Database Systems

Instructor: Hao-Hua Chu, Winston Hsu

Fall Semester, 2010

Assignment 4: Buffer Manager

Deadline: 23:59, Dec. 8 (Wednesday), 2010
This is a group assignment, and at most 2 students per group are allowed.

Cheating Policy: If you are caught cheating, your grade is 0.
Late Policy: We will not accept any assignment submissions after Wednesday.

Introduction

In this assignment, you will have to implement a simple buffer management layer (without support for concurrency control and recovery) for the Minibase database system. The code for the underlying Disk Space Manager will be given. HTML documentation for Minibase is available on web (http://www.cs.wisc.edu/coral/minibase/project.html). In particular, you should read the description of the DB class, which you will use extensively in this assignment.

Note: The Minibase buffer manager layer differs from what you have to implement in that it contains methods to support concurrency control and recovery.

Design Overview and Implementation Details

The *buffer pool* is collection of *frames* (page-sized sequence of main memory bytes) that is managed by the Buffer Manager. It should be stored as an array **bufPool[numbuf]** of Page objects. 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 <*page number*, *frame number*> 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 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 is in,

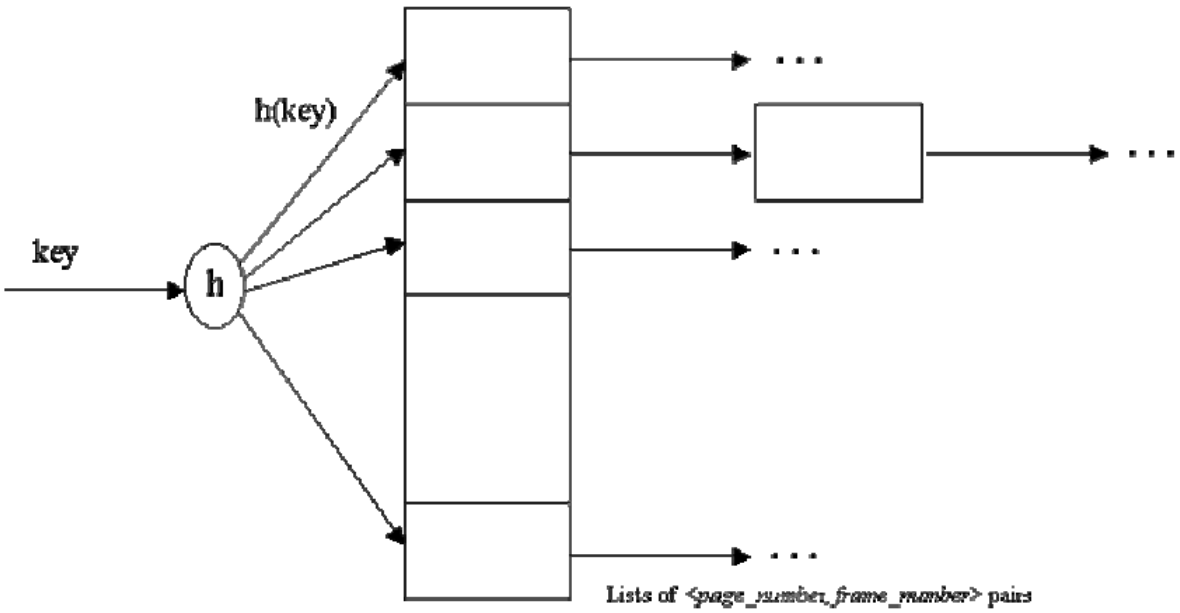

Figure 1: Hash Table

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

$$h(value) = (a * value + b) \bmod HTSIZE$$

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: Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is in the pool, you have to make sure that the page pinned as READ_WRITE_MODE cannot be pinned again (write exclusive condition). On the other hand, the page has been pinned before cannot be

pinned again as READ_WRITE_MODE. In sum, the page can only be pinned multiple times in READ_MODE.

If the page is not in the pool, it should be brought in as follows:

- Choose a frame for replacement, using the LOVE/HATE replacement policy.
- If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that is contains to disk, using the appropriate DB class method).
- 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.
- 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.
- *Pin* the requested page by incrementing the *pin_count* in the descriptor for this frame and return a pointer to the page to the requester.

The Love/Hate Replacement Policy

Theoretically, the best candidate page for replacement is the page that will be last requested in the future. Since implementing such policy requires a future predicting oracle, all buffer replacement policies try to approximate it one way or another. The LRU policy, for example, uses the past access pattern as an indication for the future. However, sequential flooding can ruin this scheme and MRU becomes more appropriate in this particular situation. In this assignment you are supposed to implement the *love/hate* replacement policy. The policy tries to enhance prediction of the future by relying on a hint from the upper levels about the page. The upper level user hints the buffer manager that the page is *loved* if it is more likely that the page will be needed in the future, and *hated* if it is more likely that the page will not be needed. The policy is supposed to maintain an MRU list for the hated pages and an LRU list for the loved pages. If a page is needed for replacement, the buffer manager selects from the list of hated pages first and then from the loved pages if no hated ones exist.
A situation may arise when a page is both loved and hated at the same time. This can happen if the page was pinned by two different users and then was unpinned by the first

one as a hated page and by the other as a loved page. In this case, assume that “love conquers hate”, meaning that once a page is indicated as loved it should remain loved.

The Buffer Manager Interface

The simplified buffer manager interface that you will implement allows a higher level program to allocate and deallocate pages on disk, to bring a disk page to 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:

```
enum MODE {
    READ_MODE, READ_WRITE_MODE;
};
class BufMgr {
    public:
    // This is made public just because we need it in your driver_test.C.
    // It could be private for real use.
    page* bufPool;
    // The physical buffer pool of pages.

    public:
    BufMgr(int numbuf, Replacer *replacer = 0);
    // Allocate “numbuf” pages (frames) for the pool in main memory.

    ~BufMgr();
    // Should flush all dirty pages in the pool to disk before shutting down
    // and deallocate the buffer pool in main memory.

    Status pinPage(PageId PageId_in_a_DB, Page*& page, MODE mode);
    // Check if this page is in buffer pool. If it is, increment the pin_count
    // and let page be a pointer to this page. If the pin_count was 0 before the
    // call, the page was a replacement candidate, but is no loner 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 DB
    // class method) and pin it.
    // Also, must write out the old page in chosen frame if it is dirty before
    // reading new page.
    // must specify read/write mode, if the pinned page has been pinned before, make sure that write
```

```
    // exclusive condition is hold, else return error

    Status unpinPage(PageId globalPageId_in_a_DB, int diry, int hate);
    // hate should be TRUE if type page is "hated" and FALSE otherwise.
    // 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
    // should decrement it.
    // If pin_count = 0 before this call, return error.

    Status newPage(PageId& firstPageId, Page*& firstpage, int
    howmany=1);
    // Find a frame in the buffer pool for the first page.
    // If a frame exists, call DB object to allocate a run of new pages and pin it in read mode.
    // (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, return error.

    Status freePage(PageId globalPageId);
    // This method should be called to delete a page that is on disk.
    // This routine must call the DB class to deallocate the page.

    Status flushPage(int pageId);
    // Used to flush a particular page of the buffer pool to disk
    // Should call the write_page method of the DB class.

    Status flushAllPages();
    // Flush all pages of the buffer pool to disk.

};
```

Compiling Your Code and Running the Tests

Copy all the files from **web site** into your working directory. The files are:

- *Makefile*: A sample makefile to compile your project. You will have to set up any dependencies by editing this file. You may also use your own makefile.

- *buf.h*: This file contains the specifications for the class BufMgr and you have to implement these specification as part of this assignment.

- *main.C, test_driver.C, BMTester.C*: Buffer manager test driver program.
- *sample_output*: Sample output of a correct implementation.
- *ErrProc.sample*: This file shows you how to use the error protocol in Minibase.

If you *make* the project, it will create an executable named *buftest*. Right now, it does not work, even can't be compiled correctly. You will need to modified buf.h to add your own structures of hash table, LRU and MRU lists. Furthermore, you should implement all these method bodies in file buf.C.

How to hand-in

Email the files "buf.h", "buf.C" and a one-page document to siriushpa@gmail.com before the deadline with the title "[DB10] hw4_ID_ID" (Ex. "[DB10] hw4_b96902xxx" or "[DB10] hw4_b96902xxx_b96902xxx").