CS 564 Programming Project 2
Buffer Manager

INTRODUCTION

The goal of the BadgerDB projects is to allow students in CS 564 to learn about the internals of a data processing engine. In this assignment, you will build a buffer manager, on top of an I/O layer that we provide.

Logistics. BadgerDB is coded in C++ and runs on the CSL machines. Here are a few logistical points:

- **Platform:** The stages will be compiled and tested on a Linux machine. We will use the g++ compiler(>= 9.3) on those machines. You are free to develop on other platforms, but you must make sure that your project works with the official configuration.

  Some points to note for working under Windows or Mac:

  - MacOS default to Apple's version of clang compiler(g++ is aliased to it), and it may include all standard headers by default. However, g++ or the original clang doesn't. So you could see compiling errors saying undefined functions when compiling on Linux while it works on MacOS.
  - Windows and MacOS does not care about cases of the file name, while Linux does.
  - Windows has a different newline character from Linux. But the compiler does not care. Just in case you run into some syntax error with scripts.
  - Windows Subsystem of Linux works when it works.

  If you need a Linux machine, you can use CSL's Linux machines by SSH on to the `rockhopper` pool remotely, using the machines `rockhopper-01.cs.wisc.edu` through `rockhopper-09.cs.wisc.edu`. Since you are enrolled in a CS class, you can activate an account (if you already don't have one) in the CSL through the following link: `https://csl.cs.wisc.edu/services/user-accounts`. When it compiles and runs on CSL's machines you are good to go.

- **Warnings:** One of the strengths of C++ is that it does compile time code checking so it can reduce run-time errors(definitely not on par with Rust, but I digress). 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.

- **Tools:** Always be on the lookout for tools that might simplify your job. Example: make for compiling and building your project, makedepend for automatically generating dependencies, perl for writing test scripts, valgrind for tracking down memory errors, gdb for debugging, and git/svn for version control. While we will not explicitly educate you about each of these, feel free to seek the TA's advice.
- **Software Engineering:** A large project such as this requires significant design effort. Spend some time thinking before you start writing code.

Evaluation. We will run a bunch of our own (private) tests to check your code. So please develop tests beyond the ones that we give you to stress test your solution. We will also browse your code to review your coding style and read your Doxygen-generated files. 80% of each project grade is allocated to the correctness test, and 20% for your coding style and clarity of documenting your code.

Academic Integrity. You are not allowed to share any code with other students in the class. Nor will you attempt to use any code from previous offerings of this course. Deviations from this will be punished to the fullest extent possible. We will use a code diffing program to find cheaters.

THE BADGERDB I/O LAYER

The lowest layer of the BadgerDB 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 to read and write pages of a file. This layer consists of two classes: a file (class File) and a page (class Page) class. These classes use C++ exceptions to handle the occurrence of any unexpected event.

Implementation of the File class, the Page class, and the exception classes are provided to you. To start, you can copy `BufMgr.tar.gz` to your private workspace, and expand this tarball using the following command:

```
/bin/zcat bufmgr.tar.gz | /bin/tar -xvf -
```

The code has been adequately commented to help you with understanding how it does what it does. Please use *Doxygen* as shown below to generate documentation files on these files. Inside the **bufmgr** directory run the following command to generate documentation files: `> make doc`. The doc files will be generated in docs directory. You can now open the `docs/index.html` file inside the browser and go through description of classes and their methods to better understand their implementation. Note that above, > is shell prompt on Linux machines and hence not part of the command.

THE BADGERDB BUFFER MANAGER

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 often larger than the amount of main memory that is available for the buffer pool, only a subset of the database pages 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 checks to see if the requested page is already in the one of the frames that constitutes 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 BadgerDB so that you understand its capabilities (described on the previous page). 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 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 shows the conceptual layout of a buffer pool. Figure 2 illustrates the execution of the clock algorithm.

In Figure 1, 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 (as-

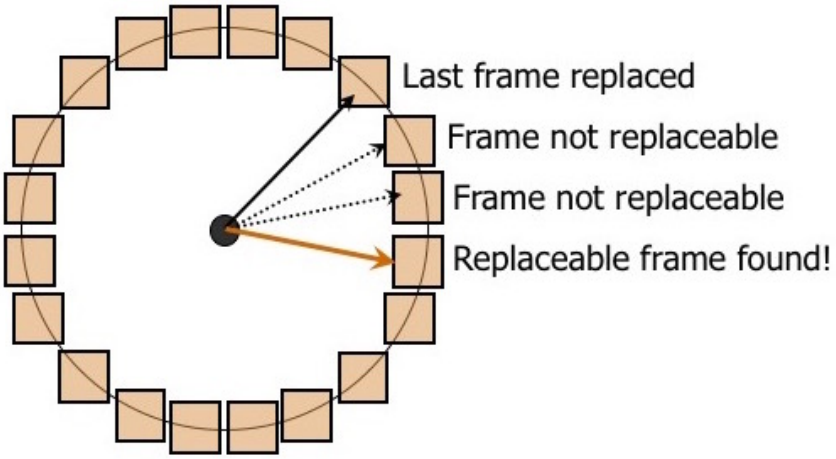

Figure 1: Structure of the Buffer Manager

suming 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 and a new page from disk is read in to that location. The details of the algorithm is given below.

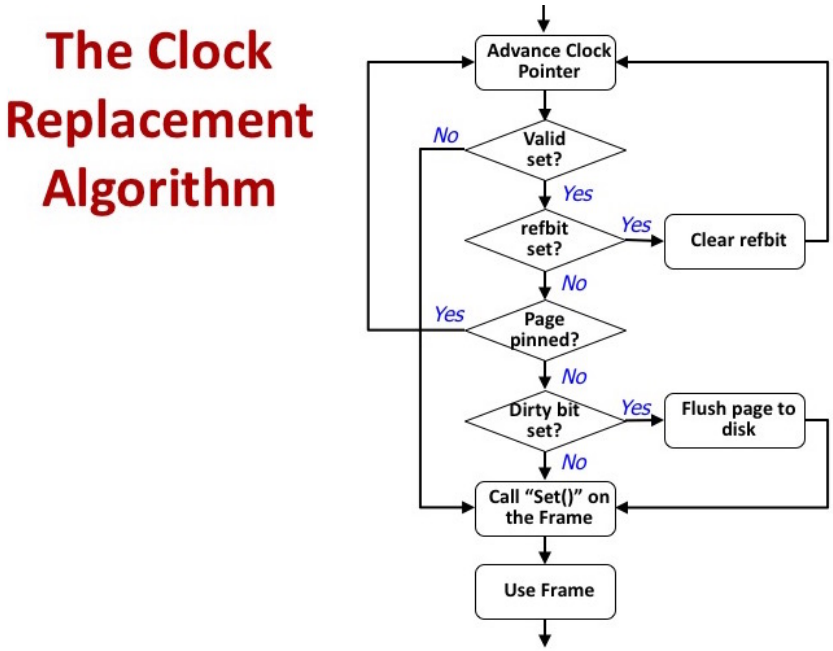

Figure 2: The Clock Replacement Algorithm

The Structure of the Buffer Manager.

The BadgerDB buffer manager uses three C++ classes: BufMgr, BufDesc and BufHashTbl. There is only one instance of the BufMgr class. A key component of this class is 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 that is used to describe the state of each frame in the buffer pool. A hash table is used to keep track of the pages that are currently resident in the buffer pool. This hash table is implemented by an instance of the BufHashTbl class. This instance is 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.

`std::shared_ptr` is C++'s shared pointer, which tracks the references of an object, and automatically frees it when nobody is pointing to it. It works like a normal pointer when dereferencing, or assigning, but instead of `new`, `make_shared` is used to return a shared pointer for a new object, and you do not need to free it manually at any time.

```
struct hashBucket {
    File file;    // pointer to a file object (more on this below)
    PageId pageNo;  // page number within a file
    FrameId frameNo;  // frame number of page in the buffer pool
    std::shared_ptr<hashBucket> next;    // next bucket in the chain
};
```

Here is the definition of the hash table.

```
  class BufHashTbl {
    private:
      /**
       *  Size of Hash Table
       */
      int HTSIZE;
      /**
       * Actual Hash table object
       */
      std::vector<std::shared_ptr<hashBucket>> ht;

      /**
       * returns hash value between 0 and HTSIZE-1 computed using file and
          pageNo
       */
      int hash(const File& file, const PageId pageNo);

    public:
```

```
    /**
     * Constructor of BufHashTbl class
     */
    BufHashTbl(const int htSize); // constructor

    /**
     * Insert entry into hash table mapping (file, pageNo) to frameNo.
     */
    void insert(const File& file, const PageId pageNo, const FrameId frameNo
        );

    /**
     * Check if (file, pageNo) is currently in the buffer pool (ie. in
     * the hash table).
     */
    void lookup(const File& file, const PageId pageNo, FrameId &frameNo);

    /**
     * Delete entry (file,pageNo) from hash table.
     */
    void remove(const File& file, const PageId pageNo);
  };
```

The BufDesc Class. The BufDesc class is used to keep track of the state of each frame in the buffer pool. It is defined as follows.

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 BufDesc's private variables 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.

```
class BufDesc {
  public:
    /**
     * Constructor of BufDesc class
     */
    BufDesc() { clear(); }

  private:
    friend class BufMgr;
    /**
     * Pointer to file to which corresponding frame is assigned
```

```
     */
    File file;

    /**
     * Page within file to which corresponding frame is assigned
     */
    PageId pageNo;

    /**
     * Frame number of the frame, in the buffer pool, being used
     */
    FrameId frameNo;

    /**
     * Number of times this page has been pinned
     */
    int pinCnt;

    /**
     * True if page is dirty;  false otherwise
     */
    bool dirty;

    /**
     * True if page is valid
     */
    bool valid;

    /**
     * Has this buffer frame been reference recently
     */
    bool refbit;

    /**
     * Initialize buffer frame for a new user
     */
    void clear();

    /**
     * Set values of member variables corresponding to assignment of frame
        to a
     * page in the file. Called when a frame in buffer pool is allocated to
        any
     * page in the file through readPage() or allocPage()
     */
    void Set(File &file, PageId pageNum);

    void Print();
};
```

The BufMgr Class. The BufMgr class is the heart of the buffer manager. This is where you write your code for this assignment.

```
class BufMgr {
  private:
    /**
     * Current position of clockhand in our buffer pool
     */
    FrameId clockHand;

    /**
     * Number of frames in the buffer pool
     */
    std::uint32_t numBufs;

    /**
     * Hash table mapping (File, page) to frame
     */
    BufHashTbl hashTable;

    /**
     * Array of BufDesc objects to hold information corresponding to every
       frame
     * allocation from 'bufPool' (the buffer pool)
     */
    std::vector<BufDesc> bufDescTable;

    /**
     * Maintains Buffer pool usage statistics
     */
    BufStats bufStats;

    /**
     * Advance clock to next frame in the buffer pool
     */
    void advanceClock();

    /**
     * Allocate a free frame.
     */
    void allocBuf(FrameId &frame);

  public:
    /**
     * Actual buffer pool from which frames are allocated
     */
    std::vector<Page> bufPool;

    /**
     * Constructor of BufMgr class
```

```
     */
    BufMgr(std::uint32_t bufs);

    /**
     * Reads the given page from the file into a frame and returns the
        pointer to
     * page. If the requested page is already present in the buffer pool
        pointer
     * to that frame is returned otherwise a new frame is allocated from the
     * buffer pool for reading the page.
     */
    void readPage(File& file, const PageId pageNo, Page*& page);

    /**
     * Unpin a page from memory since it is no longer required for it to
        remain in
     * memory.
     */
    void unPinPage(File& file, const PageId pageNo, const bool dirty);

    /**
     * Allocates a new, empty page in the file and returns the Page object.
     * The newly allocated page is also assigned a frame in the buffer pool.
     */
    void allocPage(File& file, PageId &pageNo, Page*& page);

    /**
     * Writes out all dirty pages of the file to disk.
     * All the frames assigned to the file need to be unpinned from buffer
        pool
     * before this function can be successfully called. Otherwise Error
        returned.
     */
    void flushFile(File& file);

    /**
     * Delete page from file and also from buffer pool if present.
     * Since the page is entirely deleted from file, its unnecessary to see
        if the
     * page is dirty.
     */
    void disposePage(File& file, const PageId PageNo);

    /**
     * Print member variable values.
     */
    void printSelf();

    /**
     * Get buffer pool usage statistics
     */
```

```
    BufStats& getBufStats() { return bufStats; }

    /**
     * Clear buffer pool usage statistics
     */
    void clearBufStats() { bufStats.clear(); }
};
```

This class is defined as follows:

`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.

`void advanceClock()`
Advance clock to next frame in the buffer pool.

`void allocBuf(FrameId& frame)`
Allocates a free frame using the clock algorithm; if necessary, writing a dirty page back to disk. Throws BufferExceededException if all buffer frames are pinned. 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.
Notes that the `frame` variable is a reference, so the function works by assigning the number to this variable. It is the same for the `PageNo` and `page` variables below, except `file`.
`void readPage(File& file, const PageId pageNo, Page*& page)`
First check whether the page is already in the buffer pool by invoking the lookup() method, which may throw HashNotFoundException when page is not in the buffer pool, 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.

For why `page` is not a shared pointer, it's because the page is owned and managed by the buffer pool. It is freed if and only if the buffer manager is freed. If `page` is a shared pointer, it will try to free this part of memory, which wouldn't work properly. This is the time when we don't need smart pointers to manage it.

`void unPinPage(File& file, const PageId PageNo, const bool dirty)`
Decrements the pinCnt of the frame containing (file, PageNo) and, if dirty == true, sets the dirty bit. Throws PAGENOTPINNED if the pin count is already 0. Does nothing if page is not found in the hash table lookup.

`void allocPage(File& file, PageId& PageNo, Page*& page)`
The first step in this method is to to allocate an empty page in the specified file by invoking the file.allocatePage() method. This method will return a 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.

`void disposePage(File& file, const PageId pageNo)`
This method deletes a particular page from file. Before deleting the page from file, it makes sure that if the page to be deleted is allocated a frame in the buffer pool, that frame is freed and correspondingly entry from hash table is also removed.

`void flushFile(File& file)`
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) and (c) invoke the Clear() method of BufDesc for the page frame.

Throws PagePinnedException if some page of the file is pinned. Throws BadBufferException if an invalid page belonging to the file is encountered.

GETTING STARTED

When you expand the tarball at `BufMgr.tar.gz` you will have a directory called bufmgr. In this directory you will find the following files:

- `Makefile` : A make file. You can make the project by typing 'make'.
- `README` : Contains some more instructions on getting hands on the project.
- `main.cpp` : Driver file. Shows how to use File and Page classes. Also contains simple test cases for the Buffer manager. You must augment these tests with your more rigorous test suite.
- `buffer.h` : Class definitions for the buffer manager

- `buffer.cpp`: Skeleton implementation of the methods. Provide your actual implementation here.
- `bufHash.h` : Class definitions for the buffer pool hash table class. Do not change.
- `bufHash.cpp` : Implementation of the buffer pool hash table class. Do not change.
- `file.h` : Class definitions for the File class. You should not change this file.
- `file.cpp` : Implementations of the File class. You should not change this file.
- `file_iterator.h` : Implementation of iterator for pages in a file. Do not change.
- `page.h` : Class definition of the page class. Do not change.
- `page.cpp` : Implementation of the page class. Do not change.
- `page_iterator.h`: Implementation of iterator for records in a page.
- `exceptions directory`: Implementation of all your exception classes. Feel free to add more files here if you need to.

Coding and Testing. We have defined this project so 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. The code should be well-documented, using Doxygen style comments. Each file should start with your name and student id, and should explain the purpose of the file. If you define new functions/methods, each of them should be preceded by a few lines of comments describing what it does and explaining the input and output parameters and return values. You need to also comment a block of code to explain why it is done in this way when you feel its purpose is unclear by just looking, but this kind of comments don't need to be in doxygen style.

DELIVERABLES

You are required to submit all the necessary material in a single zipped folder (use GZip or WinZip). Upload the zipped folder on `Canvas` (Programming Project 2). The folder should include **only** the source files. We will copy the buffer.cpp and exceptions directory, compile the buffer manager, and then run it. After that we will run your own testing code.

Right before submission, take a few minutes to double check that what you are submitting is working! From past experience, changing some code without testing leads to a large part of errors.