Lab 2: Buffer Pool 1

Instructor: Beom Heyn Kim

beomheynkim@hanyang.ac.kr

Department of Computer Science

Outline

- Project Overview
- Assignment
- Appendix: Background on Buffer Pool

Submission policy

- Grace Period
  - All students have **1 day** of grace period through the semester.
  - When you submit late, the exceeded time will be subtracted from your grace period in the unit of 1 day instead of direct penalty.
  - If you late over the grace period, we will **not** score your submission.
  - Example
    - 1 day of late submission: No penalty
    - 2 days of late submission: Penalty (No score)
    - Two late submissions, each one day late: Penalty (No score for later one)
- Preparation for the rest of projects
  - If you have any troubles implementing the first project, you must ask us for help. Because all projects are connected.

Project Overview

- This programming project is to implement a **buffer pool** in your storage manager.
  - The buffer pool is responsible for moving physical pages, which we called **blocks** in the lecture, back and forth from main memory to disk.
  - It allows a DBMS to support databases that are larger than the amount of memory that is available to the system.
  - The buffer pool's operations are transparent to other parts in the system.
    - For example, the system asks the buffer pool for a page using its unique identifier (page_id_t) and it does not know whether that page is already in memory or whether the system has to go retrieve it from disk.
  - **Your implementation will need to be thread-safe**.
    - Multiple threads will be accessing the internal data structures at the same time and thus you need to make sure that their critical sections are protected with latches (these are called "locks" in operating systems).

Project Overview (Cont.)

- You will need to implement the following two components in your storage manager:
  - **Least Recently Used (LRU) Replacement Policy**
  - **Buffer Pool Manager**
- We are providing you with stub classes that contain the API that you need to implement.
  - You **should not** modify the signatures for the pre-defined functions in these classes.
    - If you modify the signatures, the test code that we use for grading will break and you will get no credit for the project.
  - You also **should not** add additional classes in the source code for aforementioned components.
    - These components should be entirely self-contained.

Project Overview (Cont.)

- If a class already contains data members, you should **not** remove them.
    - For example, the BufferPoolManager contains DiskManager and Replacer objects.
        - These are required to implement the functionality that is needed by the rest of the system.
        - On the other hand, you may need to add data members to these classes in order to correctly implement the required functionality.
        - You can also add additional helper functions to these classes. The choice is yours.

Project Overview (Cont.)

- You are allowed to use any built-in C++17 containers in your project unless specified otherwise.
  - It is up to you to decide which ones you want to use.
  - Note that these containers are not thread-safe and that you will need to include latches in your implementation to protect them.
  - You may not bring in additional third-party dependencies (e.g. boost).

Outline

- Project Overview
- Assignment
- Appendix: Background on Buffer Pool

Assignment: LRU Replacement Policy

- The LRU Replacement Policy component is responsible for tracking page usage in the buffer pool.
    - You will implement a new sub-class called LRUReplacer in src/include/buffer/lru_replacer.h and its corresponding implementation file in src/buffer/lru_replacer.cpp.
    - LRUReplacer extends the abstract Replacer class (src/include/buffer/replacer.h), which contains the function specifications.
- The size of the LRUReplacer is the same as buffer pool since it contains placeholders for all of the frames in the BufferPoolManager.
    - However, not all the frames are considered as in the LRUReplacer. The LRUReplacer is initialized to have no frame in it.
    - Then, only the newly unpinned ones will be considered in the LRUReplacer.

Assignment: LRU Replacement Policy (Cont.)

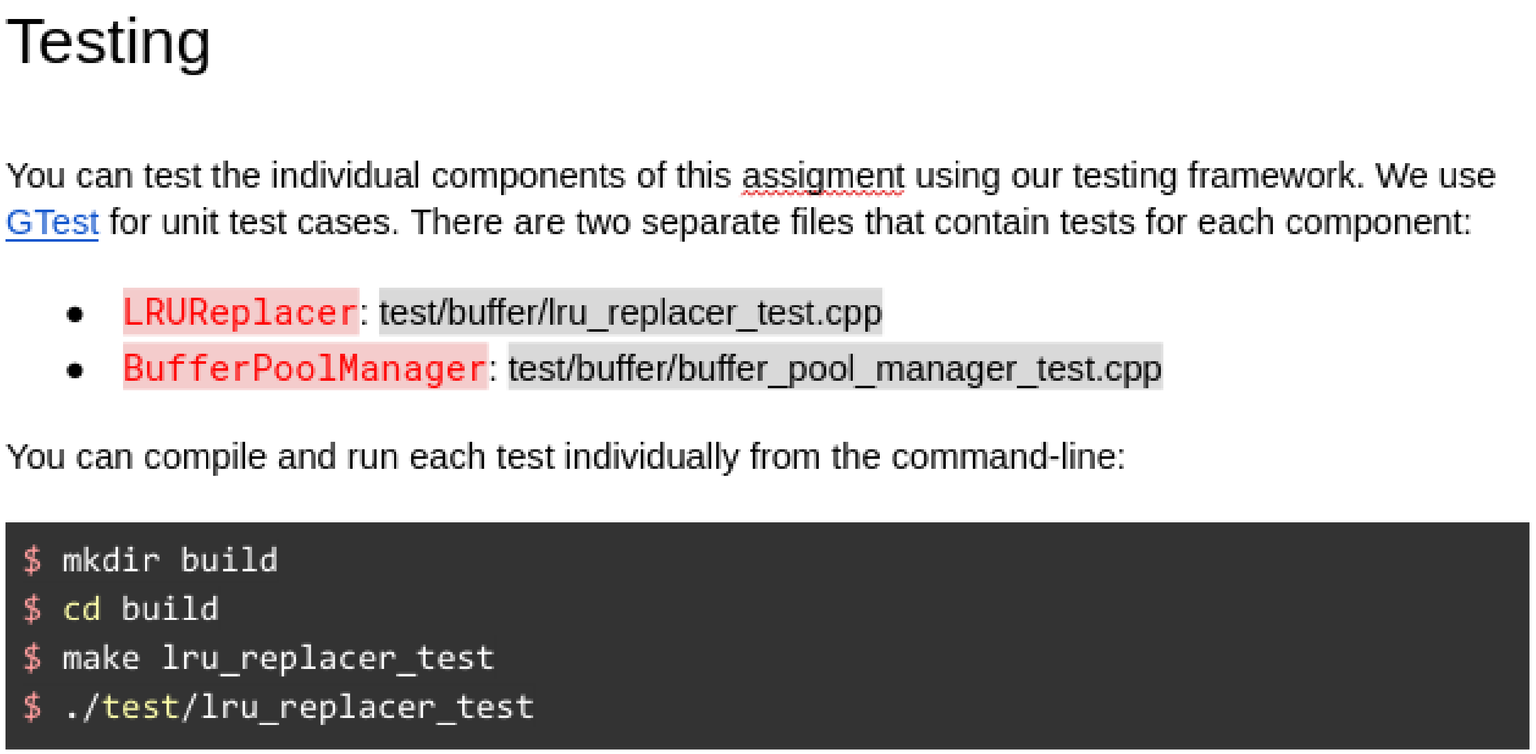
- You will need to implement the approximation of *LRU* policy. You will need to implement the following methods:
  - Victim(T*) : Remove the object that was accessed the least recently compared to all the elements being tracked by the Replacer, store its contents in the output parameter and return True. If the Replacer is empty, return False.
  - Pin(T) : This method should be called after a page is pinned to a frame in the BufferPoolManager. It should remove the frame containing the pinned page from the LRUReplacer.
  - Unpin(T) : This method should be called when the pin_count of a page becomes 0. This method should add the frame containing the unpinned page to the LRUReplacer.
  - Size() : This method returns the number of frames that are currently in the LRUReplacer.

Assignment: LRU Replacement Policy (Cont.)

- The implementation details are up to you.
  - You are allowed to use either list or vector among built-in STL containers. (Do not use other types of STL containers)
  - You can assume that you will not run out of memory, but you must make sure that the operations are thread-safe.
  - To protect critical sections to make operations thread-safe, you can use mutex.

- **Hint:** The basic idea is that you maintain a container containing the collection of frames that have been unpinned. Think about how you may find the oldest frame that has been unpinned among all collections of unpinned frames. (It can be easier than CLOCK approximation of LRU)

Assignment: LRU Replacement Policy (Cont.)



Testing

You can test the individual components of this assigment using our testing framework. We use GTest for unit test cases. There are two separate files that contain tests for each component:

- `LRUReplacer`: test/buffer/lru_replacer_test.cpp
- `BufferPoolManager`: test/buffer/buffer_pool_manager_test.cpp

You can compile and run each test individually from the command-line:

```
$ mkdir build
$ cd build
$ make lru_replacer_test
$ ./test/lru_replacer_test
```

Outline

- Project Overview
- Assignment
- Appendix: Background on Buffer Pool

Appendix: Background on Buffer Pool

Introduction to Buffer Pool

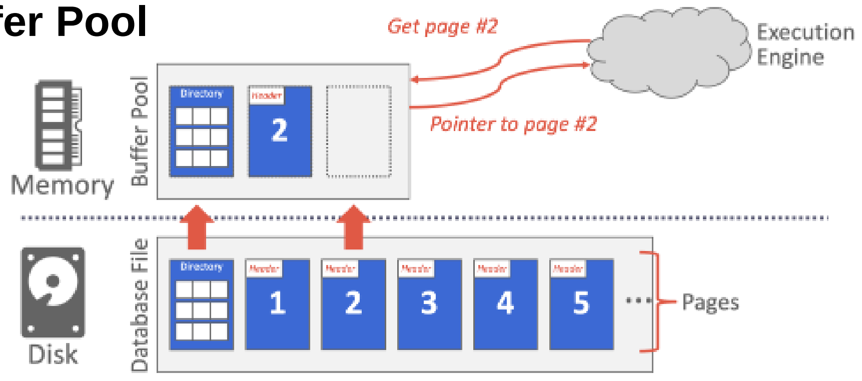


Figure 1: Disk-oriented DBMS.

The DBMS is responsible for managing its memory and moving data back-and-forth from the disk. Since, for the most part, data cannot be directly operated on in the disk, any database must be able to efficiently move data represented as files on its disk into memory so that it can be used. A diagram of this interaction is shown in Figure 1. An obstacle that DBMS faces is the problem of minimizing the slowdown of moving data around. Ideally, it should "appear" as if the data is all in the memory already. The execution engine shouldn't have to worry about how data is fetched into memory. Instead, Buffer Pool takes care of this data movement between memory and disk.

Appendix: Background on Buffer Pool (Cont.)

Buffer Pool

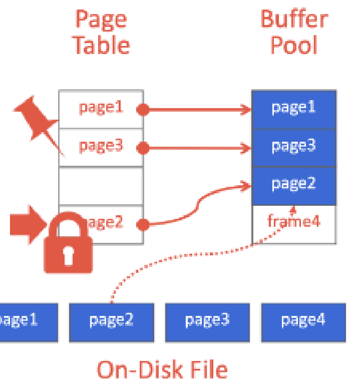


Figure 2: Buffer pool organization and meta-data

The buffer pool is an in-memory cache of pages read from disk. It is essentially a large memory region allocated inside of the database to store pages that are fetched from disk.

The buffer pool's region of memory is organized as an array of fixed size pages. Each array entry is called a frame. When the DBMS requests a page, an exact copy is placed into one of the frames of the buffer pool. Then, the database system can search the buffer pool first when a page is requested. If the page is not found, then the system fetches a copy of the page from the disk. Dirty pages are buffered and not written back immediately. See Figure 2 for a diagram of the buffer pool's memory organization.

Appendix: Background on Buffer Pool (Cont.)

Buffer Pool Meta-data

The buffer pool must maintain certain meta-data in order to be used efficiently and correctly.

Firstly, the *page table* is an in-memory hash table that keeps track of pages that are currently in memory. It maps page ids to frame locations in the buffer pool. Since the order of pages in the buffer pool does not necessarily reflect the order on the disk, this extra indirection layer allows for the identification of page locations in the pool.

Note: The page table is not to be confused with the *page directory*, which is the mapping from page ids to page locations in database files. All changes to the page directory must be recorded on disk to allow the DBMS to find on restart.

The page table also maintains additional meta-data per page, a dirty-flag and a pin/reference counter.

The *dirty-flag* is set by a thread whenever it modifies a page. This indicates to the storage manager that the page must be written back to disk.

The *pin/reference counter* tracks the number of threads that are currently accessing that page (either reading or modifying it). A thread has to increment the counter before they access the page. If a page’s count is greater than zero, then the storage manager is not allowed to evict that page from memory.

Appendix: Background on Buffer Pool (Cont.)

Buffer Replacement Policy

When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool. A replacement policy is an algorithm that the DBMS implements that makes a decision on which pages to evict from the buffer pool when it needs space. Implementation goals of replacement policies are improved correctness, accuracy, speed, and meta-data overhead.

The Least Recently Used replacement policy maintains a timestamp of when each page was last accessed. The DBMS picks to evict the page with the oldest timestamp. This timestamp can be stored in a separate data structure, such as a queue, to allow for sorting and improve efficiency by reducing sort time on eviction.

The CLOCK policy is an approximation of LRU without needing a separate timestamp per page. In the CLOCK policy, each page is given a reference bit. When a page is accessed, set to 1. To visualize this, organize the pages in a circular buffer with a "clock hand". Upon sweeping, check if a page's bit is set to 1. If yes, set to zero, if no, then evict it. In this way, the clock hand remembers position between evictions.

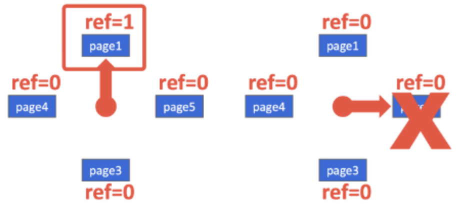


Figure 3: Visualization of CLOCK replacement policy. Page 1 is referenced and set to 1. When the clock hand sweeps, it sets the reference bit for page 1 to 0 and evicts page 5.

The End