



PROJECT #1 - BUFFER POOL

 **Do not post your project on a public Github repository.**

OVERVIEW

During the semester, you will be building a new disk-oriented storage manager for the BusTub DBMS. Such a storage manager assumes that the primary storage location of the database is on disk.

The first programming project is to implement a **buffer pool** in your storage manager. The buffer pool is responsible for moving physical pages 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).

You will need to implement the following two components in your storage manager:

- LRU Replacement Policy
- Buffer Pool Manager Instance
- Parallel Buffer Pool Manager

This is a single-person project that will be completed individually (i.e. no groups).

Release Date: Sep 13, 2021

Due Date: Sep 26, 2021 @ 11:59pm

PROJECT SPECIFICATION

For each of the following components, 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 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 these components. These components should be entirely self-contained.

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.

implementation to protect them. You may not bring in additional third-party dependencies (e.g. boost).

TASK #1 - LRU REPLACEMENT POLICY

This 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 maximum number of pages for the `LRUReplacer` is the same as the size of the buffer pool since it contains placeholders for all of the frames in the `BufferPoolManager`. However, at any given moment not all the frames are considered to be in the `LRUReplacer`. The `LRUReplacer` is initialized to have no frames in it. Then, only the newly unpinned ones will be considered to be in the `LRUReplacer`.

You will need to implement the LRU policy discussed in the class. You will need to implement the following methods:

- `Victim(frame_id_t*)` : Remove the object that was accessed least recently compared to all the other elements being tracked by the `Replacer`, store its contents in the output parameter and return `True`. If the `Replacer` is empty return `False`.
- `Pin(frame_id_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(frame_id_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`.

The implementation details are up to you. You are allowed to use built-in STL containers. You can assume that you will not run out of memory, but you must make sure that the operations are thread-safe.

Lastly, in this project you are expected to implement only the LRU replacement policy. You don't have to implement the clock replacement policy, even if there is a corresponding file for it.

TASK #2 - BUFFER POOL MANAGER INSTANCE

Next, you need to implement the buffer pool manager in your system (`BufferPoolManagerInstance`). The `BufferPoolManagerInstance` is responsible for fetching database pages from the `DiskManager` and storing them in memory. The `BufferPoolManagerInstance` can also write dirty pages out to disk when it is either explicitly instructed to do so or when it needs to evict a page to make space for a new page.

To make sure that your implementation works correctly with the rest of the system, we will provide you with some of the functions already filled in. You will also not need to implement the code that actually reads and writes data to disk (this is the `DiskManager` in our implementation). We will provide that functionality for you.

All in-memory pages in the system are represented by `Page` objects. The `BufferPoolManagerInstance` does not need to understand the contents of these pages. But it is important for you as the system developer to understand that `Page` objects are just containers for memory in the buffer pool and thus are not specific to a unique page. That is, each `Page` object contains a block of memory that the `DiskManager` will use as a location to copy the contents of a physical page that it reads from disk. The `BufferPoolManagerInstance` will reuse the same `Page` object to store data as it moves back and forth to disk. This means that the same `Page` object may contain a different physical page throughout the life of the system. The `Page` object's identifier

Each **Page** object also maintains a counter for the number of threads that have "pinned" that page. Your **BufferPoolManagerInstance** is not allowed to free a **Page** that is pinned. Each **Page** object also keeps track of whether it is dirty or not. It is your job to record whether a page was modified before it is unpinned. Your **BufferPoolManagerInstance** must write the contents of a dirty **Page** back to disk before that object can be reused.

Your **BufferPoolManagerInstance** implementation will use the **LRUReplacer** class that you created in the previous steps of this assignment. It will use the **LRUReplacer** to keep track of when **Page** objects are accessed so that it can decide which one to evict when it must free a frame to make room for copying a new physical page from disk.

You will need to implement the following functions defined in the header file (`src/include/buffer/buffer_pool_manager_instance.h`) in the source file (`src/buffer/buffer_pool_manager_instance.cpp`):

- **FetchPgImp**(page_id)
- **UnpinPgImp**(page_id, is_dirty)
- **FlushPgImp**(page_id)
- **NewPgImp**(page_id)
- **DeletePgImp**(page_id)
- **FlushAllPagesImpl**()

For **FetchPgImp**, you should return NULL if no page is available in the free list and all other pages are currently pinned. **FlushPgImp** should flush a page regardless of its pin status.

For **UnpinPgImp**, the `is_dirty` parameter keeps track of whether a page was modified while it was pinned.

Refer to the function documentation for details on how to implement these functions. Don't touch the non-impl versions, you need those to grade your code.

Note: **Pin** and **Unpin** within the contexts of the **LRUReplacer** and the **BufferPoolManagerInstance** have inverse meanings. Within the context of the **LRUReplacer**, pinning a page implies that we shouldn't evict the page because it is in use. This means we should remove it from the **LRUReplacer**. On the other hand, pinning a page in the **BufferPoolManagerInstance** implies we want to use a page, and that it should not be removed from the buffer pool.

TASK #3 - PARALLEL BUFFER POOL MANAGER

As you probably noticed in the previous task, the single Buffer Pool Manager Instance needs to take latches in order to be thread safe. This can cause a lot of contention as every thread fights over a single latch when interacting with the buffer pool. One potential solution is to have multiple buffer pools in your system, each with its own latch.

ParallelBufferPoolManager is a class that holds multiple **BufferPoolManagerInstance**s. For every operation the **ParallelBufferPoolManager** picks a single **BufferPoolManagerInstance** and delegates to that instance.

We use the given page id to determine which specific **BufferPoolManagerInstance** to use. If we have `num_instances` many **BufferPoolManagerInstance**s, then we need some way to map the given page id to a number in the range `[0, num_instances)`. For this project we will be using the modulo operator, `page_id mod num_instances` will map the given `page_id` to the correct range.

When the **ParallelBufferPoolManager** is first instantiated it should have a starting index of 0. Every time you create a new page you will try every **BufferPoolManagerInstance**, starting at the starting index, until one is successful. Then increase the starting index by one.

You will need to implement the following functions defined in the header file (src/include/buffer/parallel_buffer_pool_manager.h) in the source file (src/buffer/parallel_buffer_pool_manager.

- `ParallelBufferPoolManager(num_instances, pool_size, disk_manager, log_manager)`
- `~ParallelBufferPoolManager()`
- `GetPoolSize()`
- `GetBufferPoolManager(page_id)`
- `FetchPgImp(page_id)`
- `UnpinPgImp(page_id, is_dirty)`
- `FlushPgImp(page_id)`
- `NewPgImp(page_id)`
- `DeletePgImp(page_id)`
- `FlushAllPagesImpl()`

INSTRUCTIONS

See the [Project #0 instructions](#) on how to create your private repository and setup your development environment.

TESTING

You can test the individual components of this assignment using our testing framework. We use [GTest](#) for unit test cases. There are three separate files that contain tests for each component:

- `LRUReplacer`: test/buffer/lru_replacer_test.cpp
- `BufferPoolManagerInstance`: test/buffer/buffer_pool_manager_instance_test.cpp
- `ParallelBufferPoolManager`: test/buffer/parallel_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
```

You can also run `make check-tests` to run ALL of the test cases. Note that some tests are disabled as you have not implemented future projects. You can disable tests in GTest by adding a `DISABLED_` prefix to the test name.

Important: These tests are only a subset of the all the tests that we will use to evaluate and grade your project. You should write additional test cases on your own to check the complete functionality of your implementation.

FORMATTING

Your code must follow the [Google C++ Style Guide](#). We use [Clang](#) to automatically check the quality of your source code. Your project grade will be [zero](#) if your submission fails any of these checks.

Execute the following commands to check your syntax. The `format` target will automatically correct your code. The `check-lint` and `check-clang-tidy` targets will print errors and instruct you how to fix it to conform to our style guide.

```
$ make format
$ make check-lint
$ make check-clang-tidy
```

Instead of using `printf` statements for debugging, use the `LOG_*` macros for logging information like this:

```
LOG_INFO("# Pages: %d", num_pages);  
LOG_DEBUG("Fetching page %d", page_id);
```

To enable logging in your project, you will need to reconfigure it like this:

```
$ mkdir build  
$ cd build  
$ cmake -DCMAKE_BUILD_TYPE=DEBUG ..  
$ make
```

The different logging levels are defined in `src/include/common/logger.h`. After enabling logging, the logging level defaults to `LOG_LEVEL_INFO`. Any logging method with a level that is equal to or higher than `LOG_LEVEL_INFO` (e.g., `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`) will emit logging information. Note that you will need to add `#include "common/logger.h"` to any file that you want to use the logging infrastructure.

We encourage you to use `gdb` to debug your project if you are having problems.

 Post all of your questions about this project on Piazza. Do not email the TAs directly with questions.

If you are having compilation problems, running `make clean` does not completely reset the compilation process. You will need to delete your build directory and run `cmake ..` again before you rerun `make`.

GRADING RUBRIC

Each project submission will be graded based on the following criteria:

1. Does the submission successfully execute all of the test cases and produce the correct answer?
2. Does the submission execute without any memory leaks?
3. Does the submission follow the code formatting and style policies?

Note that we will use additional test cases to grade your submission that are more complex than the sample test cases that we provide you.

LATE POLICY

See the [late policy](#) in the syllabus.

SUBMISSION

After completing the assignment, you can submit your implementation to Gradescope:

<https://www.gradescope.com/courses/286490/>

You only need to include the following files:

- `src/include/buffer/lru_replacer.h`
- `src/buffer/lru_replacer.cpp`
- `src/include/buffer/buffer_pool_manager_instance.h`
- `src/buffer/buffer_pool_manager_instance.cpp`
- `src/include/buffer/parallel_buffer_pool_manager.h`
- `src/buffer/parallel_buffer_pool_manager.cpp`

run the bash file to make things easier for you.

```
$ zip project1-submission.zip \  
  src/include/buffer/lru_replacer.h \  
  src/buffer/lru_replacer.cpp \  
  src/include/buffer/buffer_pool_manager_instance.h \  
  src/buffer/buffer_pool_manager_instance.cpp \  
  src/include/buffer/parallel_buffer_pool_manager.h \  
  src/buffer/parallel_buffer_pool_manager.cpp
```

You can submit your answers as many times as you like and get immediate feedback.

NOTES ON GRADESCOPE AND AUTOGRADER

If you are timing out on Gradescope, it's likely because you have a deadlock in your code or your code is too slow and does not run in 60 seconds. If your code is too slow it may be because your **LRUReplacer** is not efficient enough.

The autograder will not work if you do not remove logs before submissions.

If the autograder did not work properly, make sure that your formatting commands work and that you are submitting the right files.

 CMU students should use the Gradescope course code announced on Piazza.


COLLABORATION POLICY

Every student has to work individually on this assignment.

Students are allowed to discuss high-level details about the project with others.

Students are **not** allowed to copy the contents of a white-board after a group meeting with other students.

Students are **not** allowed to copy the solutions from another colleague.

 **WARNING:** All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. Plagiarism **will not** be tolerated. See CMU's **Policy on Academic Integrity** for additional information.

Last Updated: Jan 07,

