CS349: Buffer Manager Assignment
Scott D. Anderson

In this assignment, you will implement a simplified version of the Buffer Manager layer, without support for concurrency control or recovery. You will be given the code for the lower layer, the Disk Space Manager.

For this assignment, you may work with another student if you would like. Please inform me of your teammate as soon as you've formed a team.

You should begin by reading the chapter on Disks and Files, to get an overview of buffer management. This material will also be covered in class.

1 The Buffer Manager Interface

The simplified Buffer Manager interface that you will implement in this assignment allows a client (a higher level program that calls the Buffer Manager) to allocate/de-allocate pages on disk, to bring a disk page into the buffer pool and pin it, and to unpin a page in the buffer pool.

The methods that you have to implement are described in the `bufman.h` interface file. You should not need to change this file, and if you do, you should leave the "public" interface unchanged. You may do anything to the private section that you want.

2 Internal Design

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 of Page objects. In my implementation, I called this array "frames." Feel free to rename private data and classes as you see fit.

In addition, you should maintain an array of *descriptors*, one per frame. Each descriptor has the following fields:

- `page_number`, which is of type `PageId_t` (an unsigned integer).
- `pin_count`, which is an `int`.
- `dirtybit`, which is `bool`.

This object describes the page that is stored in the corresponding frame. A page is identified by a *page number* that is generated by the DBF class when the page is allocated, and is unique over all pages in the database. The `PageId_t` type is defined as an unsigned integer type in `global_defs.h`.

A simple *hash table* should be used to figure out what frame a given disk page occupies. The hash table should be implemented (entirely in main memory) by using an array of pointers to lists of $\langle$ *page number, frame number* $\rangle$ pairs. The array is called the *directory* and each list of pairs is called a *bucket*. Given a *page number*, you should apply a *hash function* to find the directory entry pointing to the bucket that contains the frame number for this page, if the page is 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. If you find such a pair, it will tell you the frame in which the page resides. This is illustrated in figure 1.

The hash function must distribute values in the domain of the search field uniformly over the collection of buckets. If we have N buckets, numbered 0 through N-1, a hash function h of the form

$$h(v) = (a * v + b) \bmod N$$

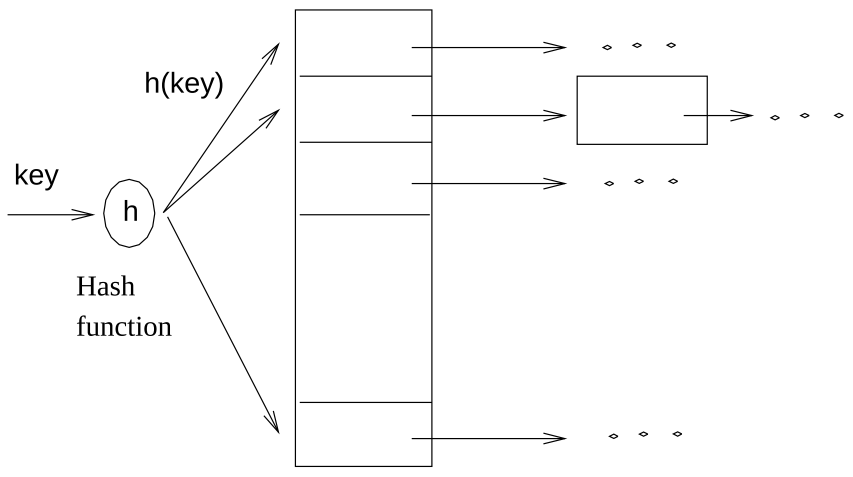

Figure 1: Hash Table

works well in practice.

When a page is requested the buffer manager should do the following:

1. Check the buffer pool (by using the hash table) to see if it contains the requested page. If the page is not in the pool, it should be brought in as follows:
   (a) Choose a frame for replacement, using the LRU replacement policy.
   (b) If the frame chosen for replacement is dirty, *flush* it (i.e., write out the page that it contains to disk, using the appropriate DB class method).
   (c) Read the requested page (again, by calling the DB class) into the frame chosen for replacement; the *pin_count* and *dirtybit* for the frame should be initialized to 0 and FALSE, respectively.
   (d) Delete the entry for the old page from the Buffer Manager's hash table and insert an entry for the new page. Also, update the entry for this frame in the bufDescr array to reflect these changes.
2. *Pin* the requested page by incrementing the *pin_count* in the descriptor for this frame. and return a pointer to the page to the requestor.

To implement the LRU replacement policy, maintain a queue of frame numbers. When a frame is to be chosen for replacement, you should pick the first frame in this list. A frame number is added to the end of the queue when the *pin_count* for the frame is decremented to 0. A frame number is removed from the queue if the *pin_count* becomes non-zero: this could happen if there is a request for the page currently in memory!

3 Where to Find Makefiles, Code, etc.

Please copy all the files from `~anderson/349/wdb` into your own local directory. Some of the files are:

- `Makefile`: A sample makefile to compile your project. You will have to set up any dependencies by editing this file. You can also design your own Makefile.
- `bufman.h`: Specifications for the class BufMgr. You have to implement these specifications as part of the assignment.
- `global_defs.h`: Specifications for error status codes and a few global types. You can add things to this file, but don't delete or modify anything. (Yes, you can add additional error status codes.)
- `page.h`: Defines the class `Page`, which is a basic disk block.
- `random_rw.cpp,h` implement the low-level random disk I/O functions.
- `dbf.cpp`: Implements the DBF object, which the buffer manager will use to do disk I/O. You won't have to call any low-level C routines yourself. `dbf.h` is the interface definition for this class and `dbf-test.cc` is a test program.
- `hashtable-test.cc` and `queue-test.cc` are test files for testing the queue and hashtables. I encourage you to test modules such as your hashtable and queue in this fashion, so that when you're coding your buffer manager, you can rely on your substrate. Feel free to modify and expand these files.
- `dbf-test.dbf` and `bufman-test.dbf` are databases built by those two test programs. You're welcome to crib ideas and code from `dbf-test.cc`; you'll have to implement your own `bufman-test.cc`.

4 Testing

You need not only to implement the solution, but you need to convince me that your code is indeed a solution. You will do this by writing a test program that puts your code through its paces. Think of everything you can, and test it.

You must document your testing, so that I understand it. You can't just turn in a bunch of code (I've not been a good example in this). You have to describe (in English) the various test cases and what the correct result is, and then show (with print statements in your code and so forth), that your code performs correctly.

You can either document your testing in the same file with your test program (`bufman-test.cc`), as a long comment, or write it in a separate file (`bufman-test.text`).

5 What to Turn In, and When

You should put your working code on Dudley and email email me its location, so that I can copy it to my own directory to run and read. The assignment is due by midnight on Wednesday, February 21st.

My solution will be made public soon after that time, so if you intend to turn your solution in late, please let me know. If I receive your code after I've made my solution public, I won't be able to give you credit.