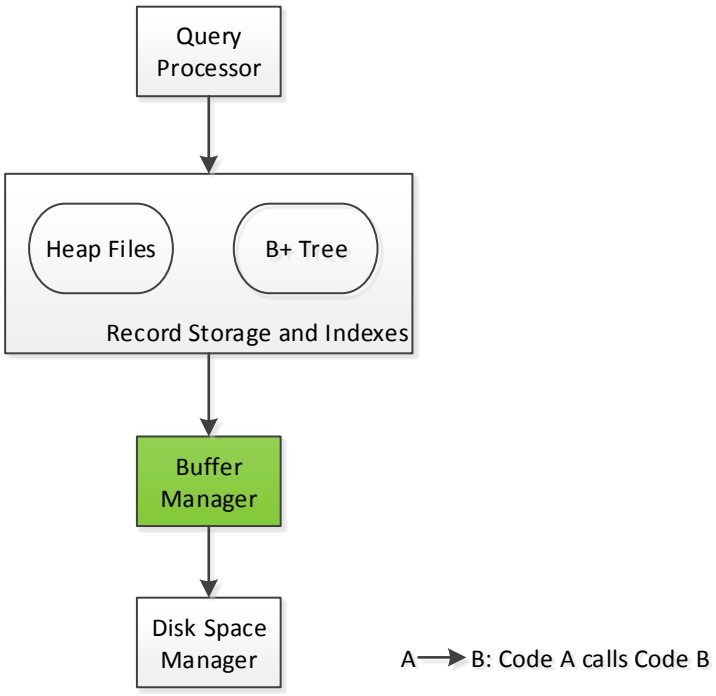
Project 3: Buffer Manager

This Assignment is due at **23:59 pm, October 20, 2014** via CMS. It is worth a total of 100 points and counts for 20% of your grade. Read first the whole assignment carefully, and review the relevant textbook materials, before you start implementing.



Introduction

In this assignment, you will implement a buffer manager. The buffer manager is responsible for the follow tasks:

- Pinning a page, reading the page from disk into an empty frame if needed.
- Unpinning a page by reducing the pin counter for that page.
- Freeing up space that is allocated to a given page on the disk.
- Flushing a page or all pages to the disk, if required.
- Keeping track of statistics of operations.

In addition, you will implement the **LRU** and **MRU** buffer replacement policies.

You are given the interface for the Buffer Manager, and you need to implement it. The interfaces for the underlying Disk Space Manager (DB class and Page class) are given. In particular, you should read *db.h,* the interface to the DB class, which you will use extensively in this assignment.

Design Overview

The buffer pool is a collection of *frames* (a page-sized sequence of bytes in main memory) that is managed by the Buffer Manager. It should be stored as a dynamically allocated array of **Frame** objects pointed to by the **frames** pointer in BufMgr. Each **Frame** object has the following fields:

pid, data, pin_count, dirty

pid is a **PageID** object, *data* is a pointer to a **Page** object, *pin_count* is an integer, and *dirty* is a boolean. This describes the page that is stored in the corresponding frame. The definition of **Frame** can be found in *frame.h*. However, unlike the BufMgr interface above which we require you to conform to, feel free to design your own interface here.

Recall that a page is identified by a unique *pid* that is generated by the DB class when the page is allocated. The **PageID** type is defined as an integer type in *minirel.h*.

The Buffer Manager Interface

All the following methods will return OK if the operation is successful, and FAIL in error conditions. They are described below:

BufMgr(int bufSize, const char* policy):

- Allocate **bufSize** frames in main memory.
- Set the LRU or MRU replacement policy according to **policy**, **policy** is either "LRU" or "MRU", otherwise, print out an error message and exit. Hint: set the **replacer** pointer in the class BufMgr appropriately.

~BufMgr():

- Flush all dirty pages in the pool to disk before shutting down.
- Deallocate any main memory structures.

Status PinPage(PageID pid, Page*& page, bool isEmpty):

- Check if this page is in the buffer pool. If so, increase its pin_count, and set the output page pointer **page** to the page pointed to by the frame.
- If there is a free frame in the buffer pool, then set the page id of the frame to **pid** and set the pin_count to 1.
- If not, choose a victim frame according to the buffer replacement policy (either **LRU** or **MRU**), and write the current page in that frame back to disk if the frame holds a page and that page is dirty.
- If this page is not in buffer pool, and **isEmpty** is false, you should read the page of **pid** from disk. You can do this by calling the method DB::ReadPage using the MINIBASE_DB object. If **isEmpty** is true, you do not need to read the page from

disk, but you should make sure that the frame's data pointer is not null, and the frame's PageID is set correctly.

Status UnpinPage(PageID pid, bool dirty):

- Find the frame in which **pid** resides, and decrement the pin_count of the frame (If *pin_count* is ZERO before this decrement operation, or the **pid** is not in the buffer pool, return FAIL).
- Set the dirty bit in the frame to true if **dirty** has the value true (i.e., in this case the upper layer told the buffer manager that it has modified the page).

Status NewPage(PageID& firstPid, Page*& firstPage, int howMany):

- Allocate a run of **howMany** new pages (use the DB class method DB::AllocatePage appropriately).
- Try to pin the page with pid **firstPid** in the buffer pool.
- **firstPid** and **firstPage** should be set appropriately.

Status FreePage(PageID pid):

- This method removes a page from the buffer pool and deallocates it from the database.
- Check the pin count if the page is in the buffer pool. If the pin count is 1, unpin the page. If the pin count is > 1, return FAIL.
- Deallocate the page from the database using the DB::Deallocate method. Note that you should deallocate the page even if it is not in the buffer pool.
- Clean up the frame as necessary (e.g. set pid to INVALID_PAGE). Do you need to write the page back to disk?

Status FlushPage(PageID pid):

- Find the frame that holds in the page. If that frame is still pinned, return FAIL.
- Otherwise, write the page back to disk (using DB::WritePage) if it is dirty.

Status FlushAllPages():

- Flush all pages of the buffer pool to disk in all cases. However, if there is some page pinned in the buffer or some error occurred while flushing pages out, return FAIL. Otherwise return OK.

unsigned int GetNumOfUnpinnedFrames():

- Return the number of unpinned buffer frames in the buffer pool.

int FindFrame(PageID pid):

- This is a private method that will be invoked only within the BufMgr class. It returns the ID of the frame, referred to as *frame number*, which the page occupies 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, and you should return INVALID_FRAME.

Finally, the two functions for resetting the statistics and reporting the statistics using standard input/output stream libraries, ResetStat() and PrintStat(), have been implemented for you. Your buffer manager will collect the following statistics:

1) The number of PinPage requests made.
2) The number of PinPage requests that result in a hit (i.e., the page is not in the buffer when the pin request is made).
3) The number of "dirty" pages written to disk.

You are responsible for collecting incrementing these counters as appropriate. See the comments in bufmgr.h for more information.

The LRU and MRU Buffer Replacement Policy

Theoretically, the best candidate page for replacement is the page that will be last requested in the future. Since implementing such policy requires an oracle that predicts the future (which, despite much effort, no one has managed to implement), all buffer replacement policies try to approximate it one way or another. The LRU policy, for example, uses the past access pattern as an indication for the future. The description of LRU and MRU buffer replacement policies can be found in Section 9.4.1 of the text book. Suggested interfaces for the LRU and MRU buffer replacement policies can be found in *lru.h* and *mru.h*, respectively. However, unlike the BufMgr interface above which we require you to conform to, feel free to design you own interface here.

Notes on Space Manager

Some buffer manager functionality needs to allocate and deallocate pages by calling the AllocatePage() and DeallocatePage() functions in the DB class (see the descriptions in the buffer manger interface for more details). The DB class maintains a *space map* to manage the pages created by the database, see the comments in db.h for more details. The space map is also stored in pages which are managed by the Buffer Manager. (You can take a glance at the DB::AllocatePage() and DB::DeallocatePage() functions in db.cpp). Therefore, if the Buffer Manager is not implemented correctly, these two functions might behave abnormally as well (e.g. you might see runtime errors when trying to allocate a new page).

Compilation and Testing

The source code for the project can be downloaded from the course management system. If you unzip this file, you will see the following directories and files.

Top-level Directory

- This contains the Visual Studio 2010 project and solution files for this assignment

Directories "src" and "include":

- **bufmgr.cpp --** the code skeleton for the Buffer Manager, where you will find the detailed specs of the methods **that you need to implement** as well as the necessary interfaces.
- **bufmgr.h** - the definition for the class BufMgr.
- **test.cpp, bmtest.cpp, test.h, bmtest.h--** the source codes for the tests that the Buffer Manager needs to pass. RunTests runs the tests on the Buffer Manager.
- **main.cpp** - the main routine that initializes the Minibase, including the buffer manager, and run the test.
- **db.h, db.cpp, page.h –** interfaces to the DB class and Page classes.
- **da_types.h, new_error.h, minirel.h, system_defs.h -** the header files for Minibase which define various types, global variables, etc.
- **replacer.h, lru.h, mru.h, frame.h-** the "private" classes used by BufMgr. Feel free to use these interfaces, or come up with your own.

Directory "lib" contains the library files you will need in order to generate an executable.

What You Need to Do

You are required to modify and fill in the gaps in **bufmgr.cpp, bufmgr.h.**

- You need to implement the methods listed in bufmgr.cpp based on specifications as described in bufmgr.h. You are free to add any private methods or members into BufMgr, or make a function inline if you want to.
- Your buffer manager should implement the **LRU** and **MRU** replacement policy, described in the textbook.
- Your code should pass the 5 provided tests in bmtest.cpp.

Files to complete: bufmgr.cpp, bufmgr.h.

Files to create/complete: lru.h, lru.cpp, mru.h , mru.cpp, frame.h, frame.cpp.

Please **do not** modify any other files in the project.

What to Turn In

You should submit the same set of files given to you at the beginning of this assignment, plus any additional header and source files you have created for this assignment. The code should compile under the Visual Studio solution file you submit. Before submission to CMS, **delete the files generated by Visual Studio (such as Debug folder, BufMgr.sdf)** and zip up your set of files into *Submission.zip*.

Please make sure to start early.

Good Luck!