _file_id);
   /* Delete file with given id in the file system; free any disk block occupied by the file. */
```



```
};
```

This file system is just a proof-of-concept, and the interface is not indicative of industrial-strength code. A few points if you are perplexed about the class definition:

- You may be tempted to ask why we don't pass the disk to the `FileSystem` constructor. The reason for separating the construction from the mounting of the file system is that it is difficult to cleanly handle errors that occur in constructors. Therefore, you should keep the operations in the constructor simple, and leave the complicated stuff in the `Mount` function, where you return an error code (not here, but in a more serious implementation).
- Why do we have to specify the file system size in the `Format` function? This is to make your life simple: by specifying the size parameter you don't need to find out how the big the disk is. Feel free to ignore the size parameter and use the entire disk; you will have to query the size of the disk on your own.

Opportunities for Bonus Points

OPTION 1: DESIGN of a thread-safe file system. (This option carries 6 bonus points.) For implementation purposes, you can assume that a file system is accessed by at most one thread at a time. If multiple threads can access the file system concurrently, there are plenty of opportunities for race conditions. You are to **describe** (in the design document) how you would handle concurrent operations to the file system in a safe fashion. This may require a change to the interface; probably not. Separate your design in two portions: (a) File System access, and (b) File access.
OPTION 2: IMPLEMENTATION of a thread-safe file system. (This option carries 6 bonus points.) For this option you are to **implement** the approach proposed in Option 1. **Note: Work on this option only after you have addressed Option 1.**

A Note about the Main File `kernel.C`

The main file for this MP is very similar in nature to the one for MP6. We have modified the code for some of the threads to access the file system.

- At this point the code in `kernel.C` instantiates a copy of a `SimpleDisk`. Feel free to replace the disk by your implementation of `BlockingDisk` if you are confident enough that it works.
- The thread function `fun3` exercises the file system as follows:
  1. First, it formats disk "C" to contain a 1 MB file system.
  2. It then mounts the file system.
  3. It then goes into an infinite loop, where it calls the function `exercise_file_system()` repeatedly.
  4. The function `exercise_file_system()` creates two empty files, with identifier 1 and 2. It then opens the two files and writes a string into each. It then closes the two files. It then opens them again and reads the content of the two files. It then checks whether the content is correct. It then deletes the two files again. (Note that this is a very simple test and is intended to only very superficially test the correctness and completeness of your file system implementation.)
  5. If you are experiencing difficulties with getting the function `exercise_file_system()` to work correctly, check it in detail. You may need to slightly modify it to make it work, depending on your implementation of the file system.

- File `kernel.C` is "scheduler-free". If you want to bring in your implementation of Blocking Disk you will have to add a scheduler to the kernel.

A Note about the Configuration

Similarly to MP6, the underlying machine will have access to two hard drives, in addition to the floppy drive that you have been using earlier. The configuration of these hard drives is defined in File `bochsrc.bxrc`. You will notice the following lines:

```
# hard disks
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="c.img", cylinders=306, heads=4, spt=17
ata0-slave: type=disk, path="d.img", cylinders=306, heads=4, spt=17
```

This portion of the configuration file defines an the ATA controller to respond to interrupt 14, and connects two hard disks, one as master and the other as slave to the controller. The disk images are given in files "c.img" and "d.img", respectively, similarly to the floppy drive image in all the previous machine problems. **Note:** There is no need for you to modify the `bochsrc.bxrc` file.

Note: If you use a different emulator or a virtual machine monitor (such as VirtualBox) you will be using a different mechanism to mount the hard drive image as a hard drive on the virtual machine. If so, follow the documentation for your emulator or virtual machine monitor.

The Assignment

1. Implement the file system as described the two classes `File` and `FileSystem` above.
2. For this, use the provided code in `file.H/C` and `file_system.H/C`, which contain definition and a dummy implementation of class `File` and class `FileSystem`, respectively.
3. Check that the test function `exercise_file_system()` generates the correct results. (Basically, the goal is to not make it throw assertion errors.)
4. If you have time and interest, pick one or more options and improve your file-system design and implementation.

What to Hand In

You are to hand in a ZIP file, called `mp7.zip`, containing the following files:

1. A design document called `design.pdf` (in PDF format) that describes your design and the implementation of your File and File System. **If you have selected any options, likewise describe design and implementation for each option. Clearly identify in your design document and in your submitted code what options you have selected, if any.**
2. A pair of files, called `file.H` and `file.C`, which contain the definition and implementation of class `File`.
3. A pair of files, called `file_system.H` and `file_system.C`, which contain the definition and implementation of class `FileSystem`.

4. Any new or modified file. Clearly identify and comment the portions of code that you have modified.

Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.

Failure to follow the handing instructions will result in lost points.
In particular, expect a significant loss of points if you submit incomplete code or code that does not compile for any reason!