Homework 1

Due: 9/25/2017@11:59pm
Team size: 2

Homework 1 - Part 0

This part of assignment is to help you get started with the development environment that you'll be using for part 1. This assignment must be in teams of two.

Details

Operating System does not always perform memory and file management optimally. Databases explicitly organize records in memory pages, pages into "heap files", and keep track of which pages are in memory at any point in time. In the next part of this assignment you will become more acquainted with buffer management.

The homework projects for this class are based on Minibase, a fully functional instructional database created at the University of Wisconsin. The version we are using was written in Java. It will be possible to work on the assignments using command-line tools on the instructional machines. It will also be possible to use the Eclipse IDE on either the instructional machines or your own computer. (Eclipse is available for free at http://www.eclipse.org.) Eclipse is a fabulous environment to program and debug in, but it is fairly resource intensive, and will perform poorly on a heavily loaded machine.

The file you will need to download is Homework1_part0.zip. This is a complete eclipse project zipped into a single file, but you can also run everything from the command line. No matter whether you want to use the command line or eclipse, start by copying " Homework1_part0.zip " to the main directory of your CS257 account, and unzip it with the command: "unzip Homework1_part0.zip".

Command Line Instructions

- Change to the "Homework_0" subdirectory: "cd Homework_0"
- Compile the java source file, which is either:
  - (unix) javac -classpath "minibase-exceptions.jar:minibase-heap.jar:minibase-diskmgr.jar:minibase-global.jar" HeapFileScan.java
  - (windows) javac -classpath "minibase-exceptions.jar;minibase-heap.jar;minibase-diskmgr.jar;minibase-global.jar" HeapFileScan.java
- Then run the program with one of the following commands:
  - (unix) java -classpath "minibase-exceptions.jar:minibase-heap.jar:minibase-diskmgr.jar:minibase-global.jar:." HeapFileScan
  - (windows) java -classpath "minibase-exceptions.jar;minibase-heap.jar;minibase-diskmgr.jar;minibase-global.jar;." HeapFileScan
- Submit the output that appears on the command line.

Eclipse Instructions

- Start up eclipse. It may ask you for a workspace, choose any subdirectory in your account.
- You'll need to create a project, so select "File" -> "New" -> "Project"
    - In the wizard that appears, select "Java Project" as the type of project, and click "Next"
    - In the next panel, specify "Homework_0" as the project name, and click on the "Create project from existing source" button and specify the unzipped Homework_0 directory.
    - Click Finish.
- If you haven't used eclipse before, you may not see anything but a "Welcome" screen at this point. If so, select "Window" -> "Open Perspective" -> "Java". If you still don't see anything but the welcome screen, you may need to click the button in the upper right of the welcome screen.
- There should now be a navigation tree in the left side of the screen, containing "Homework_0".
- If "Project" -> "Build Automatically" is not selected, you will need to right-click on the "Homework_0" project in the package explorer, and select "Build Project" from the popup menu.
- Right-click on the "Homework_0" project in the package explorer, and select "Run As..." -> "Java Application". A selection box will appear, from which you must choose "HeapFileScan" as the class to run.
- The output from the program will appear in the console window, submit this text.

The HeapFileScan program uses the database's "heapfile" package to:

- create a database on disk.
- create two files in that database containing different numbers of records,
- then read through the files using a sequential scan to make sure they contain the right number of records.

Optional Section

For next part, it will help to know some more about how files work in the database. If you're inclined, spend some time using the debugger to see how files are created, records inserted, and files scanned. This is optional, but you might learn something.

Some background: first, a Minibase *database* is a fixed collection of of pages. All information stored in the database must fit within these pages. Initially these pages are allocated on disk, and a *buffer manager* migrates these pages back and forth to memory whenever they need to be read or written. In most applications, the number of pages that fit in memory is smaller than the size of the database on disk, so the buffer manager needs to be selective about what it keeps in memory (you'll be implementing a buffer manager in the next part of this homework

1) In the "HeapFileScan.java" file, the following initialization takes place:

```
Minibase.initBufMgr(new DummyBufMgr());
```

This creates a buffer manager for the database to use. In this case, we're using the

DummyBufferManager which is not smart, and just allocates main memory for every page, leading to lots of swapping on any large database. After the buffer manager is created, the database is created on disk with a given size. This needs to happen after the buffer manager is created, since the database needs to use the buffer manager to make sure certain pages with database catalog information get into memory, and then are written to disk.

```
Minibase.initDiskMgr(dbpath, 100);
```

Minibase keeps files, called *heapfiles*, as a set of pages that can be either on disk or in memory, and movement between disk and memory is performed by the buffer manager. Some of the pages contain only tuples, i.e., fixed length data records. Other pages contain directory information, indicating which database pages are part of the file. Some of the classes used to implement heapfiles are:

- heap.Tuple - the class that encapsulates a fixed length record with a specified number of fields, each field is either an integer, float, or fixed length string. This class can be serialized as a fixed length array of bytes, to put into a page. Each tuple starts with an array of data types and offsets, so that the structure of the tuple can be determined from the bytes.
- diskmgr.Page - this class represents a fixed size page that can moved between the database on disk and the buffer pool in memory. The contents of the page are raw data and not interpreted.
- heap.HFPage - this is a subclass of diskmgr.Page used for holding records (arrays of bytes). A record could be a tuple, or anything else that can be serialized as an array of bytes. Each record put into a *slot* in the page. Space is reserved at the beginning of each page for holding the number of slots used, amount of free space, and page IDs for the current, previous, and next page in the file. After this is an array of number pairs, one for each slot, listing the size and offset of each record slot. Each record can be a different size, though of course records may not be larger than the size of a page. In this implementation, the size/offset array grows up from the beginning of the page, and the record-occupied space grows down from the end up the page, with the middle always a contiguous area of free space. The HFPage class provides a way to iterate through the records stored in a page.
- heap.Heapfile - this class describes a database file. The file has a 2-level structure, with directory pages containing pointers to data pages, which contain the actual records. Both types of pages are implement as HFPage objects - the data pages contain whatever Tuples are added to the file, but the directory pages instead contain DataPageInfo objects (see below). When a heapfile is created, it has only a single directory page. Each time a record is added, the directory pages are searched for a data page with enough space to hold the new record. If none is found, a new data page is created, and an entry for that data page is added to the directory page. When directory pages are filled up, new ones are added as needed.
- heap.DataPageInfo - this class is similar to Tuple, in that it can be serialized into an array of bytes for storage in a page. While a Tuple is a general data structure, the DataPageInfo only holds 3 integers describing a page: the pageID, number of records, and available space in the page. DataPageInfo is more compact that Tuple, however, lacking the array of size/offsets at the beginning.

- heap.Scan - this class provides methods for doing a scan of a heapfile, iterating record-by-record over the file.

The source code for the heap package can be found on Canvas. If you're using eclipse, you can point it to this jar file when stepping through heap classes in the debugger.

Homework 1 - Part 1

Reading and writing pages from main memory to disk is an important task of a database system. Main memory is partitioned into collections of pages, and the collection of pages is called the *buffer pool*. The buffer pool is organized into *frames*, and each frame in the buffer pool can hold a page that is brought in from the disk. The *buffer manager* is responsible for bringing pages from disk to the buffer pool when they are needed, and writing pages back to the disk when they have been updated. The buffer keeps a *pin* count and *dirty* flag for each frame in the buffer pool. The pin count records the number of times a page has been requested but not released, and the dirty flag records whether the page has been updated or not. As the buffer pool fills, some pages may need to be removed in order to make room for new pages. The buffer manager uses a *replacement policy* to choose pages to be flushed from the buffer pool. The strategy used can greatly affect the performance of the system. *LRU* (least recently used), *MRU* (most recently used) and *Clock* are different policies that appropriate to use under different conditions.

Background

In this assignment, you will implement a simple buffer manager for the *Minibase* database system. In previous part, you should have already downloaded minibase, compiled it, and run it. Below we briefly describe the components of Minibase you need to know about for this assignment.

The global package (global-source.zip) contains definitions that will be used throughout the project. The most important class to become familiar with for this assignment is Systemdefs.java.
Systemdefs maintains the state of the minibase database, including the database on disk (JavabaseDB) and the buffer pool (JavabaseBM).

The diskmgr package (diskmgr-source.zip) manages the database on disk. The disk manager is already implemented for you; you will not need to make any modifications. Its two most important classes are Page.java, which implements a disk page, and DB.java, which provides an API to create and delete a database of pages (on disk) and read and write pages to that database. Your buffer manager will make use of this package, and in particular, the DB class.

The exceptions package provides the base definition of a minibase exception. The ChainException class maintains a stack of exceptions thrown from different components in minibase (such as disk manager and buffer manager). Many exceptions have already been defined that you can use to indicate various error conditions that can occur. You can also define additional subclasses of ChainException to handle specific error conditions if there is not already an existing exception.

Homework project description

Your job is to implement the buffer manager (in the bufmgr package) that will allow a higher-level minibase component to allocate and deallocate pages on a disk, as well as pin, read from and write to, and unpin disk pages in the buffer pool. Your buffer manager should be able to create a buffer pool with frames for disk pages and support MRU and Clock replacement policies.

Download Homework1_part1.zip to get started. This is a complete eclipse project zipped into a single file, but you can also run everything from the command line. The unzipped directory will contain a few things to help you get started:

The *bufmgr* directory: This directory contains class definitions that you will need to modify. It contains class definitions for the following classes:

- BufMgr.java: This is the class that implements the buffer manager. An initial implementation is provided for you that keeps the buffer in main memory. The class definition is provided, and the current implementation uses a hash table for pages and all of them reside in main memory -- each request for a new page simply adds to the hash table. You will need to fill in the details to pin, unpin, read and write pages to disk, and use a replacement policy to select pages to remove from the buffer pool when it is full.
- BufMgrFrameDesc.java: The buffer manager maintains an array of these frames to describe each page in the buffer pool.
- BufMgrReplacer.java: This is an abstract class that defines the strategy to replace pages in the buffer pool if it is full. You will need to extend this with 2 subclasses to implement different replacement policies: Clock and MRU.
- Clock.java: This class extends BufMgrReplacer.java to implement a Clock strategy. The class definition is provided, but you will need to fill in the details.
- MRU.java: This class extends BufMgrReplacer.java to implement an MRU strategy. The class definition is provided, but you will need to fill in the details.

The *tests* directory: This directory contains a driver program that you can use to test your buffer manager. It contains a single class:

- BMDriver.java. BMDriver extends the class TestDriver that is found in the global package. TestDriver is a generic test harness that you can extend for each of your homework assignments to test your code. The current implementation of BMDriver simply creates a buffer pool, runs any methods with the prefix 'test', and then exits.
- After you have unzipped the file, you can test that everything is working by running BMDriver as a Java application. You should see the following output:

```
Running Buffer Manager tests....

Invoking test: test1

Test 1 is not implemented.

Invoking test: test2

Test 2 is not implemented.

...Buffer Manager tests completed successfully.

Cleaning up …
Done. Exiting...
```

Logistics

This homework project will be carried out in teams of two. Please choose a partner as soon as possible. Create a README file and submit it with your project and mention the name and student id of group members.

Note that your project will be tested with respect to the functionality of the replacement policies, and the facilities that the Buffer Manager provides to the rest of the minibase components under a variety of workloads. Make sure you are extra careful about uncaught exceptions or null pointers lurking in "dark corners" of your code.

Submission Instructions

- Create a new directory named hw1.
- Inside that directory create a file named README where you should mention the members of your group (names and student id number), what works, and what doesn't. Include instructions to run your program.
- You should submit your eclipse project through Canvas.

Each group only needs to submit once. If there are multiple submissions, the last submission by any group member will be used for grading and the calculation of slip days.