Advanced Database Organization - Summer 2023
CS 525 - All Sections
Programming Assignment II: Buffer Manager
Due: Friday, September 22nd 2023 by 23h59

---

1. Task

The goal of this assignment is to implement a simple buffer manager. The buffer manager manages a fixed number of pages in memory that represent pages from a page file managed by the storage manager implemented in `assignment_1`. The memory pages managed by the buffer manager are called `page frames` or `frames` for short. We call the combination of a page file and the page frames storing pages from that file a `Buffer Pool`.
The buffer manager should be able to handle more than one open buffer pool at the same time. However, there can only be one buffer pool for each page file. Each buffer pool uses one page replacement strategy that is determined when the buffer pool is initialized. You should at least implement two replacement strategies `FIFO` and `LRU`. Your solution should implement all the methods defined in the `buffer_mgr.h` header explained below.
Make use of existing debugging and memory checking tools. At some point you will have to debug an error. See the main assignment page (Programming Assignment: Organization) for information about debugging. Memory leaks are errors!

2. Buffer Pool Functionality and Concepts

A buffer pool consists of a fixed amount of page frames (pages in memory) that are used to store disk pages from a page file in memory. Clients of the buffer manager can request pages identified by their position in the page file (`page number`) to be loaded in a page frame. This is called `pinning` a page.

Internally, the buffer manager has to check whether the page requested by the client is already cached in a page frame. If this is the case, then the buffer simply returns a pointer to this page frame to the client. Otherwise, the buffer manager has to read this page from disk and decide in which page frame it should be stored (this is what the replacement strategy is for). Once an appropriate frame is found and the page has been loaded, the buffer manager returns a pointer to this frame to the client. The client can then start to read and/or modify that page.

Once the client is done with reading or writing a page, he needs to inform the buffer manager that he no longer needs that page. This is called `unpinning`. Furthermore, the buffer manager needs to know whether the page was modified by the client. This is realized by requiring the client to call a function to tell the buffer manager that the page is `dirty`. The buffer needs this information for replacing pages in the buffer pool. If a dirty page is evicted from the buffer pool, then the buffer manager needs to write the content of this page back to disk. Otherwise, the modifications done by the client would be lost.

Since buffer pools are used concurrently by several components of a DBMS, the same page can be pinned by more than one client. **Making the functions of the buffer manager thread-safe is not part of the assignment**.

The number of clients having pinned a page is called the `fix count` of that page. The buffer manager can only evict pages with `fix count 0` from the pool, because a non-zero fix count indicates that at least one client is still using the page. Pinning a page increases its fix count by `1`, unpinning the page reduces its fix count.

2.1. Some hints and reminders.

- Independent of the page replacement strategy, the buffer manager is only allowed to evict pages with fix count zero. This has to be taken into account when implementing page replacement strategies.
- Dirty pages can be evicted from the pool if they have a fix count `0`, but have to be written back to disk before the eviction
- If a dirty page is written back to disk and has fix count `0`, then it is no longer considered dirty.
- You buffer manager needs to maintain a mapping between page numbers and page frames to enable fast look-ups from page number to page frame and vice versa.

2.2. Optional Extensions.

Realize these optional extensions for extra credit (`5%`) and extra fun 🙂

- Make the buffer pool functions thread safe. This extension would result in your buffer manager being closer to real life buffer manager implementations.
- Implement additional page replacement strategies such as `CLOCK` or `LRU-k`.

3. Interface

The header for the buffer manager interface is shown below. Your solution should implement all functions defined in this header.

`buffer_mgr.h`

```
01 | #ifndef BUFFER_MANAGER_H
02 | #define BUFFER_MANAGER_H
03 |
04 | // Include return codes and methods for logging errors
05 | #include "dberror.h"
06 |
07 | // Include bool DT
08 | #include "dt.h"
09 |
10 | // Replacement Strategies
11 | typedef enum ReplacementStrategy {
12 |   RS_FIFO = 0,
13 |   RS_LRU = 1,
14 |   RS_CLOCK = 2,
15 |   RS_LFU = 3,
16 |   RS_LRU_K = 4
17 | } ReplacementStrategy;
18 |
19 | // Data Types and Structures
20 | typedef int PageNumber;
21 | #define NO_PAGE -1
22 |
23 | typedef struct BM_BufferPool {
24 |   char *pageFile;
25 |   int numPages;
26 |   ReplacementStrategy strategy;
27 |   void *mgmtData; // use this one to store the bookkeeping info your buffer
28 |                   // manager needs for a buffer pool
29 | } BM_BufferPool;
30 |
31 | typedef struct BM_PageHandle {
32 |   PageNumber pageNum;
33 |   char *data;
34 | } BM_PageHandle;
35 |
36 | // convenience macros
37 | #define MAKE_POOL()                                    \
38 |   ((BM_BufferPool *) malloc (sizeof(BM_BufferPool)))
39 |
40 | #define MAKE_PAGE_HANDLE()                             \
41 |   ((BM_PageHandle *) malloc (sizeof(BM_PageHandle)))
42 |
43 | // Buffer Manager Interface Pool Handling
44 | RC initBufferPool(BM_BufferPool *const bm, const char *const pageFileName,
45 |                   const int numPages, ReplacementStrategy strategy,
46 |                   void *stratData);
47 | RC shutdownBufferPool(BM_BufferPool *const bm);
48 | RC forceFlushPool(BM_BufferPool *const bm);
49 |
50 | // Buffer Manager Interface Access Pages
51 | RC markDirty (BM_BufferPool *const bm, BM_PageHandle *const page);
52 | RC unpinPage (BM_BufferPool *const bm, BM_PageHandle *const page);
53 | RC forcePage (BM_BufferPool *const bm, BM_PageHandle *const page);
54 | RC pinPage (BM_BufferPool *const bm, BM_PageHandle *const page,
55 |             const PageNumber pageNum);
56 |
57 | // Statistics Interface
58 | PageNumber *getFrameContents (BM_BufferPool *const bm);
59 | bool *getDirtyFlags (BM_BufferPool *const bm);
60 | int *getFixCounts (BM_BufferPool *const bm);
61 | int getNumReadIO (BM_BufferPool *const bm);
62 | int getNumWriteIO (BM_BufferPool *const bm);
63 | #endif
```

3.1. **Data structures.** The header defines two important data structures. The `BM_BufferPool` and the `BM_PageHandle`.

- The `BM_BufferPool` stores information about a buffer pool: the name of the page file associated with the buffer pool (`pageFile`), the size of the buffer pool, i.e., the number of page frames (`numPages`), the page replacement strategy (`strategy`), and a pointer to bookkeeping data (`mgmtData`). Similar to the first assignment, you can use the `mgmtData` to store any necessary information about a buffer pool that you need to implement the interface. For example, this could include a pointer to the area in memory that stores the page frames or data structures needed by the page replacement strategy to make replacement decisions.

```
typedef struct BM_BufferPool {
  char *pageFile;
  int numPages;
  ReplacementStrategy strategy;
  void *mgmtData;
} BM_BufferPool;
```

- The `BM_PageHandle` stores information about a page. The page number (position of the page in the page file) is stored in `pageNum`. The page number of the first data page in a page file is `0`. The data field points to the area in memory storing the content of the page. This will usually be a page frame from your buffer pool.

```
typedef struct BM_PageHandle {
  PageNumber pageNum;
  char *data;
} BM_PageHandle;
```

3.2. **Buffer Pool Functions.** These functions are used to create a buffer pool for an existing page file (`initBufferPool`), shutdown a buffer pool and free up all associated resources (`shutdownBufferPool`), and to force the buffer manager to write all dirty pages to disk (`forceFlushPool`).

- `initBufferPool` creates a new buffer pool with `numPages` page frames using the page replacement strategy `strategy`. The pool is used to cache pages from the page file with name `pageFileName`. Initially, all page frames should be empty. The page file should already exist, i.e., this method should not generate a new page file. `stratData` can be used to pass parameters for the page replacement strategy. For example, for `LRU-k` this could be the parameter `k`.
- `shutdownBufferPool` destroys a buffer pool. This method should free up all resources associated with buffer pool. For example, it should free the memory allocated for page frames. If the buffer pool contains any dirty pages, then these pages should be written back to disk before destroying the pool. It is an error to shutdown a buffer pool that has pinned pages.
- `forceFlushPool` causes all dirty pages (with fix count `0`) from the buffer pool to be written to disk.

3.3. **Page Management Functions.** These functions are used pin pages, unpin pages, mark pages as dirty, and force a page back to disk.

- `pinPage` pins the page with page number `pageNum`. The buffer manager is responsible to set the pageNum field of the page handle passed to the method. Similarly, the `data` field should point to the page frame the page is stored in (the area in memory storing the content of the page).
- `unpinPage` unpins the page `page`. The `pageNum` field of `page` should be used to figure out which page to unpin.
- `markDirty` marks a page as dirty.
- `forcePage` should write the current content of the page back to the page file on disk.

3.4. **Statistics Functions.** These functions return statistics about a buffer pool and its contents. The print debug functions explained below internally use these functions to gather information about a pool.

- The `getFrameContents` function returns an array of `PageNumbers` (of size `numPages`) where the `i`th element is the number of the page stored in the `i`th page frame. An empty page frame is represented using the constant `NO_PAGE`.
- The `getDirtyFlags` function returns an array of `bools` (of size `numPages`) where the `i`th element is `TRUE` if the page stored in the `i`th page frame is dirty. Empty page frames are considered as clean.
- The `getFixCounts` function returns an array of `ints` (of size `numPages`) where the `i`th element is the fix count of the page stored in the `i`th page frame. Return `0` for empty page frames.
- The `getNumReadIO` function returns the number of pages that have been read from disk since a buffer pool has been initialized. You code is responsible to initializing this statistic at pool creating time and update whenever a page is read from the page file into a page frame.
- `getNumWriteIO` returns the number of pages written to the page file since the buffer pool has been initialized.

4. Error handling and Printing buffer and page content

The initial `assign2` folder contains code implementing several helper functions.

4.1. `buffer_mgr_stat.h` **and** `buffer_mgr_stat.c`**.** The `buffer_mgr_stat.h` provides several functions for outputting buffer or page content to `stdout` or into a string. The implementation of these functions is provided so you do not have to implement them yourself.

- `printPageContent` prints the byte content of a memory page.
- `printPoolContent` prints a summary of the current content of a buffer pool. The format looks like that:

```
{FIFO 3}: [0 0],[3x5],[2 1]
```

  - `FIFO` is the page replacement strategy. The number following the strategy is the size of the buffer pool (number of page frames). Each part enclosed in `[]` represents one buffer frame. The first number is the page number for the page that is currently stored in this buffer frame. The "`x`" indicates that the page is `dirty`, i.e., it has to be written back to disk before it can be replaced with another page. The last number is the `fix count`.
  - For example, in the buffer shown above the first frame stores the disk page `0` with a fix count of `0`. The second page frame stores the disk page `3` with a fix count of `5` and this page is dirty.

4.2. `dberror.h` **and** `dberror.c`**.** The `dberror.h` header defines error codes and provides a function to print an error message to `stdout`.

5. Source Code Structure

You source code directories should be structured as follows. You should reuse your existing storage manager implementation. So before you start to develop, please copy your storage manager implementation from `assign1` to `assign2`.

- Put all source files in a folder `assign2` in your git repository
- This folder should contain at least
  - the provided header and `C` files
  - a make file for building your code `Makefile`.
  - a bunch of `*.c` and `*.h` files implementing the buffer manager
  - `README.txt/README.md`: A markdown or text file with a brief description of your solution

  Example, the structure may look like that:

```
git
   assign2
       README.md
       Makefile
       buffer_mgr.h
       buffer_mgr_stat.c
       buffer_mgr_stat.h
       dberror.c
       dberror.h
       dt.h
       storage_mgr.h
       test_assign2_1.c
       test_assign2_2.c
       test_helper.h
```

6. Test cases

The `test_assign2_1.c` file implements several test cases using the `buffer_mgr.h` interface using the `FIFO` strategy. Please let your make file generate a `test_assign2_1` binary for this code. This test also tests the `LRU` strategy. You are encouraged to extend it with new test cases or use it as a template to develop your own test files.