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 `FileSystem` and `File`, respectively.

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 (for more details check out the provided file `file_system.H`):

```
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. */
   Inode * 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 to 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.

Class `File` implements sequential read/write operations on an individual file (for details check out `file.C`):

```
class File {
     /* -- your file data structures here ... */
public:
    File(FileSystem * _fs, int _id);
    /* Constructor for the file handle. Set the current position to be at the
       beginning of the file. */
    ~File();
    /* Closes the file. Deletes any data structures associated with the file
       handle. */
    int Read(unsigned int _n, char * _buf);
    /* Read _n characters from the file starting at the current position and
       copy them in _buf.  Return the number of characters read. Do not read
       beyond the end of the file. */
    int Write(unsigned int _n, const char * _buf);
    /* Write _n characters to the file starting at the current position. If the
        write extends over the end of the file, extend the length of the file
       until all data is written or until the maximum file size is reached. Do
       not write beyond the maximum length of the file. Return the number of
       characters written. */
    void Reset();
    /* Set the current position to the beginning of the file. */
    bool EoF();
    /* Is the current position for the file at the end of the file? */
};
```

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

Why Separate File and File System?

We want to separate the different aspects of file management into different classes. Class `FileSystem` manages the reading/writing from/to disk. It also manages the storing of the inode list in the INODES block and the free-block list (in the FREELIST block, depending on your implementation). Class `File`, on the other hand, handles the sequential read and write of sequences of bytes from and to the file. In order to do this, it maintains a current position in the file.

The information about files and how they are stored in the file system is stored in objects of class `Inode` (check `file_system.H` for details):

```
class Inode {
  friend class FileSystem; // The inode is in an uncomfortable position between
  friend class File;       // File System and File. We give both full access
                           // to the Inode.

private:
  long id; // File "name"

  /* You will need additional information in the inode, such as allocation
     information. */

  FileSystem *fs; // It may be handy to have a pointer to the File system.
                  // For example when you need a new block or when you want
                  // to load or save the inode list. (Depends on your
                  // implementation.)

  /* You may need a few additional functions to help read and store the
     inodes from and to disk. */
};
```

An Entire File System? That's soo much Work!

This MP is not asking you to write a full-fledged file system, but rather a simple proof of concept. To make life easier, we add a few constraints to the functionality of the file system:

1. Implement a single-level directory. The file system can manage a collection of files, but no directories. This means that you don't need to implement directory files or even a separate directory to look up files. Instead, you can store the file names in the inodes and use the inode list as directory to find files.

2. Limit the length of files to at most one block! This greatly simplifies the implementation of the file allocation. Simply store the single block number of the file in the inode.

3. Keep the free-block list as simple as possible. Feel free to store the state of each block (used or free) as a character in a free-block array, which you store in the FREELIST block on the disk.

Implementation Hints

Unless you want to, don't get carried away with this machine problem. In particular, don't try to imitate existing file systems. This would be too much work. We list a few hints to make your life easier.

First, it will help to designate one block to store the inode list; call it the INODES block. This can be Block 0 on the disk. (We won't need a super block.) Use the next block to store the free-block list; call it the FREELIST block. Implement the free-block list as a "bitmap" (of chars, not of bits, if you want to keep it simple).

Whenever you create a file, find a free block and assign it to the file. Store its block number in the inode. This block number will be the entire allocation information for this file, as files are at most one block long.

Implement four functions to read and write the inode list and the free-block list from and to disk.

The sequential operations on files are implemented in class `File`. Whenever you open the file, you copy the content of the block to a 512 Byte cache. (You write the cache back to disk when you close the file.) You keep a 'current position', which indicates where the next character will be read/written from/to the file. As you, say, read a string from the file, you start copying characters one-by-one from the cache to the buffer. As you do that, you make sure that you don't overshoot the end-of-file, i.e. your current position does not go beyond the length of the file (store in the inode). When you write, you do the opposite: you copy character-by-character from the buffer to the cache, starting from the current position. If you exceed the length of the file, you simply extend the end of the file by incrementing the file length in the inode as you keep writing. In both cases make sure that you don't overshoot the maximum file size, which is one block.

Opportunities for Bonus Points

The maximum file size of maximum one block (512 Byte) is a major limitation of the file system described above. If you are interested, you can propose how you would improve the file system to allow for larger file sized.

OPTION 1: DESIGN of an extension to the basic file system to allow for files that are up to 64kB long. (This option carries 6 bonus points.) How would you extend the implementation above to allow for files that are up to 64kB long? Describe in detail how you would implement the allocation of files. What functions would have to be added, which functions would have to be modified, how? How would the data structures change? (Be precise. Points will not be given for general hand waving.) Separate the changes to the file allocation to changes in the read and write operations in class `File`.

OPTION 2: IMPLEMENTATION of the extensions proposed in Option 1. (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 your basic implementation works and after you have addressed Option 1.**

A Note about the Main File `kernel.C`

The main file for this MP is greatly simplified compared to previous MP's. In particular, it does not contain threads.

- 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. (not recommended)
- The main thread exercises the file system as follows:
  1. First, it formats disk "C" to contain a 128 KB file system.
  2. Independently from the formatting, it creates a new file system.
  3. It then mounts the file system.
  4. It then goes into an infinite loop, where it calls the function `exercise_file_system()` repeatedly.
  5. 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.)
- 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.

A Note about Stack Memory

As you write your code, keep in mind that the kernel "thread" (i.e. the thread that start all the other threads and then dies) has 8kB of stack. We don't have other threads in this MP. If you plan to store lots of data locally, make sure that you don't overrun the stack. Use the heap instead.

The Assignment

1. Implement the file system as described in the three classes `FileSystem`, `File`, and `Inode` 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 classes `File`, `Inode`, and `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, with name `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 the 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. All the source files and the `makefile` needed to compile the code.
3. Clearly identify and comment changes to existing code.
4. 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.

Note: Pay attention to the capitalization in file names. For example, if we request a file called `file.H`, we want the file name to end with a capital `H`, not a lower-case one. While Windows does not care about capitalization in file names, other operating systems do. This then causes all kinds of problems when the TA grades the submission.

Failure to follow the handin instructions will result in lost points.