CSCD 427 Project #1 (50 points)

Due: 11:59pm on April 19, 2017

PROJECT DESCRIPTION

Reading and writing pages from main memory to disk is an important task of a database system. Main memory is partitioned into collections of pages, and the collection of pages is called the *buffer pool*. The buffer pool is organized into *frames*, and each frame in the buffer pool can hold a page that is brought in from the disk. The *buffer manager* is responsible for bringing pages from disk to the buffer pool when they are needed, and writing pages back to the disk when they have been updated. The buffer keeps a *pin* count and *dirty* flag for each frame in the buffer pool. The pin count records the number of active users a page has, and the dirty flag records whether the page has been updated or not. As the buffer pool fills up, some pages may need to be removed in order to make room for new pages. The buffer manager uses a *replacement policy* to choose pages to be overwritten from the buffer pool. The strategy used can greatly affect the performance of the system. *LRU* (least recently used), *MRU* (most recently used) and *Clock* are different policies that appropriate to use under different conditions.

Your job is to implement a buffer manager simulator that can pin, unpin, read from, and write to disk pages in the buffer pool. Your buffer manager should support ***LRU*** replacement policy.

GETTING STARTED

You should at least implement the following three classes:

- **`BufHashTbl`** class which represents a page table implemented by the buffer manager. You should implement a separate chaining hash table which includes at least the following methods:
  - **`insert():`** insert a <frame#, page#> pair into the page table.
  - **`lookup():`** search for a page from the page table, and return the frame# if the page is found.
  - **`remove():`** remove a <frame#, page#> pair from the page table.
- **`Frame`** class which represents a single frame in the buffer pool. You should at least implement the following methods:
  - **`getPin()`**
  - **`incPin()`**
  - **`decPin()`**
  - **`isDirty()`**
  - **`setDirty()`**
  - **`displayPage():`** display the contents in the current frame.
  - **`updatePage():`** append the new contents to the current frame.
- **`BufMgr`** class which implements the buffer management protocol. You should at least implement the following methods:
  - **`pin():`** follow "I want Page X" protocol
  - **`unpin():`** follow "I'm done with Page X" protocol
  - **`createPage()`**: create a new page. Since you are not implementing a real database system, you may assume a page is an actual file on the disk. For example, if you want to create page[5] (assume index starts from 0), you should create a new file named 5.txt, and set its initial contents to the page index, i.e., "This is Page 5." (One note: in a real DBMS, creating a new page is implemented by the disk space manger instead of the buffer manager.)
  - **`readPage()`**: read from disk to buffer, i.e., read the corresponding .txt file.
  - **`writePage()`**: write from buffer to disk, i.e., overwrite the corresponding .txt file.
  - **`displayPage()`**: call the corresponding frame's `displayPage()` method.
  - **`updatePage()`**:call the corresponding frame's `updatePage()` method.

- **`BufMgrTester`** class which holds the **`main()`** method.

Your program will take one command line argument **`a_number`** which defines the size of the buffer pool.

```
% java BufMgrTester a_number
```

Your program should prompt the user to select from one of the five options:

1. Create pages: this creates a certain number of initial pages
2. Request a page: follow "I want Page X" protocol to pin the page and display the contents of the requested page to user. Note that this is the page obtained from the buffer pool, not the page obtained from the disk. In other words, it is possible that the contents in the buffer have been changed, but the system has not written the new contents back to the disk yet.
3. Update a page: first display the contents of the page; then allow the user to **append** new contents to the page. Note that this is the change made to the page in the buffer pool, not the change made to the disk. Assume update operation is issued by an existing active user, so the buffer manager doesn't need to update the pin count of the page.
4. Relinquish a page: follow "I'm done with Page X" protocol to unpin the page.

-1. Quit.

As an example, the following shows a sequence of operations and their effects:

Step	Command	Buffer	Disk	Display to user
1	java BufMgrTester 4	A 4-frame buffer pool is created		
2	1 (with parameter 6)		six .txt files (from 0.txt to 5.txt) are created	
3	2 (with parameter 3)	page[3] is loaded into **frame[0]** (frame index also starts from 0)		"This is Page 3."
4	2 (with parameter 1)	page[1] is loaded into **frame[1]**		"This is Page 1."
5	2 (with parameter 5)	page[5] is loaded into **frame[2]**		"This is Page 5."
6	3 (with parameter 3)	**frame[0]** now has: "This is Page 3. I changed Page 3." The dirty bit should be set to 1.	**3.txt still has:** "This is page 3."	First display "This is Page 3"; then ask user to enter the appended contents. Assume user entered "I changed Page 3."
7	2 (with parameter 0)	page[0] is loaded into **frame[3].**		"This is Page 0."
8	2 (with parameter 3)	frame[0] now has pin count of 2.	**3.txt still has:** "This is page 3."	"This is Page 3. I changed Page 3."
9	4 (with parameter 3)	frame[0] now has pin count of 1.	**3.txt still has:** "This is page 3."	
10	4 (with parameter 3)	frame[0] becomes free, but the contents of it are dirty.	**3.txt still has:**	

			"This is page 3."	
11	4 (with parameter 0)	frame[3] becomes free		
12	2 (with parameter 2)	page[2] is loaded into **frame[0] based on LRU**	**3.txt now has:** "This is Page 3. I changed Page 3."	"This is Page 2."
13	4 (with parameter 1)	frame[1] becomes free		
14	2 (with parameter 3)	page[3] is loaded into **frame[3] based on LRU.**		"This is Page 3. I changed Page 3."
15	-1 Quit			

SUBMISSION

You need to submit a single zip file as your solution via the Canvas system. In this zip file, you need to include:

- All the java source files you have created.
- An output file which shows the test result of running your program.
- A readme file which includes all the information you would like to share with the instructor and/or the grader. If there's nothing to share, this file can be omitted.

GRADING

- (10 points) Compilable program files
- (30 points) Program Correctness
- (10 points) Testing (results.jpg, results.pdf, etc.)