File-System Implementation

Programming Projects

Designing a File System

This project consists of designing and implementing a simple file system on top of a simulated disk. The structure of this file system closely follows UNIX-based file systems, where a superblock is used to describe the state of the file system. Internally, each file is stored in the file system using an inode data structure. The structure of the simulated disk appears in Figure 12.1.

Simulated Disk

The simulated disk uses a file named `DISK` to simulate a disk with `NUM_BLOCKS` blocks of `BLOCK_SIZE` bytes per block. The simulated disk is represented with the Java class `Disk.java` (Figure 12.2).

First, we explore the following three methods in this class:

1. `void read(int blockNum, byte[] buffer)`—Read from the specified block number on disk into a buffer.
2. `void write(int blockNum, byte[] buffer)`—Write the contents of the buffer to the specified block number on disk.

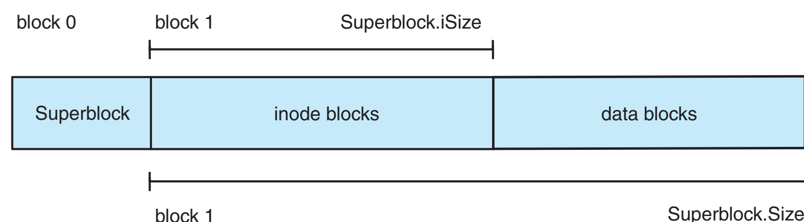

Figure 12.1 Structure of the simulated disk.

```
public class Disk {
   public void read(int blocknum, byte[] buffer) {}
   public void read(int blocknum, SuperBlock block) {}
   public void read(int blocknum, InodeBlock block) {}
   public void read(int blocknum, IndirectBlock block) {}
   public void write(int blocknum, byte[] buffer) {}
   public void write(int blocknum, SuperBlock block) {}
   public void write(int blocknum, InodeBlock block) {}
   public void write(int blocknum, IndirectBlock block) {}
   public void stop() {}
}
```

Figure 12.2 The `Disk` class.

3. `int stop(boolean removeFile)`—Stop the disk and report how many read and write operations have taken place. If `removeFile` is `true`, this method will also delete the `DISK` file.

In each of these three methods, `blockNum` must be in range `0..Disk.NUM_BLOCKS-1` and `buffer` must be a byte array of exactly `Disk.BLOCK_SIZE` bytes. Note that blocks must be read and written in whole-block units, not portions of a block.

The `Disk` constructor looks for a file named `DISK` in the current directory. If no such file exists, the program will create one and initialize the first block to all 0s. All other blocks must be written at least once before the block can be read.

Overloaded versions of `read()` and `write()` are described later in this project.

Superblock

Block 0 of the disk is the **superblock**, which contains information about the rest of the disk. You will want to keep a copy of this block in memory at all times. It should be read in from the `DISK` file when the file system starts up and written back out to the `DISK` file before shutting down. The superblock holds the following variables:

1. `int size`—Number of blocks in the file system
2. `int iSize`—Number of index blocks
3. `int freeList`—First block on the free list

The size of the file system is recorded in the superblock to allow the file system to use less than the whole disk and to support various disk sizes. In all the data structures on disk, a **pointer** to a disk block is an integer in the range `1..Disk.NUM_BLOCKS-1`. Since block 0 (the superblock) is treated uniquely, you can use a block number of 0 to represent a null pointer. The superblock is represented with the file `SuperBlock.java`, which is available online (www.os-book.com).

Free-Space List

Free-space management will use a grouping technique similar to that described in Section 12.5.3. Specifically, each block of the free list contains `Disk.POINTERS_PER_BLOCK` block numbers, where `Disk.POINTERS_PER_BLOCK` = `Disk.BLOCK_SIZE/4` (4 is the size of an integer in bytes). The first of these is the block number of the next block on the free list. The remaining entries are block numbers of additional free blocks (whose contents are assumed to be meaningless). While the system is running, you will want to keep a copy of the first block of the free list in memory.

File Structure

Each file in the system is described by an inode (Figure 12.9) and is defined by the following class:

```
class Inode {
   // size of an inode in bytes
   public final static int SIZE = 64;
   int flags;
   int owner;
   int fileSize;
   public int pointer[] = new int[13];
}
```

If the `flags` field is 0, is unused. (In a real file system, the bits of this integer further distinguish different types of file—directories, regular files, symbolic links, and the like—and indicate permissions. You do not have to implement these features. Similarly, you may ignore the `owner` field.) The `fileSize` field indicates the current size of the file, in bytes.

Block 0 of the disk is the superblock, while blocks 1 through `SuperBlock.iSize` are packed with inode blocks. Each inode block contains `Disk.BLOCK_SIZE/Inode.SIZE` inodes. Inodes are numbered consecutively starting at 1—not 0! Hence, block 1 of the disk contains inodes `1..Disk.BLOCK_SIZE/Inode.SIZE`, and so on. Files are referenced by these numbers (called *inumbers*). In a real file system, directory files are used to translate mnemonic names to inumbers, but for this project, we will use inumbers directly. We represent an inode block with the following class:

```
class InodeBlock {
   Inode node[] = new Inode[Disk.BLOCK_SIZE/Inode.SIZE];

   public InodeBlock() {
      for (int i = 0; i < Disk.BLOCK_SIZE/Inode.SIZE; i++)
         node[i] = new Inode();
   }
}
```

The remaining blocks may be allocated as direct or indirect blocks or placed on the free list. They are collectively known as **data blocks**. The

```
public interface FileSystem {
   public int formatDisk(int size, int iSize);
   public int shutdown();
   public int create();
   public int open(int inumber);
   public int inumber(int fd);
   public int read(int fd, byte[] buffer);
   public int write(int fd, byte[] buffer);
   public int seek(int fd, int offset, int whence);
   public int close(int fd);
   public int delete(int inumber);
}
```

Figure 12.3 The `FileSystem` interface.

data blocks containing the contents of the files are called **direct blocks** (Figure 12.9). The `pointer` array in the `Inode` class point—either directly or indirectly—to these blocks. The first ten pointers point to direct blocks. The eleventh pointer—`pointer[10]`—points to an indirect block. This indirect block contains pointers to the next `Disk.BLOCK_SIZE/4` direct blocks of the file. (`IndirectBlock.java` is available online.) `pointer[11]` points to a double indirect block. It is filled with pointers to indirect blocks, each of which contains pointers to data blocks. Similarly, the final pointer—`pointer[12]`—points to a triple indirect block. It is important to note that the actual size of the file is determined by the `fileSize` field of the inode, not by the pointers.

A null pointer (either in the inode or in one of the indirect blocks) may indicate a hole in the file. For example, if the `fileSize` field indicates that there should be five blocks, but `pointer[2]==0`, then the third block constitutes a hole. Similarly, if the file is large enough and `pointer[10]==0`, then blocks 11 through `POINTERS_PER_BLOCK + 10` are all holes. An attempt to read from a hole acts as if the hole were filled with 0s; an attempt to write to a hole causes the hole to be filled in. Blocks are allocated as necessary and added to the file. Holes are created by seeking beyond the end of the file and then writing.

Other Disk Operations

The data structures `SuperBlock`, `InodeBlock`, and `IndirectBlock` are all the same size, so they may be written to or read from any disk block. For convenience, we provide three overloaded versions of `read()` and `write()` to the `Disk` class which allow read and write operations of the inode blocks, indirect blocks, and superblock.

File System Operations

For this project, you must implement the ten methods in the `FileSystem` interface illustrated in Figure 12.3. A description of each method follows:

- `formatDisk()` initializes the disk to the state representing an empty file system: It fills in the superblock and links all the data blocks into the free

list. The `size` field represents the number of disk blocks in the file system, and `iSize` represents the number of inode blocks.

- `shutdown()` closes all open files and shuts down the simulated disk.
- `create()` creates a new empty file and `open()` locates an existing file. Each method returns an integer in the range from 0 through 20 called a **file descriptor** (**fd** for short). (The upper limit of 20 is reasonable, since generally a process has a limited number of files open at any one time.) The file descriptor is an index into an array called a **file descriptor table** representing open files. Each entry is associated with one open file and also contains a **file pointer**, which is initially set to 0. The argument to `open()` is the inumber of an existing file. The method `inumber()` returns the inumber of the file corresponding to an open file descriptor.
- The methods `read()`, `write()`, `seek()`, and `close()` behave similarly to their UNIX counterparts. The `read()` method reads up to `buffer.length` bytes starting at the current seek pointer. The return value is the number of bytes read. If there are fewer than `buffer.length` bytes between the current seek pointer and the end of the file (as indicated by the `size` field in the `Inode` class), only the remaining bytes are read. In particular, if the current seek pointer is greater than or equal to the file size, `read()` returns 0, and the buffer is unmodified. (The current seek pointer cannot be less than 0.) The seek pointer is incremented by the number of bytes read.
- The `write()` method transfers `buffer.length` bytes from `buffer` to the file starting at the current seek pointer, advancing the seek pointer by that amount. Note that we allow the seek pointer to be greater than the size of the file: In this situation, holes may be created.
- The method `seek()` modifies the seek pointer as follows:

```
public static final int SEEK_SET = 0;
public static final int SEEK_CUR = 1;
public static final int SEEK_END = 2;
. . .

switch (whence) {
   case SEEK_SET: seekPointer = offset; break;
   case SEEK_CUR: seekPointer += offset; break;
   case SEEK_END: seekPointer = file_size + offset; break;
}
```

In case 0 (`SEEK_SET`), the offset is relative to the beginning of the file. In case 1 (`SEEK_CUR`), the offset is relative to the current seek pointer. In case 2 (`SEEK_END`), the offset is relative to the end of the file.

For cases 1 and 2, the value of the parameter offset can be positive or negative; however, the resulting seek pointer must always be positive or 0. If a call to `seek()` results in a negative value for the seek pointer, the seek pointer is unchanged, and the call returns −1. Otherwise, the value returned is the new seek pointer, representing the distance in bytes from the start of the file.

- The method close() writes the inode back to disk and frees the file table entry.
- The delete() method frees the inode and all of the blocks of the file. An attempt to delete a file that is currently open results in an error.
- shutdown() closes all open files, flushes all in-memory copies of disk structures out to disk, calls the stop() method on the disk, and prints debugging or statistical information.

File Table

The **file table** is a data structure for keeping track of open files. For each file, you will need a pointer to an in-memory copy of its inode, its inumber, and its current seek pointer. The code for FileTable.java is provided for you and is available online. However, it is important that you fully understand how to use it from the FileSystem interface. In particular, understand the methods for allocating and freeing slots in this table, determining whether a file descriptor is valid, and accessing the data associated with a file descriptor.

The file table uses a byte array called bitmap that indicates whether the specified index represents an open file. bitmap is an array of 0s and 1s; 1 denotes an open file and 0 denotes an open slot. The file table includes a file descriptor for each file. FileDescriptor.java (available online) contains data about a file including its inode, inumber, and current seek pointer. The file table maintains an array of FileDescriptor objects. The index in the bitmap array corresponds to the file descriptor object in fdArray with the same index (denoted by parameter fd). Each file descriptor object is an instance of the class FileDescriptor that maintains information about the specified file.

Implementation Hints

We recommend breaking down this project into smaller, more manageable tasks. Below is one strategy that lists the individual tasks in the approximate order required:

1. **Free-space management**. Write methods for allocating and freeing disk blocks. Also write the portion of formatDisk() that builds the free list in the first place.
2. **Block access within a file**. Write a method that takes an inode and a block offset within the file and returns a pointer to the block number of the corresponding block. The method should have a boolean argument fillHole that specifies what to do if the indicated block does not exist. If the block does not exist and fillHole is false, the method should simply return 0; if fillHole is true, the method should allocate a block, add it to the file, and return its block number. The first version is appropriate for read(), and the second is appropriate for write(). You might want to first write and debug this method for small files that can be stored entirely with direct blocks. Later, you can modify it to handle large files

as well. For large files, if `fillHole` is true, you may need to allocate one or two indirect blocks and add them to the file.

3. **Accessing inodes**. You will need methods to read a specific inode from disk (given its inumber) and write back a modified inode. Remember that you can only read and write whole blocks, so to write an inode, you will have to read the block containing the inode, modify the inode, and then write the block back out. You will also need a method to allocate inodes.

4. **Reading and writing arbitrary ranges**. At this point, implementing `read()` and `write()` should be straightforward. An individual read request may touch parts of several blocks, so you will need a loop that reads each of the blocks and copies the appropriate portion of it into the appropriate part of the buffer argument of the read call. The implementation of `write()` is slightly more complicated; if a block is only partially modified, you have to read its old value, copy data from the client's buffer into the appropriate portion of the block, and then write it back out.

5. **Miscellaneous**. Fill in the remaining methods of the `FileSystem` interface. Perhaps the only nontrivial remaining piece is `delete()`, which must return all the data and indirect blocks to the free list and clear the flags field of the inode. It must also check that the file is not currently open.

Testing

You should thoroughly test all ten required methods in your implementation of `FileSystem`, including creating, reading, writing, closing, reopening, and clearing all sorts of files (small, large, filled with holes, and so forth). You should also test the error checking in your code. The main program we supply online —`TestFS.java`—should be helpful. Consult the **README** file for further instructions for testing your implementation.