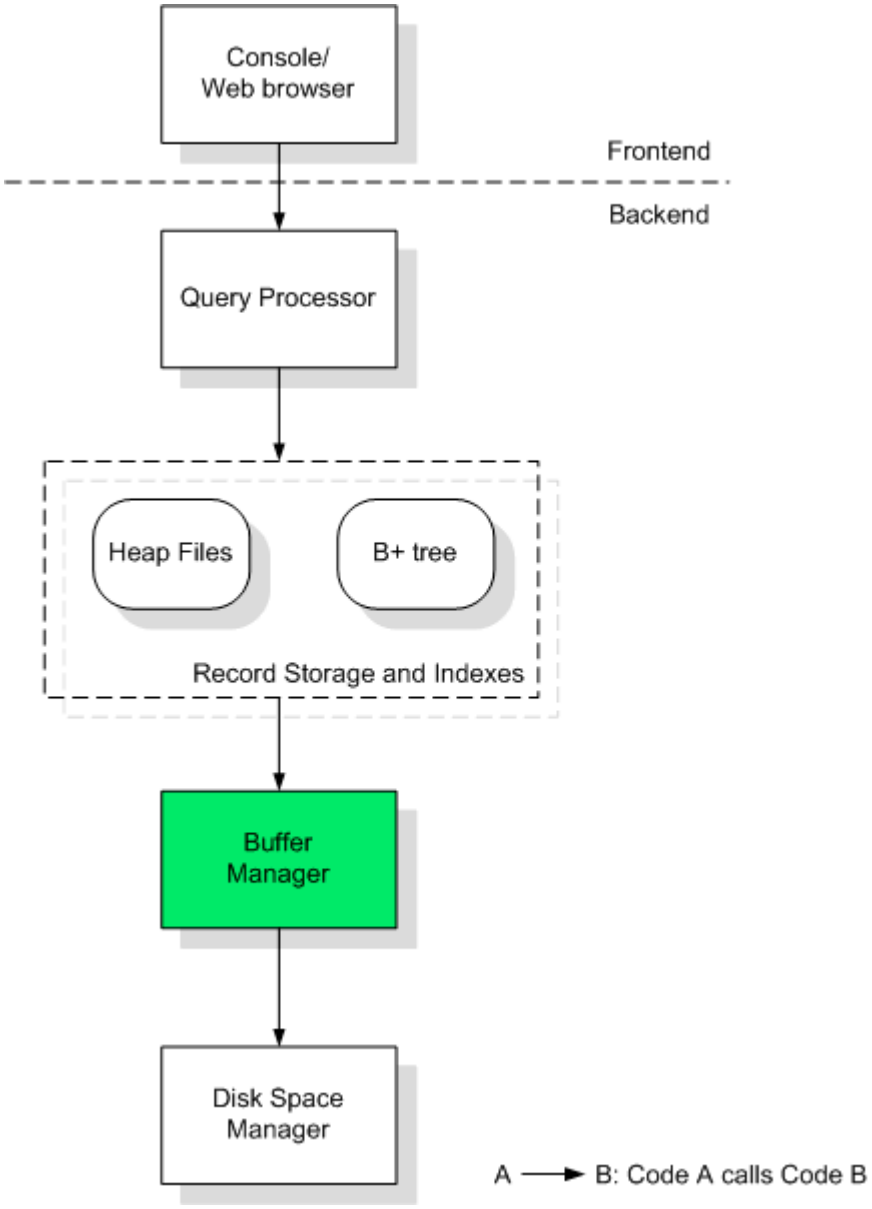
Project 2: Buffer Manager



First read the whole assignment carefully, and review the relevant textbook materials, before you start implementing.

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

- Pins a page to the buffer, bringing a page from the disk, if needed
- Unpins a page in the buffer by reducing the pin counter for that page
- Frees up space that is allocated to a given page on the disk.
- Flushes a page or all pages to the disk, if required
- Keeps track of statistics of operations on buffer pool

In addition, you will implement a buffer replacement policy (optional: **LRU**).

This assignment has been designed to be more flexible compared to the last one. You are still given a well-defined interface for the major database component **Buffer Manager**, and required to implement it. However, you are free to design the interfaces for other classes that will be used in this assignment. We have provided interfaces for these classes as well for your reference, and you are free to base your implementation on them as well.

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

Design Overview

The *buffer pool* is a collection of *frames* (page-sized sequence of main memory bytes) that is managed by the **Buffer Manager**. It should be stored as an array **frames[bufsize]** of **Frame** objects. Each **Frame** object is a record with 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 structure 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 you own interface here.

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

The Buffer Manager Interface

In the following methods, return **OK** if the operation is successful, and **FAIL** in error conditions (e.g. the buffer is full). The methods that you need to implement are described below:

BufMgr(int bufsize): Allocate "bufsize" pages (frames) for the pool in main memory.

~BufMgr(): Flush all dirty pages in the pool to disk before shutting down, and deallocate the buffer pool in main memory.

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 not, choose a victim frame according to the buffer replacement policy, and write the current page in that frame back to disk if the frame holds a page and it is dirty. If this page is not in buffer pool, and **isEmpty** is true, the buffer manager pins the frame to an empty page, by setting the page ID of the frame to the input **pid**. Otherwise, the frame should read the input page of **pid** from disk (using the appropriate **DB** class method **DB::ReadPage**). The global instance of **Disk Space Manager** is **MINIBASE_DB**. You should involve the member functions of **DB** class from **MINIBASE_DB**.

Status UnpinPage(PageID pid, bool dirty): Find the frame, and decrement **pin_count** of the frame. If **pin_count**==0 before this decrement operation, return **FAIL**). The condition **dirty**==true holds if the client has modified the page. In this case, this call should set the dirty bit for this frame.

Status NewPage(PageID& firstPid, Page*& firstPage, int howMany): Allocate a run of new pages according to **howMany** (using the appropriate **DB** class method **DB::AllocatePage**), and set **firstPid** and **firstPage** appropriately. Then try to pin the first page in the buffer.

Status FreePage(PageID pid): Find the page in the buffer pool, and deallocate the page (using **DB::DeallocatePage**).

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, 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) Number of PinPage requests made.
2) Number of PinPage requests that result in a miss (i.e. the page is not in the buffer when the pin request is made)
3) Number of "dirty" pages written to disk.
4) Total time taken to run the test. (This will be collected by the code in the test cases)

The Buffer Replacement Policy

A description of buffer replacement policies can be found in Section 9.4.1 of the textbook and was discussed in the lectures (see the slides). It is required to implement a replacement policy. In particular, a solution where the page to evict is chosen at random is considered acceptable (for a mark S). The current definition of the replacement policy is defined in **replace.h**, where an interface to implement the **Clock** policy is given. However, unlike the **BufMgr** interface above which we require you to conform to, feel free to design you own interface here.

Optionally (for a mark S+), the LRU replacement policy is to be implemented. Theoretically, the best candidate page for replacement is the page that will be last requested in the future. Since implementing such policy requires a future predicting oracle, 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.

Compilation and Testing

Assuming your work directory is ~/, copy the file *BufMgr.zip* with the skeleton code into your working directory and unzip it using the command *unzip BufMgr.zip*. This will result in a folder *~/BufMgr/* containing the source code and CMake build files. Create a directory *~/BufMgr-bin/* , change to this directory and run *cmake ~/BufMgr/* in order to create the makefiles for the project. Now run *make* to compile the source code. (To re-compile the source code after having modified parts of the code, it suffices to run *make*; *cmake* does not need to be run again.) The resulting executable file is *./minibase-bufmgr*. Running the executable before completing the practical at least partially gives a segmentation fault: The initialization of the database calls the constructor and function **PinPage** in *bufmgr.cpp* and therefore you will not be able to run the tests until they are completed correctly.

The directory *~/BufMgr/include* contains header files and the directory *~/BufMgr/bufmgr* contains the test files and a skeleton of the code you will be completing. The important files are:

- **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 Minibase, including the buffer manager, and run the test.
- **db.h, page.h –** interfaces to the **DB** and **Page** classes.
- **db.cpp.for-reference** – (in directory doc/) the implementation of the **DB** class (**do not** modify this file --- it has been provided solely to understand how the **DB** class works.)
- **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, 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* and *bufmgr.h*, specifically:

- 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.

- Implement a buffer replacement policy of your choice (for mark S). For mark S+, your buffer manager should implement the **LRU** replacement policy, described in the textbook and explained in the lectures.

- Your buffer manager should pass all 5 tests in the sequential order of 1 through 5.

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

Files to create/complete: *replacer.h*, *replacer.cpp*, *frame.h*, *frame.cpp.*

Please **do not** modify any other files in the project. **Remember** that if you add any interface implementation, you also need to add the name of the file in *bufmgr/CMakeLists.txt*.

Debug Your Code

For segfaults, first try to compile with debugging symbols (-g), and run with valgrind. In 99% of cases this will point you to the bug.

One way (very simple) of debugging your code is to add print statements in the test case. Read the test file (it's pretty straightforward), and add a print statement any time you feel it necessary to check whether the variable is assuming the value you expect. [NOTE: do not forget to flush otherwise printing may be misleading as the message to be printed may not be flushed automatically in case of a segfault!]

If you want to do things properly, then use the C debugger from command line. Or many IDEs already have an integrated debugging environment (Eclipse has one, but don't know which IDE you are using).

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. These files should be zipped up into a file named *BufMgr-<FirstName-LastName>.zip* (using the command *zip –r BufMgr-First-Last.zip ~/BufMgr*) or *BufMgr-<FirstName-LastName>.tgz* (using the command *tar -cvzf BufMgr-First-Last.tgz ~/BufMgr*). These files should be organized in the same directory structure as supplied. **Upload your solution (ie zip/tgz file) via the course website.** In addition to this submission, you will have to present your solution to the demonstrator in one of the sessions.

Deadline: End of your practical session in Week 6. Good Luck!