CS 430: DATABASE MANAGEMENT SYSTEMS

Assignment 6: Buffer Manager
Due: April 14, 2009

1 Introduction

This original assignment is designed by University of Wisconsin-Madison. In our revised assignment, you will implement a simplified version of the Buffer Manager layer, without support for concurrency control or recovery. You will be given the code for the lower layer, the Disk Space Manager.

You should begin by reading the chapter on Disks and Files, to get an overview of buffer management. This material will also be covered in class. In addition, HTML documentation is available for Minibase, which you can read using Internet Browser. In particular, you should read the description of the DB class, which you will call extensively in this assignment. The Java documentation for the *diskmgr* package can be found at javaminibase/javamini_hwk/handout/BufMgr/javadoc/packages.html. You should also read the code under diskmgr/ carefully to learn how package is declared and how Exceptions are handled in Minibase.

This assignment is designed for LINUX-SPECIFIC platform. The java codes you turn in should work on LINUX machine. We will only accept the codes that work on LINUX machine. You can see the particular code in BMTest.java, etc.

2 The Buffer Manager Interface

The simplified Buffer Manager interface that you will implement in this assignment allows a client (a higher level program that calls the Buffer Manager) to allocate/de-allocate pages on disk, to bring a disk page into the buffer pool and pin it, and to unpin a page in the buffer pool.

The methods that you have to implement are described below:

```
public class BufMgr {

  /**
   * Create the BufMgr object.
   * Allocate pages (frames) for the buffer pool in main memory and
   * make the buffer manager aware that the replacement policy is
   * specified by replacerArg (i.e. Clock, LRU, MRU etc.).
   *
```

```
   * @param numbufs: number of buffers in the buffer pool.
   * @param replacerArg: name of the buffer replacement policy.
   */

  public BufMgr(int numbufs, String replacerArg) {};
/**------------------------------------------------------------*/

  /**
   * Pin a page.
   * First check if this page is already in the buffer pool.
   * If it is, increment the pin_count and return a pointer to this
   * page.  If the pin_count was 0 before the call, the page was a
   * replacement candidate, but is no longer a candidate.
   * If the page is not in the pool, choose a frame (from the
   * set of replacement candidates) to hold this page, read the
   * page (using the appropriate method from {\em diskmgr} package) and pin it.
   * Also, you must write out the old page in chosen frame if it is dirty
   * before reading new page.  (You can assume that emptyPage==false for
   * this assignment.)
   *
   * @param pageId: page number in the minibase.
   * @param page: the pointer point to the page.
   * @param emptyPage: true (empty page); false (non-empty page)
   */

  public void pinPage(PageId pin_pgid, Page page, boolean emptyPage) {};
/**------------------------------------------------------------*/

  /**
   * Unpin a page specified by a pageId.
   * This method should be called with dirty==true if the client has
   * modified the page.  If so, this call should set the dirty bit
   * for this frame.  Further, if pin_count>0, this method should
   * decrement it. If pin_count=0 before this call, throw an exception
   * to report error.  (For testing purposes, we ask you to throw
   * an exception named PageUnpinnedException in case of error.)
```

```
   *
   * @param pageId: page number in the minibase.
   * @param dirty the dirty bit of the frame
   */

  public void unpinPage(PageId pageId, boolean dirty) {};
/**-------------------------------------------------------------*/

  /**
   * Allocate new pages.
   * Call DB object to allocate a set of new pages and
   * find a frame in the buffer pool for the first page
   * and pin it. (This call allows a client of the Buffer Manager
   * to allocate pages on disk.) If buffer is full, i.e., you
   * can't find a frame for the first page, ask DB to deallocate
   * all these pages, and return null.
   *
   * @param firstpage the address of the first page.
   * @param howmany total number of allocated new pages.
   *
   * @return the first page id of the new pages.  null, if error.
   */

  public PageId newPage(Page firstpage, int howmany) {};
/**-------------------------------------------------------------*/

  /**
   * This method should be called to delete a page that is on disk.
   * This routine must call the method in diskmgr package to
   * deallocate the page.
   *
   * @param globalPageId the page number in the data base.
   */

  public void freePage(PageId globalPageId) {};
/**-------------------------------------------------------------*/
```

```
  /**
   * Used to flush a particular page of the buffer pool to disk.
   * This method calls the write_page method of the diskmgr package.
   *
   * @param pageid the page number in the database.
   */

  public void flushPage(PageId pageid) {};
/**-------------------------------------------------------------*/
 };
```

3 Internal Design

The *buffer pool* is a collection of *frames* (page-sized sequence of main memory bytes) that is managed by the Buffer Manager. Conceptually, it should be stored as an array bufPool[numbuf] of Page objects. However, due to the limitation of Java language, it is not feasible to declare an array of Page objects and later on writing string (or other primitive values) to the defined Page. To get around the problem, we have defined our Page as an array of bytes and deal with buffer pool at the byte array level. Therefore, when implementing your constructor for the BufMgr class, you should declare the buffer pool as *bufpool[numbuf][page_size]*. Note that the size of minibase pages is defined in the interface *GlobalConst* of the *global* package. Before coding, please make sure that you understand how Page object is defined and manipulated in Java minibase.

In addition, you should maintain an array bufDescr[numbuf] of *descriptors*, one per frame. Each descriptor is a record with the following fields:

page number, pin_count, dirtybit.

The *pin_count* field is an integer, *page number* is a PageId object, and *dirtybit* is a boolean. This describes the page that is stored in the corresponding frame. A page is identified by a *page number* that is generated by the DB class when the page is allocated, and is unique over all pages in the database. The PageId type is defined as an integer type in minirel.h.

A simple *hash table* should be used to figure out what frame a given disk page occupies. The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of $\langle$ *page number, frame number* $\rangle$ pairs. The array is called the *directory* and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket

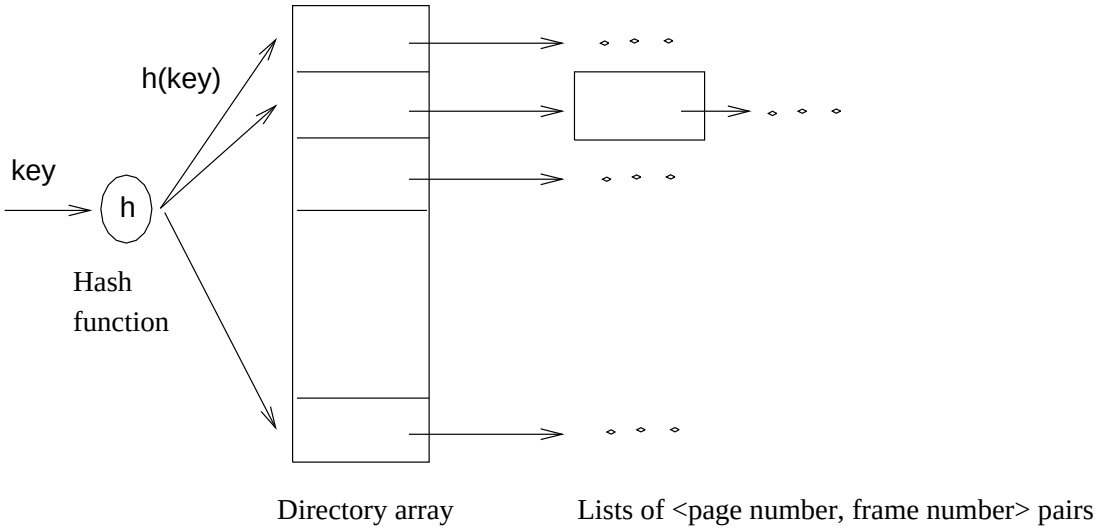

Figure 1: Hash Table

that contains the frame number for this page, if the page is in the buffer pool. If you search the bucket and don't find a pair containing this page number, the page is not in the pool. If you find such a pair, it will tell you the frame in which the page resides. This is illustrated in Figure 1.

The hash function must distribute values in the domain of the search field uniformly over the collection of buckets. If we have HTSIZE buckets, numbered 0 through M-1, a hash function h of the form $h(value) = (a * value + b)\ mod\ HTSIZE$ works well in practice. HTSIZE should be chosen to be a prime number.

When a page is requested the buffer manager should do the following:

1. Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is not in the pool, it should be brought in as follows:

   (a) Choose a frame for replacement, using the Clock replacement policy.

   (b) If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that it contains to disk, using the appropriate DB class method).

   (c) Read the requested page (again, by calling the DB class) into the frame chosen for replacement; the *pin_count* and *dirtybit* for the frame should be initialized to 0 and FALSE, respectively.

   (d) Delete the entry for the old page from the Buffer Manager's hash table and insert an entry for the new page. Also, update the entry for this frame in the bufDescr array to reflect these changes.

2. *Pin* the requested page by incrementing the *pin_count* in the descriptor for this frame. and return a pointer to the page to the requestor.

To implement the Clock replacement policy, maintain a queue of frame numbers. When a frame is to be chosen for replacement, you should pick the first frame in this list. A frame number is added to the end of the queue when the *pin_count* for the frame is decremented to 0. A frame number is removed from the queue if the *pin_count* becomes non-zero: this could happen if there is a request for the page currently in the frame!

4 Where to Find Code.

Please copy all the files into your own local directory. The contents are:

- under *bufmgr* directory
  You should have all your code for *bufmgr* package implemented under this directory.
- under *diskmgr* direcotry
  You should study the code for *diskmgr* package carefully before you implement the *bufmgr* package. The detailed description of the files are not included. Please refer to the java documentation of the packages.
- under *tests* directory
  TestDriver.java, *BMTest.java*: Buffer manager test driver program.

You can find other useful information by reading the java documentation on other packages, such as the *global* package and the *chainexception* package. Make sure you import the bufmgrAssign.lib to work with your project.

5 Error Protocol

Though the Throwable class in Java contains a snapshot of the execution stack of its thread at the time it was created and also a message string that gives more information about the error (exception), we have decided to maintain a copy of our own stack to have more control over the error handling.

We provide the *chainexception* package to handle the Minibase exception stack. Every exception created in your *bufmgr* package should extend the ChainException class. The exceptions are thrown according to the following rule:

- Error caused by an exception caused by another layer:

```
For example: (when try to pin page from diskmgr layer)

  try {
    SystemDefs.JavabaseBM.pinPage(pageId, page, true);
  }
  catch (Exception e) {
    throw new DiskMgrException(e, "DB.java: pinPage() failed");
  }
```

- Error not caused by exceptions of others, but needs to be acknowledged.

```
For example: (when try to unpin a page that is not pinned)

      if (pin_count == 0) {
        throw new PageUnpinnedException (null, "BUFMGR: PAGE_NOT_PINNED.");
      }
```

Basically, the ChainException class keeps track of all the exceptions thrown accross the different layers. Any exceptions that you decide to throw in your *bufmgr* package should extend the ChainException class.

For testing purposes, we ask you to throw the following exceptions in case of error (use the exact same name, please):

- BufferPoolExceededException
  Throw a BufferPoolExceededException when try to pin a page to the buffer pool with no unpinned frame left.
- PagePinnedException
  Throw a PagePinnedException when try to free a page that is still pinned.
- HashEntryNotFoundException
  Throw a HashEntryNotFoundException when page specified by PageId is not found in the buffer pool.

Feel free to throw other new exceptions as you see fit. But make sure that you follow the error protocol when you catch an exception. Also, think carefully about what else you need to do in the *catch* phase. Sometimes you do need to unroll the previous operations when failure happens.

6 What to Turn In, and When

You should turn in copies of your code together with copies of the output produced by running the tests. Also, you should email the code packages to Jason before the due the day. The assignment is due on March 24th before class. The solution will be made public after that time, and solutions turned in after that time will not receive any credit. So be sure to turn in whatever you have, even if it is not working fully, at that time.