UNIVERSITY OF WISCONSIN–MADISON
COMPUTER SCIENCES DEPARTMENT

CS 564: DATABASE MANAGEMENT SYSTEMS

Assignment 1: Buffer Manager
Due: September 27, 1996

Instructors: Jeff Naughton and Raghu Ramakrishnan

1 Introduction

In this 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 will carry out this assignment, and subsequent ones, in teams of two. Please choose a partner as soon as possible, or send mail to xbao or cychan so that we can find a partner for you.

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 Netscape. There is a link to the Minibase home page in the CS564 home page. In particular, you should read the description of the DB class, which you will call extensively in this assignment. The header file for the DB class is in *db.h*, and there is a link to this file.

Important Note: The Minibase buffer manager interface differs from what you have to implement (described below) in that it contains methods that are required to support concurrency control and recovery.

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:

```
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);
// Allocate 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,int emptyPage=0);

// Check if this page is in 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;
// be sure to remove this from the LRU list of candidates.
// 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.  (You can assume that emptyPage == 0 for
// this assignment.)

     Status  unpinPage(PageId globalPageId_in_a_DB, int dirty=FALSE);

// 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, and if it
// becomes zero, should update the LRU list by adding an entry for this frame.
// If pin_count=0 before this call, return error.

       Status  newPage(PageId& firstPageId, Page*& firstpage,int howmany=1);

// Call DB object to allocate a run 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 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

};
```

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. 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 resides. This is illustrated in Figure ??.

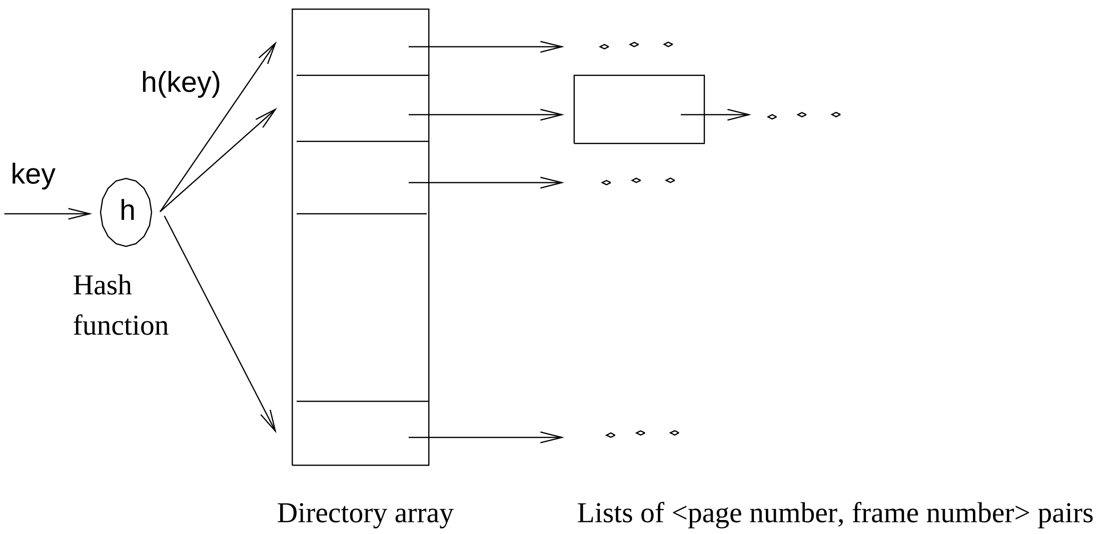

Figure 1: Hash Table

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 $h(value) = (a * value + b) \ mod \ HTSIZE$ 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 LRU 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 LRU 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 Makefiles, Code, etc.

Please copy all the files from ~cs564-1/fall96/proj1/src into your own local 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 can also design your own Makefile.
- *buf.h*: Specifications for the class BufMgr. you have to implement these specifications as part of the assignment. You must also adhere to the error protocol described in:

  http://www.cs.wisc.edu/coral/minibase/system/error.html

- *main.C*, *test_driver.C*, *BMTester.C*: Buffer manager test driver program.
- *ErrProc.sample*: a sample program to show you how to use the error protocol

You can find other useful include files (*db.h*, *page.h*, *test_driver.h*, *BMTester.h*, *minirel.h* and *new_error.h*) in ~cs564-1/fall96/proj1/include.

5 Error Protocol

Be sure to follow the error protocol described in:

http://www.cs.wisc.edu/coral/minibase/system/error.html

An example file illustrating the use of the error protocol is available in

~cs564-1/fall96/proj1/src/ErrProc.sample

It covers much of what you need to know about the protocol.

You can look at *new_error.h* for more details. It is in

~cs564-1/fall96/proj1/include/new_error.h

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 provided by the TAs. The assignment is due at 5PM on September 27th. 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.

I emphasize that late submissions will not receive any credit. Computers—and life!—being what they are, expect everything to take longer than you expect, even taking this expectation into account. So start early, and plan on getting things done well before the due date. Nothing short of a nuclear explosion (in the CS building, not the South Pacific) constitutes a valid reason for an extension.