CS 564 Homework #3

Mineral Project - Stage 3

Introduction

Minirel project stages will enable you to learn how to build a database system. You will build a working single-user DBMS that can execute certain simple SQL queries. The objective is to learn how a DBMS is organized and what goes on inside it when queries are executed. We will help you out by supplying the topmost and lowermost DBMS layers. The topmost layer is a parser that parses SQL queries and calls appropriate functions in the lower layers to perform relational operations. The lowermost layer is the disk I/O layer which reads and writes pages from and to the disk. In your case, the disk will just be the UNIX file system.

At the end of each stage, we will take a checkpoint and evaluate your work. In Stage 3, you will implement a buffer manager. Stage 4 involves implementing heap files - a collection of pages that are used to hold a relation. Stage 5 implements the database catalogs and various utilities. Finally, in Stage 6 you will implement several relational operators. In this class, you will implement Stages 3, 4, and 6. At the end of it all, you will have implemented a miniature DBMS. It will not do everything that a commercial DBMS does, but it should be able to handle a fairly useful set of queries. While doing these stages you shall learn how to tackle large programming projects and in the process have fun as well! Good luck!

Buffer Manager: Overview

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 the 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 the disk to the buffer pool, the DBMS software can update the information stored on the page, causing the copy in the buffer pool to be different from the copy on the disk. Such pages are termed “**dirty**”.

Since the database on the disk itself is usually about 100 times larger than the 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 residents. 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 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 the requested page from the disk into the frame that has been freed.

The following section describes the I/O layer of Minirel. 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, you need to know 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.

The Minirel I/O Layer

The lowest layer of the Minirel database systems is the I/O layer. This layer allows the upper level of the system to create/destroy files, allocate/deallocate pages within a file and read and write pages of a file. This layer consists of two classes: a file (class File) and a database (class DB) class. Let us start with a description of the DB class. We will provide you with an implementation of this layer.

```
class DB {
 public:
  DB();                          // initialize open file table
  ~DB();                         // clean up any remaining open files
  const Status createFile(const string & fileName) const;
                                 // create a new file
  const Status destroyFile(const string & fileName) const;
                                 // destroy a file release all space
  const Status openFile(const string & fileName, File* & file);
                                 // open a file
  const Status closeFile(File* file);
                                 // close a file
 private:
  OpenFileHashTbl   openFiles;
                          // hash table mapping files to file handles
};
```

The task of the DB class is to maintain a table of all files that are open. Each file corresponds to a relation (or an index) and is implemented as an OS (UNIX) file. If a file has already been opened (possibly by another query), then the DB class detects this (by looking in the openFiles table) and just returns the file handle (a pointer of type File*) without actually opening the UNIX file again. In this way (and in other similar situations that you will encounter later) the DBMS can maintain tight control over the objects that it uses.

The following is a description of what each method in the DB class does.

```
const Status createFile(const string & fileName) const
```

Creates a new UNIX file called fileName in the current working directory. Returns OK if no errors occurred, BADFILE if fileName is empty, FILEEXISTS if a file with this name already exists, and UNIXERR if a Unix error occurred.

```
const Status destroyFile(const string & fileName) const
```

Destroys the file named fileName. An open file cannot be destroyed. Returns OK if no errors occurred, BADFILE if fileName is empty, FILEOPEN if the file is open, and UNIXERR if a Unix error occurred.

```
const Status openFile(const string & fileName, File* & file)
```

Opens the file named fileName and returns a pointer to the corresponding File object. First checks if the file is already open. If so, then a pointer to the file object that has already been created is returned and a reference count (inside the File object) is incremented. Otherwise, the UNIX file is actually opened and used to initialize a new File object. The fileName and the pointer to this File object are inserted into the openFiles hash table. Returns OK if no errors occurred, BADFILE if fileName is empty, and UNIXERR if a Unix error occurred.

```
const Status closeFile(File *file)
```

This closes the file pointed to by the file. If after decrementing, the reference count for the file becomes 0, the corresponding UNIX file is closed, the entry in the openFiles table is removed and the file object is deleted. Returns OK if no errors occurred, BADFILEPTR if the file is NULL, and UNIXERR if a Unix error occurred.

```
class File {
  friend DB;

 public:
  const Status allocatePage(int& pageNo); // allocate a new page
  const Status disposePage(const int pageNo); // release space for the page
  const Status readPage(const int pageNo,
                        Page* pagePtr) const; // read page from file
  const Status writePage(const int pageNo,
                         const Page* pagePtr) const; // write page to file
  const Status getFirstPage(int& pageNo) const;
                                        // return pageNo of first page

 private:

  File(const string & fname);            // initialize file object
  ~File();                               // deallocate file object

  static const Status create(const string & fileName);
  static const Status destroy(const string & fileName);

  const Status open();
  const Status close();

  const Status intread(const int pageNo,
                 Page* pagePtr) const;         // internal file read
  const Status intwrite(const int pageNo,
                  const Page* pagePtr);        // internal file write

  string fileName;                     // The name of the file
  int openCnt;                         // # times file has been opened
  int unixFile;                        // unix file stream for file
};
```

The File class implements the DBMS abstraction of a File by providing a wrapper around the file system facilities provided by the OS. Many of the functions of this class (the private ones) are to be called only by the DB class and should not be called directly by the upper layers (which should use only the public methods). Hence the DB class is a friend of the File class. A (somewhat strange) example of this is the constructor and destructor methods of the File class which are private because a File object can only be created and destroyed by a DB object. The main thing that you should remember is that a File should never be constructed, destructed, created, destroyed, opened, or closed directly. These should only be done by calling the appropriate functions of the DB class. This is an example of how a well-designed DBMS is implemented in terms of layers.

We now describe the public methods of the File class. As a good object-oriented programmer, that is the only part of the File class that you should really be concerned with.

```
const Status allocatePage(int & pageNo)
```

Allocates a disk page for the current file and returns the page number in pageNo. Returns OK if no errors occurred and UNIXERR if a UNIX error occurred.

```
const Status disposePage(const int pageNo)
```

Releases the page pageNo. This page is added to the free list. Returns OK if no errors occurred, BADPAGENO if pageNo is not a valid page and UNIXERR if there is a Unix error.

```
const Status readPage(const int pageNo, Page* pagePtr) const
```

Reads page pageNo from disk into the memory address specified by pagePtr. Returns OK if no errors occurred, BADPAGENO if pageNo is not a valid page, BADPAGEPTR if pagePtr is not a valid address, and UNIXERR if there is a Unix error.

```
const Status writePage(const int pageNo, const Page* pagePtr) const
```

Writes page pageNo from the address specified by pagePtr to disk. Returns OK if no errors occurred, BADPAGENO if pageNo is not a valid page, BADPAGEPTR if pagePtr is not a valid address, and UNIXERR if there is a Unix error.

```
const Status getFirstPage(int& pageNo) const
```

Returns the (physical) page number of the first page of the file. This will be useful in the second part of the project. Returns OK if no errors occurred, UNIXERR if a Unix error occurred.

Both the DB and the File class can be found in the files db.C and db.h

Error Handling

We have defined a class called Error in the files error.h and error.C. You can create an instance of this class (e.g. Error err;) and then print error messages from any method of any class by invoking err.print(status). As you develop new classes you should add new error codes and messages in the Error class. Be sure to check the return codes of each function that you call and make sure that all functions return some status. ALL error messages should be printed using the Error class.

Buffer Manager: 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 clock hand 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 the disk is read into that location. The details of the algorithm are given below.

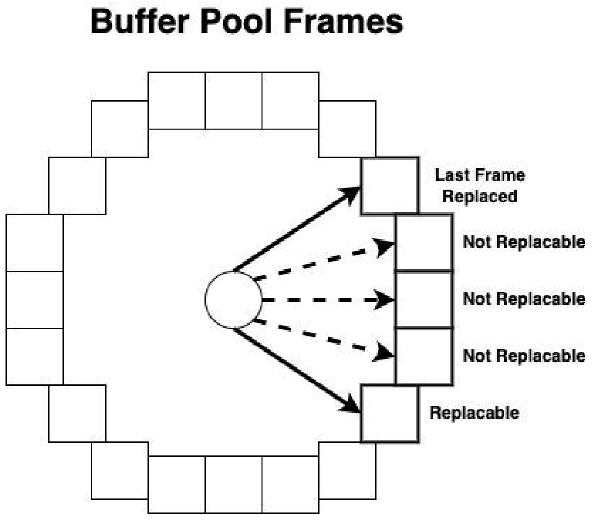

Figure 1

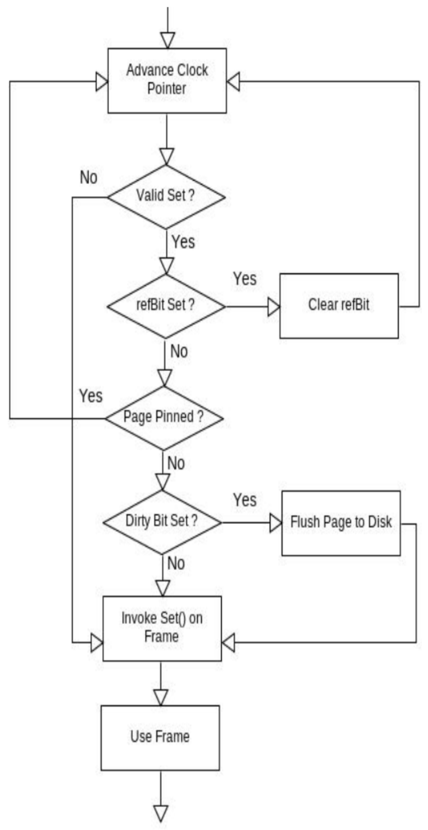

Figure 2. CLOCK Replacement Algorithm

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):

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

Here is the definition for the hash table.

```
class BufHashTbl
{
  private:
    hashBucket**  ht; // pointer to actual hash table
    int HTSIZE;
    int hash(const File* file, const int pageNo);
        // returns a value between 0 and HTSIZE-1

  public:
    BufHashTbl(int htSize); // constructor
    ~BufHashTbl(); // destructor

    // insert entry into hash table mapping (file,pageNo) to frameNo;
    // returns OK or HASHTBLERROR if an error occurred
    Status insert(const File* file, const int pageNo, const int frameNo);

    // Check if (file,pageNo) is currently in the buffer pool (ie. in
    // the hash table.  If so, return the corresponding frameNo via the
frameNo
    // parameter.  Else, return HASHNOTFOUND
    Status lookup(const File* file, const int pageNo, int& frameNo);

    // remove entry (file,pageNo) from hash table. Return OK if
    // page was found.  Else return HASHTBLERROR
    Status remove(const File* file, const int pageNo);
};
```

`const 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.

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

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

```
const 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) opens 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.

Clock hand: 1. check valid bit pin count (evict page if pincnt = 0) reference bit (try to avoid removing from buffer if something has most recently been accessed), keep track of user counts as well to see how many people are using the frame.

The BufMgr Class

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

numBufs: memset is used to fill buffer with page

bufftable: 1 to 1 correspondence to bufPool. Check if valid frame, if its hot frame or not etc. (meta data about each page) see phone photo bufdesc. RefBit = True if you just have accessed it!

hashTable:

```
 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

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

accesses, disk reads, disk writes

pointer to array of pages:

```
    void advanceClock()
    {
       clockHand = (clockHand + 1) % numBufs;
    }

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

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

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

    const Status flushFile(File* file);
  };
```

`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 a 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)` doesn't insert...but will tell if possible to insert

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, 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. The 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 the 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. The 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 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 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 Zip file for this stage (your instructor will give the precise instruction). Inside the Zip file, 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.C - Skeleton implementations of the methods. You should provide the actual implementations.
bufHash.C - 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 this.
db.C - Implementations of the DB and File classes. You should not change these
page.C - 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.C - Error class implementation
testbuf.C - Test driver. Feel free to augment this if you want.

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.

Submission Instructions

The Zip file for Stage 3 is called 564-part3.zip and can be downloaded by clicking on "Files" in Canvas.

Each project group should create a directory named after their group name. Suppose the group name is G6. Then the directory is "G6". In this directory, please place the file buf.C (which we have asked you to change) and a file G6.txt, which lists the full names and emails of all group members.

Zip the directory into a file G6.zip, then upload it to Canvas. Each group should have just ONE member uploading this zip file. To upload, click on "Assignments", then on Project Stage 3. You should see a button that allows you to upload the zip file.

The above instruction is for a fictional group named G6. You should modify it accordingly, using your true group name.

IMPORTANT:
1) You should have been editing only file buf.C. So you should upload only that file (plus the G6.txt file). We will add your buf.C to our code and compile it. Make sure that your code compiles and passes all the tests in testbuf.c ON CS MACHINES, as we will compile your code using CS machines.
2) You must submit by the deadline. No exception. We need this so that we can release the next stage (which contains the solution to this stage) on time. Even if you have not passed all the tests, just submit what you have by the deadline.

Logistics

These stages will be done in C++. For many of you, this is likely to be the biggest programming project you have ever done so far and so it might be useful to keep the following points in mind.

Platform: The stages will be compiled and tested on the CS department Unix machines using the latest supported g++ compiler. You are free to do some development using other platforms but you must make sure that your project works with the official configuration.
Warnings: One of the strengths of C++ is that it does a lot of compile-time checking of the code (consequently reducing run-time errors). Try to take advantage of this by turning on as many compiler warnings as possible. The makefile that we will supply will have -Wall on as default.
Software Engineering: A large project such as this requires significant design effort. Spend some time thinking before you start writing code.
Partners: Each stage should be done by the entire project team. It is unlikely that one person will be able to finish the stages. In real life, all software development is a team effort and you will be well-served if you can pick up some team management skills during this project. Disagreements can and will happen and it is your responsibility to get the project going regardless of differences. All members of the team will get the same grade for each project stage.
Due Dates: Your group must submit the solution by the due dates. No late submission will be accepted. This is because the code for the next stage will contain the solution to the previous stage. So we cannot release the next stage until all groups have submitted the solution for the previous stage. Thus, if you submit late, you are delaying the start of the next stage. This is not acceptable because we are already on a very tight project schedule. Hence, if you do not submit by the deadline, we will have no choice but to

grade your submission as zero. Even if you only have a partial code, you should still submit it by the deadline.

Exams: The final thing that you should note is that all topics covered in the stages will be fair game for questions during the midterm and final. So make sure you know about what you (and your partner) did for the project.