COSI 131a: Fall 2014

Programming Assignment 3

Due date: Part 1: November 25, 2014 11:55 PM ET

Part 2: December 3, 2014 11:55 PM ET

Overview

In this project, you will supply modules that complete the design of a simple file system on top of a simulated disk. Your file system provides a standard seek/read/write interface and uses inodes and single-, double-, and triple-indirection to support large files (similar to the class UNIX file system model we learned in class).

Your task is to add code for bitwise operations to complete the free space bitmap, and to add indirection to support files larger than 10 blocks. There is a large amount of code provided to you in this project, so it will be important to pay attention to interfaces and boundaries between modules to avoid getting overwhelmed by the surrounding code as you add your solution.

Quick Reference

Task 1:

You will modify at least the following two files:

1. **Bitwise.java**. Finish the implementation of the following methods:
    - public static boolean isset(int i, byte b)
    - public static boolean isset(int i, byte bytes[])
    - public static byte set(int i, byte b)
    - public static void set(int i, byte bytes[])
    - public static byte clear(int i, byte b)
    - public static boolean clear(int i, byte bytes[])
    - public static String toString(byte b)
2. **MyFileSystem.java**. Modify **getDirectBlock** to support large files using single-indirection.

Task 2:

MyFileSystem.java. Modify **getDirectBlock** to support even larger files using double-indirection and triple-indirection.

For both tasks, you may not modify the following files without first getting permission from TAs:

- FileSystem.java
- Shell.java

You may modify and create other classes / files at your discretion.

Expected Behavior

- Support large files with single-, double-, and triple-indirection.
- Allow holes in files.
- Work even if constants are changed (such as "buying a bigger disk," i.e., changing Disk.NUM_BLOCKS).

This list is not a substitute for reading the entire assignment and asking questions.

System Documentation

The file system with which you will be working has a fairly large codebase, including modules for creating a simulated disk, representing different types of disk blocks, and managing free space, as well as the file system itself. You will want to utilize the following references in addition to familiarizing yourself with the source code:

- PA3 - Design Overview document.
- API reference in the src/doc folder.

Your Tasks

Getting the Code

- Download the source code archive, unpack it wherever you like and import it in Eclipse. (in Eclipse: File -> Import -> Existing Projects into Workspace -> Browse -> locate the unzipped project -> Open -> Next -> Finish)
- Review all the files, making note of the following (this is not a complete manifest, but describes some of the files that are most important for you to understand first):
    - **FileSystem.java** contains the File System interface. The Shell and unit tests assume that MyFileSystem implements FileSystem. You may **not** modify FileSystem.java or make MyFileSystem implement a different interface.
    - **MyFileSystem.java** contains an implementation of the FileSystem interface. You will implement indirection to support large files in this file. This file includes a static nested top-level class **DirectBlock** which is of importance when implementing **getDirectBlock**.
    - **Bitwise.java** contains a utility class for doing bitwise operations. You will make some small additions in this file to fix the free map.
    - **FreeMap.java** contains an implementation of a free space bitmap that is stored both in the SuperBlock and in other free space bitmap *blocks*. You will not need to modify the FreeMap but you need to understand how it works, especially the methods find(), clear(int blockNum), and save().
    - **Shell.java** a basic shell that can interact with any FileSystem.

- Review what changes you have to make and how to test them.
    - Fix the Bitwise operations and test them by running the test: TestBitwise.java.
    - Run the Shell and test the File System. Notice that it crashes if you try to write to a large file.
    - Add indirection in MyFileSystem.java and test it by running the test: TestMyFileSystem.java.

Fix Bitwise

The FreeMap class is already implemented for you. FreeMap relies on methods from the Bitwise utility class. The following methods in Bitwise.java are unfinished and require you to implement them:

- public static boolean isset(int i, byte b)
- public static boolean isset(int i, byte bytes[])
- public static byte set(int i, byte b)
- public static void set(int i, byte bytes[])
- public static byte clear(int i, byte b)
- public static boolean clear(int i, byte bytes[])
- public static String toString(byte b)

See the javadoc comments in Bitwise.java for more information about these methods. Remember that "setting" a bit means set it to **1**, while "clearing" a bit means clear it to **0**.

You cannot use the standard assignment and indexing operators to which you are accustomed in Java when you want to access bits; the smallest unit you can directly address is a byte. So, you must use bitwise operators on bytes to test the values of bits. For your assignment, the bitmap will be stored as a byte array. You may want to brush up on bitwise operators (http://en.wikipedia.org/wiki/Bitwise_operation) and their use in java (http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html). An important point to remember is that in Java, all types are signed, even bytes. You may have to take the sign bit into account depending on what operations you use for manipulating the bytes in the bitmap.

You may not use java.util.BitSet or any other bit-manipulation classes available on Internet. You also may not "simulate" a bitmap using an array of booleans. You must write your own code for reading and writing the bitmap, and it must pack the free status for 8 blocks into each byte. Think about how you can use the bitmasks[] array in Bitwise.java class to implement these methods.

Indirect Blocks

PA3 does not support indirection "out-of-the-box". Finding the block that contains a byte offset in the file is implemented in the following utility function in MyFileSystem:

```
/**
 * Get a DirectBlock object representing the direct block given
 * the current seek position in the open file identified by fd. A
 * DirectBlock references the direct block and offset within that
```

```
     * block containing the current seek position.
     *
     * If the current seek position is within a hole or beyond the end
     * of a file, then if create is true then a block will be
     * allocated to fill the hole. If there is no more free space in
     * the file system, null will be returned. If the seek position is
     * in a hole and create is false, then a block containing zeroes
     * will be returned.
     *
     * @param   fd    valid file descriptor of an open file
     * @param         mode  MODE.w if holes should be filled, MODE.r
     *                otherwise (holes will be read as blocks
     *                           of all zeros)
     * @returns DirectBlock       block and offset in that block where the
     *                                 seek position of fd can be found
     */
    private DirectBlock getDirectBlock(int fd, MODE mode) {
    }
```

In the code provided to you, this function works correctly for the first 10 blocks of the file. Accessing bytes in a logical block number higher than 10 will cause the program to crash. It is up to you to use single-, double-, and triple-indirect blocks to solve this problem. You can read about how indirect pointers work at the System Design document provided to you.
The purpose of the mode parameter is to allow getDirectBlock to work when writing to holes or beyond the end of a file. Remember that when writing to blocks that are only accessible through indirect blocks, those indirect blocks may need to be allocated along with the direct block where the actual file data is stored.

Implementation Notes

You may add a new abstraction layer to represent the search for direct blocks, or perhaps a set of direct blocks accessed in a range. Whether or not this simplifies the task in your mind is up to you. However, be careful not to make the project more complex than it needs to be. Your goal is to avoid having to rewrite existing code, and instead add a relatively small amount of new code that adds an important new feature.

Testing

Unit Tests

Included with the PA3 code are unit tests for Bitwise and MyFileSystem:

- TestBitwise.java
- TestMyFileSystem.java

The first three classes in the TestMyFileSystem.java test the interface of the file system which is provided to you. You should pass these tests even if you have not implemented indirection. The last class “TestMyFileSystem.Indirection” tests your implementation of indirection that is included in your tasks.

Feel free to add new unit tests if you wish. However, be aware that your TAs will be using the aforementioned two classes to test your code, so make sure that you pass these tests before writing your own.

Before submitting the first part of the assignment (bitwise operations and single-indirection) make sure that you pass all of the tests in the TestBitwise.java class as well as all the tests associated with single-indirection from my TestMyFileSystem.java class.

For the second part of the assignment you should submit your complete code (including the code for the first part) and ideally you should be passing all of the tests included in these two java classes.

The Command Interpreter

The main method in class Shell implements a simple command interpreter that is useful for testing your file system. You can either use it interactively by running the Shell class or you can have it run a test script that contains multiple commands that should be given as an argument parameter when you run the Shell. You will find example scripts in the *test_data* folder. Input lines starting with "/*" or "//" are ignored (the latter are echoed to the output). Other lines have the format:
[var =] command [args]
The optional prefix var = causes the result of the command to be assigned to a variable. In any case, the result of the command is printed. There is one command for each of the ten methods of class FileSystem as well as three additional commands: help, vars, and quit. The help command prints a list of commands, the vars command lists the current values of all interpreter variables that have been assigned values, and the quit command terminates the program. With the exception of the second argument to write, each argument can be either an integer or the name of a variable. The command:
write fd pattern size
writes size bytes to the indicated file at the current offset. The data is generated by repeating pattern over and over the required number of times.

How to Submit

Create a zip or tar.gz file containing your completed code. You should name this file: last name underscore first name. For example if your name is Amanda Bynes you should name your file: Bynes_Amanda.zip or Bynes_Amanda.tar.gz. This file should contain the src folder within the archive -not as the top level (when the archive is opened the src folder should be visible). Upload this archive to the submission folder on Latte. Remember your code should

be compilable using Java 1.7. Please also include a README file that contains a description of your implementation and any misbehavings your code might have. This is an individual project and all submitted work must be your own, however you are allowed to discuss your solution with other students, **do not share your code**. If you have extensive discussions with another student please indicate this in your README file.