Project 4, Phase 2 Report

Section 1:

Name: Rishi Dewan

Lindsey O'Niell

Slip Days Used: 3 (Overall total slip days used: 4)

To run our unit test (for PTree) file:

```
$: make
$: java PtreeUnit
```

To see the output of our PTree Unit Test, refer to the PTreeUnitOutput file.

Section 2:

ActiveTransactionList.java - This is the class that stores the ActiveTransactionList for the Atomic Disk (whose underlying data structure is a ConcurrentHashMap with a <TransID,Transaction> key value pair). Transactions that have been committed are removed from this list.

ADiskUnit.java - Provides some relative testing for the classes and functios of the atomic disk.

ADisk.java - The core Atomic Disk class which is in essence a wrapper for the Disk class provided. It uses the disk to issue read/write requests to disk by using a redo log first, and also does the actual implementation of Transaction commits/aborts.

Callbacktracker.java - Implements the Diskcallback interface which handles requests in a synchronous manner and then returns the DiskResults of each request. This class uses a tag system in which requests come in in the form of integers called tags, and we use condition variables to make the CallbackTracker wait for and not wait for specific tags.

FlatFS.java – This class is for an implementation of the Flat File System which is reliant on its PTree instance. This system writes ranges of bytes from/to blocks on disk. Our FlatFS java file was not fully complete at the time of turn in, as the write method was not completely finished.

LogStatus.java - Class that keeps track of the head, tail, and checkpoint of the log. It waits on the condition whether there are enough log sectors to allocate.

PTree.java – This class provides the abstraction of the multi-level persistent level tree. In this class, we implement the reading/writing of data using data blocks/indirect blocks/double indirect blocks.

PTreeUnit.java – This is the class where our sanity checks for our PTree class is written. Here, we do some sample reads and writes and then check our results using assert statements to determine whether the reads/writes are executed correctly.

Transaction.java - The class that defines the concept of a transaction. Each transaction class has a Transaction state (defined as part of an enum), a list of sector locations to write to, a list of byte arrays which contains data to write to disk.

TransID.java - IMPORTANT NOTE. PLEASE READ: We felt that using this class was not really necessary, as it would only serve as a wrapper for an integer (if only Java had typedefs). So we used ints as representations of TransIDs.

WriteBackList.java - This class stores the committed Transactions that need to be written to disk courtesy of the WriteBack Thread.

WriteBackThread.java - A created subclass of the Thread class that pulls Transactions off the WriteBack list and writes them to disk using the Disk's startRequest method. We pass in an ADisk object so that we can allow the synchronized interface between the WriteBack Thread and the WriteBack List.

Section 3:

The high-level design of our ADisk class consists of creating transactions by appending Transaction objects to the data structure (ConcurrentHashMap) of the ActiveTransaction List, appending them to the Writeback List (Queue) after serializing (converting them to byte[]) the Transaction objects upon calling commit on them and writing the serialized renditions to the log. We also created a Writeback Thread (extends Thread) which pulls Transactions from the WriteBackList and waits for new Transactions to enter the Writeback List. The serialized Transaction is represented as such (each sector is separated by a bar (|)):

| METADATA | DATA-1 | DATA-2 | DATA-3 | .... | DATA-N | COMMIT |

Where METADATA consists of 3 things: The magic number for the header, which lets the log scanner know that it has come across a potentially committed transaction, the transID of this transaction, and the N sector locations to which the transaction is going to write to. The COMMIT is comprised of a magic number for commit and the transID.

The way that we used locks and threads in our ADisk was organized in a way that almost every class we implemented had its own lock, and the methods of each class is locked at the beginning and unlocked at the end. We synchronize threads in the following ways in each class:

The high-level design of PTree is a design that we based on ADisk. Our ADisk stores a maximum of 512 trees, each tree being 128 bytes. Hence, 4 trees per sector, then hence 128 sectors to store all trees. This is a figure for the design of our ADisk + PTree:

[Log | FreeList | Array of TNodes | Usable Disk Space]

Our design reserves 1025 sectors for the Log, 2 Sectors for the on-disk FreeList and 128 sectors for the Array of TNodes (Totaling 1155 sectors reserved). The design of our free Node is represented by this figure:

[Metadata | Custom Metadata | 8 Direct Pointers | Indirect Pointer | Double Indirect Pointers]

By Pointers, we mean Pointers to data blocks on disk. The total size of our TNode is 128 bytes:
64 bytes for the metadata, 24 bytes for our custom metadata, 32 bytes for the direct pointers, and 4 bytes each for the indirect and double indirect pointer.
Considering that our design states that a TNode is 128 bytes, a sector can hold 4 TNodes evenly without any sector fragmentation.

Flat File System is dependent upon the PTree. Most of the operations of the Flat File System uses the PTree instance. We considered the inumbers to be the TNodes of the PTree and the files on disk to be the trees themselves. Whenever we attempt to write/read a file we read segments of blocks on Disk based on the offset and the number of bytes given.

Section 4:

The overall design of our PTree was primarily dependent on the design of our ADisk, which PTree's atomicity is dependent on ADisk's atomicity. This is true because PTree uses it's underlying instance of ADisk to perform the atomic transactions done to disk. Given a series of reads and writes to blocks on Disk, then we use our ADisk instance to execute these reads/writes in the form of a transaction. In order to do atomic writes to data blocks, we begin a transaction using ADisk.beginTransaction(). Afterwards, we will use the writeData and readData methods to do the reads/writes to disk blocks. When all of our data we intend to read/write has been collected in the transaction, we will then commit the transaction using ADisk.commitTransaction(). The PTree data will be written in the manner similar to the way normal sectors are written to in a transaction: written to the log (in case of the need to recover) and then to the write back list. The atomicity of PTree is insured using the atomicity of ADisk. Whenever a transaction is not successfully committed to the log and disk, it will be as if the transaction never occurred, demonstrating the ACID property for atomicity. In summary, because we designed our ADisk to handle writes atomically, we used it to make PTree atomic as well. We based the FlatFS's atomicity based on Ptree's atomicity.

Section 5:

At a high-level, we tested the PTree using a testing class called PTreeUnit. We tested the functionality of the method of PTree using the following methods:

testFreeList() - tests the functionality of the free to show what blocks are free/used. This test passed.

TestFreeList2() - iterator through the freeList and change the value of the bits in the FreeList and test whether they are free or not. This test passed.

testCreateTree() - Creates a transaction, then creates some Trees (Tnodes), then commit them. The Test passed.

TestCreateTree2() - Create a transaction, then create the maximum number of writes under that transaction. We then commit the transaction, and check to see if the corresponding sectors that are a part of the TNode in the TNode Array are considered used. This test passed.

TestWriteData() - Create a transaction, then perform a series of reads/writes of block buffers filled with numbers into disk. We write to direct blocks, indirect blocks, and double indirect blocks based on the blockIDs we write to. We then pull off data from the sectors we just wrote to and compare the values that we have written to on disk. The test passed.

TestReadData() - What we do is do more reads from the blocks that we wrote to in testWriteData(). The test passed.

TestMultipleWrites() - We create a transaction, then create some TNodes where we perform writes to disk using those TNodes. Additionally, we deleted a TNode in which we free the space of the TNode using deleteTree(). We then check to see whether the space was actually freed (i.e. filled with a zero buffer). We then check the rest of the writes performed in other TNodes to check whether their values that were written to disk are correct. The test passed.