CS 564
Assignment 3 - The Buffer Manager

Due: 11:30PM on Thursday, 6th November 2014
Teams of at most size 2 allowed

Introduction

Part 3 of the project involves implementing a buffer manager for your database system, henceforth called "Minirel". A database **buffer pool** is an array of fixed-sized memory buffers called **frames** that are used to hold database pages (also called disk blocks) that have been read from disk into memory. A page is the unit of transfer between the disk and the buffer pool residing in main memory. Most modern database systems use a page size of at least 8,192 bytes. Another important thing to note is that a database page in memory is an exact copy of the corresponding page on disk when it is first read in. Once a page has been read from disk to the buffer pool, the DBMS software can update information stored on the page, causing the copy in the buffer pool to be different from the copy on disk. Such pages are termed "**dirty**".

Since the database on disk itself is usually about 100 times larger than amount of main memory available on a machine, 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 memory resident. Whenever the buffer manager receives a request for a data page, the buffer manager must check to see whether the requested page is already in the one of the frames that constitute the buffer pool. If so, the buffer manager simply returns a pointer to the page. If not, the buffer manager frees a frame (possibly by writing to disk the page it contains if the page is dirty) and then reads in the requested page from disk into the frame that has been freed.

Before reading further you should first read the documentation that describes the I/O layer of Minirel so that you understand its capabilities. In a nutshell the I/O layer provides an object-oriented interface to the Unix file with methods to open and close files and to read/write pages of a file. For now, the key thing you need to know is that opening a file (by passing in a character string name) returns a pointer to an object of type File. This class has methods to read and write pages of the File. You will use these methods to move pages between the disk and the buffer pool.

Buffer Replacement Policies and the Clock Algorithm

There are many ways of deciding which page to replace when a free frame is needed. Commonly used policies in operating systems are FIFO, MRU and LRU. Even though LRU is one of the most commonly used policies it has high overhead and is not the best strategy to use in a number of common cases that occur in database systems. Instead, many systems use the *clock* algorithm that approximates LRU behavior and is much faster.

Figure 1 illustrates the execution of the clock algorithm. Each square box corresponds to a frame in the buffer pool. Assume that the buffer pool contains NUMBUFS frames, numbered 0 to NUMBUFS-1. Conceptually, all the frames in the buffer pool are arranged in a circular list. Associated with each frame is a bit termed the *refbit*. Each time a page in the buffer pool is accessed (via a readPage() call to the buffer manager) the refbit of the corresponding frame is set to *true*. At any point in time the clock hand (an integer whose value is between 0 and NUMBUFS - 1) is advanced (using modular arithmetic so that it does not go past NUMBUFS - 1) in a clockwise fashion. For each frame that the clockhand goes past, the refbit is examined and then cleared. If the bit had been set, the corresponding frame has been referenced "recently" and is not replaced. On the other hand, if the refbit is *false*, the page is selected for replacement (assuming it is not pinned - pinned pages are discussed below). If the selected buffer frame is *dirty* (ie. it has been modified), the page currently occupying the frame is written back to disk. Otherwise the frame is just *cleared* (using the clear() function) and a new page from disk is read in to that location. The details of the algorithm is given below.

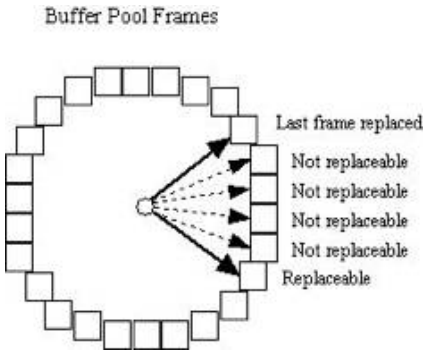

Figure 1. Buffer Pool Frames

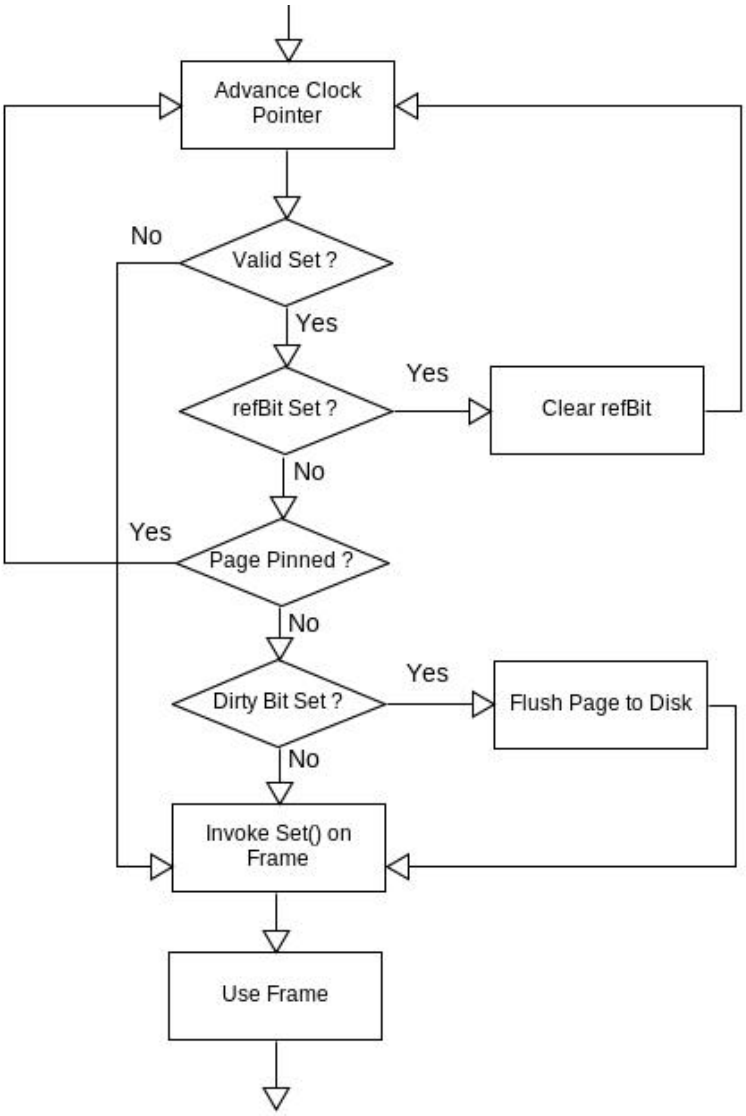

Figure 2. CLOCK Replacement Algorithm

The Structure of the Buffer Manager

The Minirel buffer manager uses three C++ classes: BufMgr, BufDesc, and BufHashTbl. There will be **one** instance of the BufMgr class. A key component of this class will be the actual buffer pool which consists of an array of NUMBUFS frames, each the size of a database page. In addition to this array, the BufMgr instance also contains an array of NUMBUFS instances of the BufDesc class which are used to describe the state of each frame in the buffer pool. A hash table is used to keep track of which pages are currently resident in the buffer pool. This hash table is implemented by an instance of the BufHashTbl class. This instance will be a private data member of the BufMgr class. These classes are described in detail below.

The BufHashTbl Class

The BufHashTbl class is used to map file and page numbers to buffer pool frames and is implemented using chained bucket hashing. We have provided an implementation of this class for your use.

The structure of each hash table entry is (here it is ok to use a struct instead of a class):

```
1.  struct hashBucket {
2.      File* file;       // pointer to a file object (more on this below)
3.      int pageNo;       // page number within a file
4.      int frameNo;      // frame number of page in the buffer pool
5.      hashBucket* next; // next bucket in the chain
6.  };
```

Here is the definition for the hash table.

```
1.  class BufHashTbl
2.  {
3.          private:
4.                  hashBucket**  ht;                           // pointer to actual hash table
5.                  int HTSIZE;
6.                  int hash(const File* file, const int pageNo); // returns a value between 0 and HTSIZE-1
7.
8.          public:
9.                  BufHashTbl(int htSize);                     // constructor
10.                 ~BufHashTbl();                              // destructor
11.
12.                 // insert entry into hash table mapping (file,pageNo) to frameNo;
13.                 // returns OK or HASHTBLERROR if an error occurred
14.                 Status insert(const File* file, const int pageNo, const int frameNo);
15.
16.                 // Check if (file,pageNo) is currently in the buffer pool (ie. in
17.                 // the hash table.  If so, return the corresponding frameNo via the frameNo
18.                 // parameter.  Else, return HASHNOTFOUND
19.                 Status lookup(const File* file, const int pageNo, int& frameNo);
20.
21.                 // remove entry (file,pageNo) from hash table. Return OK if
22.                 // page was found.  Else return HASHTBLERROR
23.                 Status remove(const File* file, const int pageNo);
24. };
```

Here are the definitions of each individual function:

```
Status insert(File* file, const int pageNo, const int frameNo)
```

Inserts a new entry into the hash table mapping file and pageNo to frameNo. Returns HASHTBLERROR if the entry already exists and OK otherwise.

```
Status lookup(File* file, const int pageNo, int& frameNo)
```

Finds the frameNo that is obtained by hashing file and pageNo. Returns HASHNOTFOUND if entry is not found and OK otherwise.

```
Status remove(File* file, const int pageNo)
```

Removes the entry obtained by hashing file and pageNo. Returns HASHTBLERROR if entry is not found, OK otherwise.

The BufDesc Class

The BufDesc class is used to keep track of the state of each frame in the buffer pool.

```
class BufDesc {
        friend class BufMgr;
        private:
                File* file;     // pointer to file object
                int pageNo;     // page within file
                int frameNo;    // buffer pool frame number
                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 refbit;    // has this buffer frame been referenced recently?

                void Clear() {  // initialize buffer frame for a new user
                        pinCnt = 0;
                        file = NULL;
                        pageNo = -1;
                        dirty = refbit = false;
                        valid = false;
                };

                void Set(File* filePtr, int pageNum) {
                        file = filePtr;
                        pageNo = pageNum;
                        pinCnt = 1;
                        dirty = false;
                        refbit = true;
                        valid = true;
                }

                BufDesc() {
                        Clear();
                }
};
```

First notice that all attributes of the BufDesc class are private and that the BufMgr class is defined to be a friend. While this may seem strange, this approach restricts access to only the BufMgr class. The alternative (making everything public) open up access too far.

The purpose of most of the attributes 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 `refbit` is used by the clock algorithm. The `valid` bit is used to indicate whether the frame contains a valid page. You do not HAVE to implement any methods in this class. However you are free to augment it in any way if you wish to do so.

The BufMgr Class

The BufMgr class is the heart of the buffer manager. This is where you actually have to do some work.

```
class BufMgr {
        private:
                unsigned int    clockHand;  // clock hand for clock algorithm
                BufHashTbl      hashTable;  // hash table mapping (File, page) to frame number
                BufDesc         *bufTable;  // vector of status info, 1 per page
                unsigned int    numBufs;    // Number of pages in buffer pool
                BufStats        bufStats;   // Statistics about buffer pool usage


                const Status allocBuf(int& frame);   // allocate a free frame using the clock algorithm
        public:
                Page            *bufPool;   // actual buffer pool

```

```
14.                 BufMgr(const int bufs);
15.                 ~BufMgr();
16.
17.                 void advanceClock() {
18.                         clockHand = (clockHand + 1) % numBufs;
19.                 }
20.
21.                 const Status readPage(File* file, const int PageNo, Page*& page);
22.
23.                 const Status unPinPage(File* file, const int PageNo, const bool dirty);
24.
25.                 const Status allocPage(File* file, int& PageNo, Page*& page);
26.
27.                 const Status disposePage(File* file, const int PageNo);
28.
29.                 const Status flushFile(File* file);
30. };
```

Here are the definitions of each individual function:

BufMgr(const int bufs)

This is the class constructor. Allocates an array for the buffer pool with bufs page frames and a corresponding BufDesc table. The way things are set up all frames will be in the clear state when the buffer pool is allocated. The hash table will also start out in an empty state. We have provided the constructor.

~BufMgr()

Flushes out all dirty pages and deallocates the buffer pool and the BufDesc table.

const Status allocBuf(int& frame)

Allocates a free frame using the clock algorithm; if necessary, writing a dirty page back to disk. Returns BUFFEREXCEEDED if all buffer frames are pinned, UNIXERR if the call to the I/O layer returned an error when a dirty page was being written to disk and OK otherwise. This private method will get called by the readPage() and allocPage() methods described below.

Make sure that if the buffer frame allocated has a valid page in it, that you remove the appropriate entry from the hash table.

const Status readPage(File* file, const int PageNo, Page*& page)

First check whether the page is already in the buffer pool by invoking the lookup() method on the hashtable to get a frame number. There are two cases to be handled depending on the outcome of the lookup() call:

Case 1) Page is *not* in the buffer pool. Call allocBuf() to allocate a buffer frame and then call the method file->readPage() to read the page from disk into the buffer pool frame. Next, insert the page into the hashtable. Finally, invoke Set() on the frame to set it up properly. Set() will leave the pinCnt for the page set to 1. Return a pointer to the frame containing the page via the page parameter.

Case 2) Page is in the buffer pool. In this case set the appropriate refbit, increment the pinCnt for the page, and then return a pointer to the frame containing the page via the page parameter.

Returns OK if no errors occurred, UNIXERR if a Unix error occurred, BUFFEREXCEEDED if all buffer frames are pinned, HASHTBLERROR if a hash table error occurred.

const Status unPinPage(File* file, const int PageNo, const bool dirty)

Decrements the pinCnt of the frame containing (file, PageNo) and, if dirty == true, sets the dirty bit. Returns OK if no errors occurred, HASHNOTFOUND if the page is not in the buffer pool hash table, PAGENOTPINNED if the pin count is already 0.

const Status allocPage(File* file, int& PageNo, Page*& page)

This call is kind of weird. The first step is to to allocate an empty page in the specified file by invoking the file->allocatePage() method. This method will return the page number of the newly allocated page. Then allocBuf() is called to obtain a buffer pool frame. Next, an entry is inserted into the hash table and Set() is invoked on the frame to set it up properly. The method returns both the page number of the newly allocated page to the caller via the pageNo parameter and a pointer to the buffer frame allocated for the page via the page parameter. Returns OK if no errors occurred, UNIXERR if a Unix error occurred, BUFFEREXCEEDED if all buffer frames are pinned and HASHTBLERROR if a hash table error occurred.

```
const Status disposePage(File* file, const int PageNo);
```

Check to see if the page actually exists in the buffer pool by looking it up in the hash table. If it does exists, clear the page, remove the corresponding entry from the hash table and dispose the page in the file as well. Return the status of the call to dispose the page in the file.

```
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 for pages belonging to the file. For each page encountered it should:

a) if the page is dirty, call file->writePage() to flush the page to disk and then set the dirty bit for the page to false
b) remove the page from the hashtable (whether the page is clean or dirty)
c) invoke the Clear() method on the page frame.

Returns OK if no errors occurred and PAGEPINNED if some page of the file is pinned.

Getting Started

Start by downloading the project tar file from Learn@UW. Extract it on the CS lab machines and in it you will find the following files:

- `Makefile` - A make file. You can make the project by typing `make'
- `buf.h` - Class definitions for the buffer manager
- `buf.cpp` - Skeleton implementations of the methods. You should provide the actual implementations
- `bufHash.cpp` - Implementation of the buffer pool hash table class. Do not change
- `db.h` - Class definitions for the DB and File classes. You should not change these
- `db.cpp` - Implementations of the DB and File classes. You should not change these
- `page.cpp` - Implementation of the page class. Do not change
- `page.h` - Class definition of the page class. Do not change
- `error.h` - Error codes and error class definition
- `error.cpp` - Error class implementation
- `testbuf.cpp` - Test driver. Feel free to augment this if you want

What do you have to do?

Provide the implementation of the member functions in the `BufMgr` class in the file `buf.cpp`. There are totally about 7 functions that you need to implement. But, to be able to accurately do that, you need to go through code in most of the other classes. The above mentioned descriptions of these functions should serve as a guideline for you to implement them. You may also want to add more use cases into the `testbuf.cpp` file to more thoroughly test the correctness of your code but these will not be used in grading your assignment. Also feel free to add more error codes into the `error.h` file and their corresponding definitions into the `error.cpp` file.

Coding and Testing

We have defined this project in such a way that you can understand and reap the full benefits of object-oriented programming using C++. Your coding style should continue this by having well-defined classes and clean interfaces. Reverting to the C (procedural) style of programming is not recommended and will be penalized. The code should be well-documented. Each file should start with

the names and student ids of the team members and should explain the purpose of the file. Each function should be preceded by a few lines of comments describing the function and explaining the input and output parameters and return values.

Handing In

To handin the stage of your project:

- Make a README file with your project files that contains your group members names and UW netids.
- **One** member of your team should tar contents of your solution directory and upload it to the *Assignment 3 Submission* dropbox folder on Learn@UW and please double-check to see that you have the README file included. **You should include all source files in the solution even if you are modifying only a couple of them.** Do not include the object files or the compiled binary.
- Please remember it is due at 11:30PM on Thursday, 6th November 2014. The late policy will be as mentioned here. If you do not have the assignment complete by that time, please turn in whatever you have in hopes of getting partial credit.

We will compile your buffer manager, 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 `testbuf.cpp`), IT IS VERY IMPORTANT TO BE FAITHFUL TO THE EXACT DEFINITIONS OF THE INTERFACES as specified here.