Lab/Project 4
DeFiler: A Simple File System in Java

CPS 210: Operating Systems

Introduction

This lab asks you to build a simple multi-threaded file system called Devil File System, or DeFiler.

DeFiler presents an API for its clients to create and share files. We call these files *dfiles*. Like a Unix file, a dfile is a sequence of bytes with a name. DeFiler clients can create, destroy, read, and write dfiles.

The set of dfiles (and related metadata) existing at any given time in a DeFiler instance is called a *volume*. DeFiler stores each volume in a logical disk, which is really a file on the underlying host. We call this file the host volume file or *virtual disk file* (VDF).

Like any good file system, DeFiler can support multiple clients. For example, DeFiler could run within a server and accept requests over a network. But for this lab, we will drive DeFiler by running it as a library within test programs. A test program creates multiple threads that call the DeFiler APIs. These threads are the *clients* of DeFiler. We run the test programs on any Java-capable computer, called the *host*. A running test program is an *instance* of DeFiler: it contains the DeFiler code and active data structures that represent the current state of a DeFiler volume backed by a VDF.

DeFiler consists of three layers of functionality, as described below. You are to implement the three layers and test that your code works correctly to provide the complete DeFiler functionality in a way that is safe and correct for multiple client threads.

This handout defines the APIs and behavior requirements for the three layers of DeFiler. As with the other labs in this course, the handout constitutes the complete and authoritative set of requirements for the lab. We may offer additional hints, suggestions, possibly useful code, or other assistance in an effort to maximize learning and minimize pain. However, any hints, suggestions, or code that we offer after publishing this handout do not change the requirements for the lab. You may ignore our attempts to help you, or not, as you see fit. If you find any ambiguity in the handout you are free to resolve it as you see fit. If an ambiguity is found, then we may suggest an interpretation, but you are not required to resolve any ambiguity in the way that we suggest.

Overview of the Three Layers of DeFiler

DeFiler has three layers of file system functionality roughly similar to the structure of a classic Unix kernel. Within each DeFiler instance, each layer is a Java object: a singleton instance of a Java class that implements the interface and functionality for the layer.

DeFiler manages the VDF as a pool of blocks to store the data and the metadata for the dfiles in the volume, in the same way that a real file system uses a real disk. Like every file, the VDF is a sequence of bytes, which may also be viewed as a sequence of blocks of some fixed *blocksize*. Each block in the VDF is named by a *block number* (blockID). The block number is an integer that is the block's offset or index in the VDF, when the VDF is viewed as an array of blocks of size blocksize.

Lowest layer: VirtualDisk. The lowest layer of DeFiler is a singleton instance of a class implementing interface **VirtualDisk**. It exposes the VDF as a block device (a disk). We call this layer the disk layer or device layer, or VirtualDisk layer, or just the driver. The driver API supports reads and writes of blocks in the VDF, named by their block number.

Middle layer: DBufferCache. The middle layer of DeFiler is a singleton instance of a class implementing abstract class **DBufferCache**. All access to the VDF is through the buffer cache, which manages buffering and caching of blocks from the VDF. It maintains a fixed number of memory buffers to store copies of the block contents in memory. For each block buffer, there is exactly one instance of a Java **Dbuffer** object (an instance of a class implementing interface **Dbuffer**) that describes the contents and status of the buffer. For example, the Dbuffer object names the VDF block stored in the corresponding buffer, if the buffer is associated with a block. Each VDF block is associated with at most one buffer/Dbuffer pair at any given time.

The DBufferCache API allows threads to find the Dbuffer associated with a given block, if it is in the cache, or to allocate a Dbuffer for the block if it is not in the cache. Once a Dbuffer exists for the block, the Dbuffer methods allow a thread to initiate I/O on the block: the thread can request the device layer to read the block contents from the VDF into the buffer, or write the contents of the buffer to the corresponding block of the VDF.

I/O operations at the VirtualDisk device driver layer operate on Dbuffer objects. Each I/O operation applies to a single Dbuffer. The driver interface is asynchronous: a read or write call posts the requested operation and Dbuffer on a request queue and returns immediately. The driver calls back to a method of Dbuffer when an I/O operation completes on the Dbuffer. Each Dbuffer has at most one pending I/O at any given time.

The Dbuffer maintains status information about the block, e.g., whether or not device I/O is in progress on the block. Other Dbuffer methods allow a thread to wait on a Dbuffer for pending I/O to complete.

Top layer: DFS. The top layer of DeFiler is a singleton instance of a class implementing interface **DFS**. It has methods for client threads to create dfiles, write data into dfiles, read data from dfiles, and destroy dfiles. The DFS layer is responsible for managing the data structures and metadata that represent the state of a DeFiler volume and the dfiles it contains: the lower layers support only I/O and caching of raw blocks in the VDF, without regard for their contents or the meaning of their contents. Each test program first calls a DFS.init method to initialize the internal data structures of the DeFiler instance.

How do we run/test DeFiler?

You will design and write your own test programs. We may also choose to provide test programs, and you may share test programs freely among groups if you choose to do so. You may submit your test programs, but you are not required to do so. But your DeFiler code should work correctly for any correct test program.

DeFiler maintains persistent state in the VDF: *persistent* just means that the data is preserved across runs. Each test program run (instance) uses exactly one VDF. The single VDF is the only persistent state for an instance.

It should be possible to run test programs multiple times in a sequence using the same VDF, or even to run different test programs in sequence over the same VDF. For example, if a program run **A** exits cleanly, and a second test program run **B** uses the same VDF, then **B** sees the volume state left by **A**, including any dfiles created by **A** and/or written by **A** and left in the volume before **A** exited.

DeFiler does not support concurrent sharing by multiple test instances. At most one test program/process/instance can run with a given VDF at any given time. It will not work for two test instances to run concurrently using the same VDF: it fails because DeFiler must synchronize client access to the underlying disk (VDF), and DeFiler has no mechanism to synchronize among client threads running in two or more different test programs/processes/instances/JVMs.

What does DeFiler do?

The top-layer DeFiler API seen by clients is the class DFS method interface. The API is simple and has many limitations to simplify the lab.

- **There is no symbolic naming**. There are no directories and no symbolic names for files. Instead, each dfile is named by a **DfileID**, whose value is a positive integer. At most one dfile is named by a given name (DfileID value) at any given time. The system (DFS layer) assigns the name for a newly created dfile, by allocating a DfileID value that is not currently in use for any other file in the volume. It is acceptable for DeFiler to impose some fixed maximum number of dfiles that may exist at any one time, not less than 512.

- **There is no seeking or random access**. All reads and writes on a dfile start at offset zero. Since every write starts at offset 0, the size of a dfile is the size of the largest write to the dfile. Each read should return at most *size* bytes, where size is the size of the dfile. The size is not necessarily a multiple of the blocksize.

- **There are no large dfiles**. Files have variable size up to some fixed maximum size, which is not less than 50 blocks.

- **There is no failure atomicity for disk writes**. For this lab, unexpected exit (i.e., failure) of a DeFiler instance or its test program may discard any incomplete writes and even leave the VDF in a corrupted state. Any writes left as dirty data in the cache (DBlockLayer buffers) die there in the block cache when the process exits. If a later test program uses the same VDF, it does not see the data from these writes in the VDF: the writes are lost. A test program running on a corrupted VDF may fail because the underlying VDF is in an unexpected state. Your initialization code should check for any inconsistencies in the VDF state. (See below.)

Even so, DeFiler as defined here implements much of the core functionality of a real file system. It supports multiple concurrent clients (e.g., test program threads). It supports sharing of dfiles among clients. It supports caching to speed access to recently used dfiles. And it manages allocation of space (blocks) for dfiles from the underlying pool of storage blocks in the VDF, which serves as a virtual disk. That means that DeFiler must keep track of which blocks in the VDF are allocated and which are free, and which blocks store the data for each dfile.

The VDF Layout

The VDF is an array of blocks. The blocksize is a fixed power of two. It is a compile-time constant. No part of your code should assume any particular value for the blocksize.

Each low-level I/O operation on the VDF reads or writes exactly one block. Unlike a real disk, you may grow the VDF by writing blocks at successively higher block numbers, even up to the maximum file size supported by Java and the underlying host.

This lab does not restrict the format and meaning of the data stored in the blocks of the VDF. However, we strongly recommend the following VDF format for simplicity.

- Block 0 is left empty, or may contain metadata of your choice.

- Following block 0 is an *inode region* consisting of one or more consecutive blocks. The data in the inode region is an array of fixed-size structures which we call *inodes*, in analogy with Unix file system designs. Each inode consists of some metadata about a specific dfile (e.g., the dfile's size), including a block map for the dfile. The block map for a dfile is an array of disk block numbers corresponding to the logical blocks of the dfile.

- The blocks following the inode region hold the data contents of various dfiles.

This recommended VDF structure leaves many questions open.

How to track usage of the blocks and inodes in the VDF? A real file system might also include metadata structures that represent which blocks and inode slots in the VDF are free and which are allocated. However, for a small and simple file system (and for this lab) it is acceptable to scan the inode region during initialization and assemble free maps in memory.

How to determine if the VDF is corrupt? An initialization scan might also check for corruption in the VDF. But what are the invariants on the VDF structure that should be checked? For the format above, we might presume that each existing dfile (DFileID) has exactly one inode, that the size of each dfile has a legal value, that the block maps of all dfiles have a valid block number for every block in the dfile (as defined by its size), and that no data block in the VDF is listed in the block map of more than one dfile.

How to know what dfiles exist in the VDF? A real file system has a directory tree rooted in a root directory, so that it is possible to discover all files in the volume by traversing the directory tree recursively from the root directory. But DeFiler does not have directories. A good option is to maintain a list of dfiles in memory in the DFS layer, and build it during initialization by scanning the inode region for valid inodes.

Requirements

The Appendix includes the required Java interfaces for **DFS**, **DBufferCache**, **VirtualDisk**, and **Dbuffer**. You are to implement these interfaces in your classes. They should work correctly with multiple client threads. DeFiler should be free of races, starvation, crashes, deadlocks, and storage leaks. It should recycle memory and storage correctly when dfiles are destroyed.

We expect that the DeFiler respects the specified layering and that the layers interact only through these published interfaces. If you have a good reason to add methods or interactions among the layers, it is acceptable to do so.

Constraints. All access to the VDF passes through the buffer cache. Also, inodes are small relative to blocks: your implementation should store multiple inodes within each block of the inode region. Your code should not depend on any specific block size or the number of blocks in the VDF. But you may assume these values are defined in advance (compile time) and that they are “reasonable”.

Your DeFiler should implement a consistent file model. That means that at least the following correctness properties (invariants) hold true within each instance. If a dfile is created, then it continues to exist until it is destroyed. At any given time at most one dfile may exist for a given DFileID. A read on a dfile returns the data that was most recently written to the dfile.

These correctness properties must also hold true for a sequence of DeFiler runs or instances using the same VDF. That is, when a test program exits cleanly, the state of the volume is stored persistently in the VDF, and the next test program run with the same VDF starts with the volume in the last saved state. As noted earlier, it is not necessary to implement failure atomicity: it is acceptable for a failure to discard writes or even corrupt the volume state.

DeFiler should enforce atomicity for individual read and write operations: read or write operations on a given dfile are serialized (mutual exclusion). For example, two concurrent writes to the same region of a dfile does not result in interleaved data from both writers within that region. Also, a read that is concurrent with a write should not observe any intermediate state of the file from a partly executed write.

DeFiler should implement a reasonable caching policy approximating a **least recently used** (LRU) replacement policy. Recently read or written blocks should remain in memory in the block buffer cache, so that subsequent reads or writes are faster, for some access patterns. Be aware that synchronization for asynchronous block I/O with LRU eviction can be a difficult synchronization problem.

For this lab you may use any prepackaged Java library classes that you wish, including classes that have their own internal synchronization. We believe that for this lab it will be easier, simpler, and more instructive to implement your own synchronization.

Appendix: Overview of DeFiler Interfaces

DFS

```
/* creates a new dfile and returns the DFileID */
public DFileID createDFile();

/* destroys the dfile named by the DFileID */
public void destroyDFile(DFileID dFID);

/* reads contents of the dfile named by DFileID into the ubuffer
 * start read at dfile offset 0
 * starting from ubuffer offset startOffset; at most count bytes are transferred
 */
public int read(DFileID dFID, byte[] ubuffer, int startOffset, int count);

/* writes to the file named by DFileID from the ubuffer
 * start write at dfile offset 0
 * starting from ubuffer offset startOffset; at most count bytes are transferred
 */
public int write(DFileID dFID, byte[] ubuffer, int startOffset, int count);

/*   List DFileIDs for all existing dfiles in the volume
 */
public List<DFileID> listAllDFiles();

/* Write back all dirty blocks to the volume, and wait for completion.
*/
public void sync();
```

DBufferCache

```
/* Get buffer for block specified by blockID
    The buffer is “held” until the caller releases it.
    A “held” buffer cannot be evicted: its block ID cannot change.
*/
public DBuffer getBlock(int blockID);

/* Release the buffer so that it may be eligible for eviction.
 */
public void releaseBlock(DBuffer dbuf);

/* Write back all dirty blocks to the volume, and wait for completion.
*/
public void sync();
```

DBuffer

```
/* Start an asynchronous fetch of associated block from the volume */
```

public void **startFetch**();

/* Start an asynchronous write of buffer contents to block on volume */
public void **startPush**();

/* Check whether the buffer has valid data*/
public boolean **checkValid**();

/* Wait until the buffer has valid data (i.e., wait for fetch to complete) */
public boolean **waitValid**();

/* Check whether the buffer is dirty, i.e., has modified data to be written back */
public boolean **checkClean**();

/* Wait until the buffer is clean (i.e., wait for push to complete) */
public boolean **waitClean**();

/* Check if buffer is evictable: not evictable if I/O in progress, or buffer is held. */
public boolean **isBusy**();

/* Reads into the ubuffer[ ] from the contents of this Dbuffer dbuf.
 * Check first that dbuf has a valid copy of the data!
 * startOffset is for the ubuffer, not for dbuf.
 * Reads begin at offset 0 in dbuf and move at most count bytes.
 */
public int **read**(byte[] ubuffer, int startOffset, int count);

/* Writes into this Dbuffer dbuf from the contents of ubuffer[ ].
 * Mark dbuf dirty! startOffset is for the ubuffer, not for dbuf.
 * Writes begin at offset 0 in dbuf and move at most count bytes.
 */
public int **write**(byte[] ubuffer, int startOffset, int count);

VirtualDisk

/*
 * Start an asynchronous I/O request to the device/disk.
 * The blockID and buffer array are given by the DBuffer dbuf.
 * The operation is either READ or WRITE (DiskOperationType).
 */
public void startRequest(DBuffer dbuf, DiskOperationType rw) throws…;

VirtualDisk upcalls DBuffer.**ioComplete**() when operation is complete.