Buffer Management Assignment
Database Implementation
CS 487/487

Introduction
In this assignment, you will implement a simplified version of the *Buffer Manager* layer of Minibase, without support for concurrency control or recovery. Learning objectives for this assignment are to familiarize yourself with the minibase programming environment and the concepts of disk and buffer management.

Where to Find Code
The code (in .tar and .zip files) is available on the course web site. The contents are:

- *bufmgr* directory/package
    - Contains a skeleton BufMgr.java which you will complete.
- *tests* directory
    - Contains BMT.java, the test code.
- *global* directory
- *diskmgr* directory
    - You will call methods in this directory
- output.txt
    - The output of BMTTest.java should look like this.

Getting Started in Eclipse
You may develop in whatever environment you choose. These are suggestions for eclipse.

All the files above are in the tarfile/zipfile in a directory called *bm*. *Extract* the bm directory from the tarfile/zipfile. *Start Eclipse* in the Java perspective (usually the default perspective). Click *File/New/Java Project*. Give it a name, click *Next*, click *Link Additional Source*, browse and add the bm directory, click *OK* and *Finish*, click *Finish.*

To build the application, click *Project/Build Project* or select *Project/Build Automatically*. You will get a few warnings because your code is a skeleton.

To run the BM test, on the left, in Package Explorer, highlight tests/BMTest.java. Then click *Run/Run As/Java Application*, and you will get an UnsupportedOperationException because your code is not complete.

Background
The best place to start understanding this assignment is in the text, Sections 9.3 and 9.4 about disk and buffer management. Minibase is structured as layers, except for the *global* package, called by all layers, and the *tests* package, which calls specific layers to test them. In this assignment, you will be given the *diskmgr* layer, which manages pages

on the disk, and you will write the next higher layer, *bufmgr*, which calls *diskmgr*. (Everything is not perfect in that a method in *diskmgr* must call a method in *bufmgr* at some point.) In the next assignment, you will write the index layer.

Internal Design: Buffer Pool, Frame Descriptors and Hash Map
Look at bufmgr/BufMgr.java. Look at the code and the comments. Your first task in the BufMgr constructor will be to initialize the buffer pool as an array of Page objects. Each object will be called a Frame.

Each frame has certain *states* associated with it. These states include whether the frame is dirty, whether it includes valid data (data which reflects data in a disk page), and if it includes valid data then what is the disk page number of the data, how many callers have pins on the data (the pin count), and any other information you wish to store, for example information relevant to the replacement algorithm. Be sure to store this information as efficiently as possible while preserving readability. You should store these states in a structure described by a separate class called *FrameDesc*. Call the structure *frametab*; I'll refer to it as the frame table, or frame descriptors.

Your *BufMgr* class will need to determine, very efficiently, what frame a given disk page occupies. You may wish to use Java's *HashMap* class to map a disk page number to a frame descriptor. The map will also tell you if a disk page is not in the buffer pool.

As you write methods such as pinPage it is very important, and a prime source of bugs, to keep the frametab and your hash map current.

Internal Design: Replacement Policy
In the pinPage method, you will need to figure out which page is best to replace. The call will be something like:

```
int frameno = replPolicy.pickVictim();
```

You will implement the Clock algorithm for this project. Section 9.4.1 discusses possible replacement policies, e.g. the widely used LRU policy. LRU has good performance in some cases. However, remembering which frame was used least recently requires some overhead. The Clock algorithm approximates LRU but has less overhead. Clock approximates LRU behavior by approximating the time of each page's last access by one bit, called the reference bit. More accurate approximations to LRU use more than one bit. We give a bit more detail here.

Each frame descriptor keeps a reference bit (refbit), which is true if the page has been referenced recently. It is set to true when the pincount is set to zero. (In class we also discussed a reference count which is an alternative to a reference bit).

Say we have N frames in the buffer pool, numbered 0 through N-1. The clock replacement policy keeps a current variable that indicates which frame is currently being considered for replacement. Frames are considered consecutively for replacement, wrapping back to 0 when from N-1 is reached.

When the clock needs to pick a victim, it first considers the current frame. If the current frame's state is invalid the frame is chosen. Otherwise, if the pin count is greater than 0, the frame is not chosen.

If the pin count is 0, then the reference bit is considered: if the reference bit is false, the frame is chosen. Otherwise, the reference bit is set to false and the next frame is considered.

But what if the above algorithm does not find a victim? If current makes a full sweep (i.e., it winds up back at the value it had to start out) without picking a victim, it should go around another time (since it's likely that now there will be frames with false refbits). If it makes another full sweep, we conclude that there are no available frames and we return an error.

Here is some possible pseudocode for the algorithm, with no comments:

```
int pickVictim ( ) {
  for(int counter = 0; counter < N*2; counter++) {
    if data in bufpool[current] is not valid, choose current;
    if frametab[current]'s pin count is 0 {
      if frametab [current] has refbit == true {
        set frametab [current]'s refbit = false
      } else {
        return current
      }
    }
    increment current, mod N
  }
  We couldn't find an available frame, so return an error
}
```

Looking at the Code

Look over the code and comments for *diskmgr* and *bufmgr* packages. Review the code for Minibase.java and DiskMgr.java and BufMgr.java. Along the way, you will discover the code in the *global* package.

Your primary task in this assignment is to fill in the stubs in BufMgr.java so that the test BMTest.java runs successfully (see the file output.txt in the tar/zip file you are given), and so that the specifications above and in the documentation are met. In your solution, you may add any other classes and/or files to the bufmgr package. In particular, you are required to support the replacement policy in a separate file called Clock.java, and the frame descriptors in a separate file called FrameDesc.java, as described below. You may also add to the global package if you find it necessary.

Skim the code for DMTest.java, in the tests package. This may be useful to see how to call the methods in DiskMgr, and to verify that DiskMgr actually works. Look more care-fully at BMTest.java.

Note that in TestDriver.java DB_SIZE is set to 10000 and BUF_SIZE is set to 100. You may wish to set these artificially lower for debugging purposes.

What to Turn In
You will demonstrate your program for grading purposes. You will also need to turn in source code - including the following file and any files you have changed. Files that must be turned in are: BufMgr.java FrameDesc.java and Clock.java.

Please turn in the the files on D2L; github links are ok.