EECS 484, FALL 2011
Minirel 2K, Part 1: Buffer Manager

Due: November 10th at 11:59 PM

Introduction

The first part of the Minirel2K project involves implementing a buffer manager. Recall that a database **buffer pool** is a collection of memory buffers (also called frames) that are used to hold database pages that have been read from disk into memory.

Since the database itself may be many times larger than the size of memory, only a subset of the database pages can fit in memory at any given time. The **buffer manager** is used to control which pages are in memory at any particular time. Whenever a request is made (by higher layers of the DBMS) for a data page, the buffer manager must check to see whether the page is already in the buffer pool. If so, the buffer manager simply returns a pointer to the page. If not, the buffer manager frees a frame, and then reads the requested page into the free frame.

Buffer Replacement Policy – "Love Hints"

There are many ways of deciding which page to replace when a free frame is needed. Common policies include LRU, MRU, and FIFO. However, as we saw in class, blindly applying one of these policies may lead to undesirable behavior (e.g., sequential flooding for LRU) since the buffer manager is unaware of the activities of higher layers of the DBMS.

One way to address this problem is for the upper layers of the DBMS to send *hints* to the buffer manager about what they are doing. For the purposes of this project, we will consider a very simple kind of hint called a *love hint*. When a caller unpins a page, the caller can send a love hint to the buffer manager, which indicates that it is likely to request the page again soon. In this case, the buffer manager should try, if possible, to keep the page in the buffer pool.

For this project, you will implement a page replacement policy that combines love hints with the *Clock* page replacement algorithm. (Clock mimics LRU, but has much less overhead.) Conceptually, the buffer frames are arranged in a circular list. A frame may be empty, or it can be used to hold a page. Each frame has the following associated bookkeeping information:

Valid bit – Set to true if the frame contains a database page

Loved bit – Set to true if someone has recently "loved" the page

Pincount – Number of callers currently using the page contained in the frame

Dirty bit – Set to true if the page in the frame has been modified since it was fetched from disk

Figure 1 provides an overview of the page replacement algorithm. At any point in time, the clock "hand" (an integer between 0 and NUMBUFS – 1) points to some frame in the buffer pool. Whenever a caller requests a page (via a call to bufMgr::allocBuf), the clock hand is advanced (using modular arithmetic so that it does not go past NUMBUFS – 1). Each time the clock passes a frame, the *loved* bit is examined and cleared. If the *loved* bit is true, this means that the page was "recently" loved, so it should not be replaced. However, if the loved bit is false, the

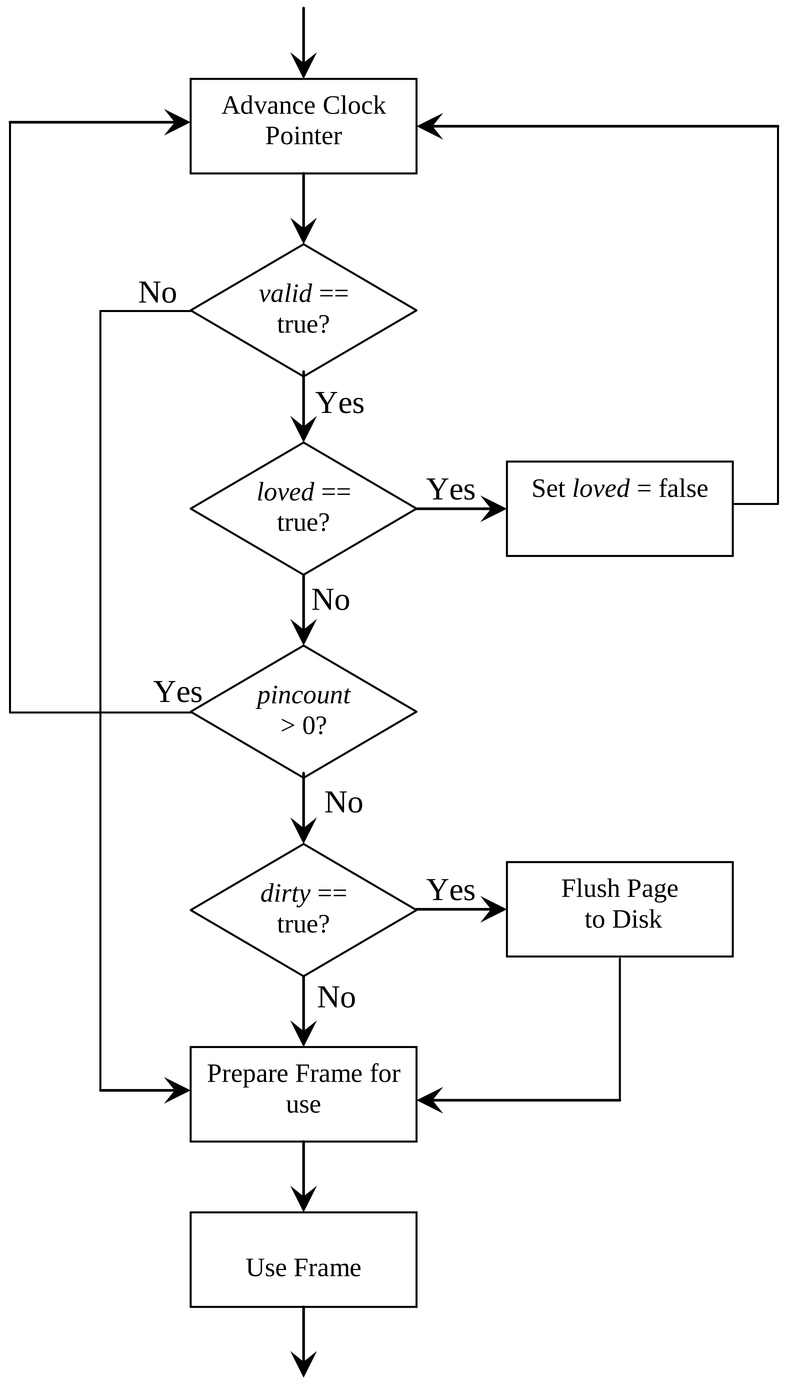

Figure 1: Clock with Love Hints Replacement Algorithm

page is considered for replacement, assuming it is not pinned.

If the page selected for replacement is dirty (*dirty* == true), then the page occupying the frame is written to disk. Otherwise, the frame is just cleared using the Clear() function, and the new page is read from disk to that location.

Class Structure

The Minirel buffer manager consists of four classes: `BufMap`, `BufDesc`, `BufMgr`, and `BufStats`. These classes, as well as your assignment, are described in detail below.

The BufMap Class (in bufMap.h, bufMap.cpp)

A buffer pool typically contains several thousand frames, which are accessed frequently by the DBMS. If an upper layer requests a particular page, a naïve approach to locating that page would be to examine every page in the buffer pool. Instead, the `BufMap` class should allow you to locate the page more efficiently. We provide the public class definition for you, but the implementation decisions are up to you. Think carefully about the functionality that the `BufMap` must provide, and choose data structures that will efficiently support this functionality.

The following is the public definition of `BufMap` (from bufMap.h):

```
// Class to keep track of pages in the buffer pool
class BufMap {
public:
  // Insert an entry mapping (file,pageNo) to frameNo.
  // Return BUFMAPERROR if the entry already exists, OK otherwise.
  const Status insert(File* file, const int pageNo, const unsigned int frameNo);

  // Find the frameNo corresponding to (file,pageNo), and return by reference.
  // Return BUFMAPNOTFOUND if the entry is not found, OK otherwise.
  const Status lookup(File* file, const int pageNo, unsigned int & frameNo) const;

  // Remove the entry corresponding to (file,pageNo).
  // Return BUFMAPNOTFOUND if entry is not found, OK otherwise.
  const Status remove(File* file, const int pageNo);
};
```

Your task is to implement each of the public methods (in bufMap.cpp). You may also add other methods and variables to this class (in bufMap.h and bufMap.cpp) as you see fit, but you may not modify the public API. *We suggest that you start this project by implementing and testing the `BufMap` class before you do anything else.*

The BufDesc Class (in buf.h)

The BufDesc class is provided to you, and is used to keep track of the state of each frame in the buffer pool. The following is the class definition:

```
// BufDesc class maintains bookkeeping information about individual buffer pool frames
class BufDesc {
    friend class BufMgr;
private:
  BufDesc();
  File* file;   // pointer to file object
  int   pageNo; // page within file
  int   pinCnt; // number of times this page has been pinned
  bool dirty;   // true if dirty;  false otherwise
  bool valid;   // true if page is valid
  bool loved; // true if someone has recently set a love "hint" for this page

  // Initialize buffer frame for a new requester
  void Clear();

  // Setup buffer frame for a particular page
  void Set(File* filePtr, int pageNum);
};
```

The purpose of most of the member variables and methods of the `BufDesc` class should be pretty obvious. The `dirty` bit, if true, indicates that the page is dirty (i.e. has been updated) and thus must be written to disk before the frame is used to hold another page. The `pinCnt` indicates how many times the page has been pinned. The `loved` flag is used by the page replacement algorithm. The `valid` bit is used to indicate whether the frame contains a valid page. It is not necessary to implement any additional methods in this class.

<u>The BufMgr Class</u> (in buf.h,buf.cpp)

The BufMgr class is the heart of the buffer manager. The bulk of this project is implementing the methods defined in this class.

The following is the class definition (found in buf.h):

```
class BufMgr {
private:
  unsigned int clockHand; // Clock hand for page replacement algorithm
  BufMap bufMap; // Data structure allows us to quickly locate a page
  BufDesc *bufTable; // Vector of information describing each buffer frame
  unsigned int numBufs;  // Number of pages in the buffer pool
  BufStats bufStats; // Statistics about buffer pool usage

  // Allocate a free frame; return the number of the frame by reference
  const Status allocBuf(unsigned int & frame);

  // Advance the clock hand
  void advanceClock();

public:
  // Buffer pool comprised of an array of in-memory Page objects
  Page *bufPool;

  BufMgr(const unsigned int bufs);
  ~BufMgr();

  // Read page identified by (file, pageNo) pair
  // Pass back the in-memory representation of the page by reference
  const Status readPage(File* file, const int PageNo, Page*& page);

  // Unpin a page
  // Call to unpin must include two pieces of information:
  // (1) Is the page dirty? and
  // (2) Is the page "loved" (likely to be needed again soon)?
  const Status unPinPage(File* file, const int PageNo, const bool dirty, const bool love);

  // Allocate a new empty page
  const Status allocPage(File* file, int& PageNo, Page*& page);

  // Write all of the dirty pages in the file back to disk
  const Status flushFile(File* file);

  // Dispose of the specified page
  const Status disposePage(File* file, const int PageNo);

  void printSelf(void); // print the contents of the buffer pool
  int numUnpinnedPages(); // get the number of unpinned pages in the buffer pool

  const BufStats & getBufStats();
  void clearBufStats();
};
```

You will need to implement the following methods. (Put your code in buf.cpp)

BufMgr(const unsigned int bufs)

Initializes the buffer pool. In the constructor, you should do two things: First, initialize the bufPool, which should contain buf pages. Second, initialize bufTable to contain bufs BufDesc objects. Notice that bufPool and bufTable are "parallel" arrays. For example bufPool[5] is a Page; the corresponding bookkeeping information is located in bufTable[5]. Initially, each frame should be clear. The BufMap should also start out empty.

~BufMgr()
Flushes out all dirty pages, regardless of whether they are pinned or unpinned, and deallocates bufPool and bufTable.

const Status allocBuf(unsigned int & frame)
Allocates a free frame using the page replacement algorithm described earlier. If necessary, write a dirty page back to disk. Returns BUFFEREXCEEDED if all buffer frames are pinned, UNIXERR if a Unix file system error occurs when a dirty page was being written to disk, or OK if the function completes without any errors.

const Status readPage(File* file, const int PageNo, Page*& page)
If the page is already in the buffer pool, increment the pinCnt and return a pointer to the page (by reference, via the page parameter). Otherwise, call allocBuf() to allocate a buffer frame. Then call file->readPage() to read the page into the buffer pool frame from the disk. Finally, invoke Set() on the BufDesc object to set up the frame properly. Return OK if no errors occurred, UNIXERR if a Unix error occurred, BUFFEREXCEEDED if all buffer frames are pinned, or BUFMAPERROR if a BufMap error occurs.

const Status unPinPage(File* file, const int PageNo, const bool dirty, const bool love)
Decrements the pinCnt of the frame containing (file, PageNo). If dirty == true, set the dirty bit for the frame. If love == true, set the loved bit for the frame. Return OK if no errors occurred, BUFMAPNOTFOUND if the page is not in the buffer pool mapping, PAGENOTPINNED if the pin count is already 0, or BADBUFFER if the (file, page) pair points to an invalid buffer frame entry (this will only happen if the buffer pool has been corrupted).

const Status allocPage(File* file, int& PageNo, Page*& page)
This method allocates a new page in the specified file by calling the File->allocatePage() method. Before allocating a new page, this method should call allocBuf() to set up a frame to hold this page. This method returns the page number of the page by reference via the pageNo parameter and a pointer to the buffer frame allocated for the page via the page parameter. Return OK if no errors occurred, UNIXERR if a Unix error occurred, BUFFEREXCEEDED if all buffer frames are pinned, or BUFMAPERROR if a BufMap error occurred.

const Status disposePage(File* file, const int PageNo)
Deallocates the page denoted by PageNo from the file. The page may or may not currently be in the buffer pool. If it is, then the frame containing the page must be cleared. Return OK if no errors occurred, UNIXERR if a Unix error occurred, PAGEPINNED if the page is currently

pinned in the buffer pool, or BADBUFFER if the (file, page) pair points to an invalid buffer frame entry (which indicates that the buffer pool has been corrupted).

In the I/O layer file implementation, the first page of a file (pageNo == 0) is used in a special way. Consequently, trying to dispose pageNo 0 should return the error BADPAGENO.

const Status flushFile(File* file)
This method will be called by DB::closeFile() when all instances of a file have been closed (in which case all pages of the file should have been unpinned). flushFile() should scan bufTable, and for each dirty page it should call file->writePage() to flush the dirty page to disk. If the page is not dirty, do nothing. Returns OK if no errors occurred, or PAGEPINNED if some page of the file is pinned.

Note on Propagating Error Codes

If your implementation of a method (caller) calls another Minirel2K method (callee), and if the callee returns an error code, then the caller must return back that same error code.

The BufStats Class (in buf.h,buf.cpp)

To monitor buffer pool usage, it is useful to have a data structure that counts various buffer manager events. For this purpose, we have defined a structure called BufStats in buf.h. BufStats has the following three counters:

1. `accesses`: To keep track of the total number of accesses to the buffer pool
2. `diskreads`: To keep track of the number of pages read from the disk, and
3. `diskwrites`: To keep track of the number of pages written to the disk.

As part of your buffer manager implementation, you will have to increment these counters inside appropriate calls to the buffer manager. Increment *accesses* whenever a page in the buffer pool is accessed via either the readPage() or the allocPage() calls (i.e. count the total number of times these two methods are called regardless of whether they succeed or fail).

Use *diskwrites* and *diskreads* to keep track of the number of disk writes and reads that are incurred by the buffer pool methods. Increment these counters only when the corresponding calls to the File class are successful. *diskreads* counts the number of times "File->readPage()" is called successfully, and *diskwrites* keeps track of the total number of successful calls to the "File->writePage()", "File->disposePage()", and "File->allocatePage()" methods.

Getting Started

Start by downloading Project 3 directory from CTools. The following is a summary of the files:

Makefile	Makefile for this project.
buf.h	Class definition for the buffer manager

buf.cpp	Skeleton implementations of the buffer manager methods. **You provide the actual implementation.**
bufMap.h, bufMap.cpp	Skeleton implementations of the BufMap class which keeps track of pages in the buffer pool. **You provide the actual implementations for these methods and any others you choose to add to the BufMap class.**
db.h, db.cpp	DB and File classes. You should not need to change these.
error.h, error.cpp	Error codes and error class. (You may add extra error codes if you like.)
testbuf.cpp	A sample test driver. Augment this file with your additional tests.

We suggest that you start this project by implementing and testing the bufMap class before moving on to the classes in buf.cpp.

Coding and Testing

Please use good coding style, including comments, when developing your solution. In your submitted solution, each file should include the names and uniquenames of all team members.

In addition to the test driver we provide, we will also test your code against our (more comprehensive) “hidden” test driver. What this means is that you should take the time to test your code carefully; don’t just assume that a clean run of testbuf means your code is perfect!

To encourage thorough testing, you must submit additional tests with your project. (Add them to testbuf.cpp.) You should include a short report (less than 3 pages) describing your tests and their utility. This report should also describe what data structures you chose for implementing BufMap and justify your design choice.

Submission instructions (using CTools)

You must submit your assignment using CTools. Please follow the following instructions:

1. Log on to a campus LINUX machine.
2. Create a directory called P3. Place *only* the following files in this directory: Makefile, buf.h, buf.cpp, bufMap.h, bufMap.cpp. db.h, db.cpp, error.h, error.cpp, testbuf.cpp. Also include a file called report.pdf, which described your design and testing methodology. Note that you must include a working Makefile, even if you are using an IDE. (We should be able to type “make” to build the executable.)
3. Then from the directory directly above the directory “P3”, type:

   ```
   > tar czvf P3.tgz P3/
   ```

   You will see message that shows a tar file being created with your solution files. At the end of running this command, upload P3.tgz into CTools using the "Upload local file". Do not use the URL option!
4. Please submit your project to one partner’s account. (Please don’t upload the assignment multiple times, across different CTools accounts.)

We will compile your code on a CAEN Linux machine using the compiler at /usr/um/bin/gcc, link it with our test driver and test it. Since we are supposed to be able to test your code with any valid driver (not only buftest.cpp) IT IS VERY IMPORTANT TO BE FAITHFUL TO THE EXACT DEFINITIONS OF THE PUBLIC API as specified above.

Grading

80% of the grade for this assignment is for correctness, 5% of the grade is based on your programming style, and 15% is for your testing / report.