dam556-project1

Danny Jensen
danje14

March 2018

1 Intro

In this project a page hander needs to be implemented. For this, a number of functions in 3 files need to be filled.
From the file *BufMgr.java* the functions: *freepage*(), *pinPage*(), *unpinPage*(), *flushPage*(), *getNumBuffers*() and *getNumUnpinned*().
From *Clock.java* only the *newPage*() function is left as is because it is not used. The check this function would make is handled in *BufMgr.java*.
And one line of code is implemented in the *Replacer*() inside *Replacer.java*.

2 Overall Status

The overall status of the project is that it is working well without any errors when run with runBmTests. The only thing that fails is the test that should fail.

3 Implementation

The implementation of the single line in *Replacer.java* What this one line of code is doing is to initialize the frame in the replacer.

```java
protected Replacer(BufMgr bufmgr) {
    this.frametab = bufmgr.frametab;
}
```

Next lets look at *Clock.java*:

```java
protected Clock(BufMgr bufmgr) {
        super(bufmgr);
        for (FrameDesc f : frametab){
            f.state = freeState;
        }
    }
```

This for loop simply sets the state of all frames to free. it looks simple but is very important.

freePage(), *pinage*() and *unpinPage*() all does allmost the same thing.
freePage() takes a frame and sets the state to free, *pinage*() sets the state of the given frame to pinned, the only one that is a litte different is *unpinPage*() it has a check to see if the pincnt is 0, before it sets the state of the frame to unpinned.
The *pickVictim*() was a tough one. to make sure that the frame was not in use the for loop that looks for a frame to remove, is run through 2 times.

```java
for(int j=0;j<2;j++){
    for ( int i = 0 ; i < frametab.length; i++ ) {
```

the first time is to set the state of an unpinned frame to unpinned, and the next then checks if it is still unpinned at that runthrough, if it is it means the frame was not used and is not in use so it index of that frame can be returned. If no frame was found that could be freed an error is returned.

The bulk of the implementation was in the *BufMgr.java* file. *freePage()* works by first checking if the page actually is in the buffer, else it just stops. If the page is in the buffer but the pin count is negative something is wrong so an error is thrown. Else the page is removed.
unpinPage() checks if the page is in the buffer and if the pin count is 0 if one of these is true someting is wrong and an exeption is thrown. if the two conditions where not true it simply unpins the page.
flushPage() simply checks if the page is dirty, if it is the its written to the disk and the dirty status is set to false.
getNumBuffers() was very easy it just returns the length of the bufpool.
getNumUnpinned() runs through every frame in the frametab and isf the pin count is 0 a counter is incremented, and returned when the for loop is done.
pinage() was the toughest function to get to work. First off it checks if the page is pinned, and if there is something wrong with the way it is pinned if there is it throws an exception.

```java
if (pageInfo != null ) {
    if (skipRead == PIN_MEMCPY && pageInfo.pincnt > 0) {
        throw new IllegalArgumentException("Page is allready pinned");
    }
}
```

If the page pinned it does not need to get pinned again so it just increases the pincnt and the replacer is notified.
But if the page is not pinned, the *pickVictim()* from *Clock.java* is used to find a space in the buffer the new page can be put into.

```java
int freeFrameNo = replacer.pickVictim();
        if (freeFrameNo < 0) {
            throw new IllegalStateException("could not free anything");
        }
```

just to make sure nothing went wrong there is a catching condition where the index number *pickVictim()* returned is not negative. If nothing is wrong the page that can be removed needs to be checked if it is dirty, if it is its written to the dis, and now it can be removed and the new page can be put into the buffer.

4 File Description

For this implementation no other files than the ones handed out was need so no new files where created.

5 Test Output

As per the picture at the end of the appendix every test was success-full and only failed the test that was supposed to fail.

6 Appendix

Listing 1: BufMgr.java

```java
package bufmgr;

import java.util.HashMap;

import global.GlobalConst;
import global.Minibase;
import global.Page;
import global.PageId;

/**
 * <h3>Minibase Buffer Manager</h3> The buffer manager reads disk pages into a
 * main memory page as needed. The collection of main memory pages (called
 * frames) used by the buffer manager for this purpose is called the buffer
 * pool. This is just an array of Page objects. The buffer manager is used by
 * access methods, heap files, and relational operators to read, write,
 * allocate, and de-allocate pages.
 */
@SuppressWarnings("unused")
public class BufMgr implements GlobalConst {

    /** Actual pool of pages (can be viewed as an array of byte arrays). */
    protected Page[] bufpool;

    /** Array of descriptors, each containing the pin count, dirty status, etc.
    protected FrameDesc[] frametab;

    /** Maps current page numbers to frames; used for efficient lookups. */
    protected HashMap<Integer, FrameDesc> pagemap;

    /** The replacement policy to use. */
    protected Replacer replacer;

    /**
     * Constructs a buffer manager with the given settings.
     *
     * @param numbufs: number of pages in the buffer pool
     */

    public BufMgr(int numbufs) {
        // initialize the buffer pool and frame table
        bufpool = new Page[numbufs];
        frametab = new FrameDesc[numbufs];
        for (int i = 0; i < numbufs; i++) {
          bufpool[i] = new Page();
          frametab[i] = new FrameDesc(i);
        }

        // initialize the specialized page map and replacer
        pagemap = new HashMap<Integer, FrameDesc>(numbufs);
        replacer = new Clock(this);
    }

    /**
     * Allocates a set of new pages, and pins the first one in an appropriate
     * frame in the buffer pool.
     *
     * @param firstpg
     *            holds the contents of the first page
     * @param run_size
     *            number of new pages to allocate
     * @return page id of the first new page
     * @throws IllegalArgumentException
     *             if PIN_MEMCPY and the page is pinned
     * @throws IllegalStateException
     *             if all pages are pinned (i.e. pool exceeded)
     */
    public PageId newPage(Page firstpg, int run_size) {
        // allocate the run
        PageId firstid = Minibase.DiskManager.allocate_page(run_size);

        // try to pin the first page
        try {pinPage(firstid, firstpg, PIN_MEMCPY);}
        catch (RuntimeException exc) {
              // roll back because pin failed
              for (int i = 0; i < run_size; i++) {
                firstid.pid += 1;
                Minibase.DiskManager.deallocate_page(firstid);
              }
              // re-throw the exception
              throw exc;
        }
        // notify the replacer and return the first new page id
        replacer.newPage(pagemap.get(firstid.pid));
        return firstid;
    }

    /**
     * Deallocates a single page from disk, freeing it from the pool if needed.
     * Call Minibase.DiskManager.deallocate_page(pageno) to deallocate the page
     *
     * @param pageno
     *            identifies the page to remove
     * @throws IllegalArgumentException
     *             if the page is pinned
     */
    public void freePage(PageId pageno) throws IllegalArgumentException {
        FrameDesc pageInfo = pagemap.get(pageno.pid);

        if (pageInfo == null) {
            return;
        }

        if (pageInfo.pincnt > 0) {
            throw new IllegalArgumentException("page is pinned and can't be free
        }

        pageInfo.pageno.pid = INVALID_PAGEID;
        pagemap.remove(pageno.pid);
        replacer.freePage(pageInfo);
        Minibase.DiskManager.deallocate_page(pageno);
    }

    /**
     * Pins a disk page into the buffer pool. If the page is already pinned,
     * this simply increments the pin count. Otherwise, this selects another
     * page in the pool to replace, flushing the replaced page to disk if
     * it is dirty.
     *
     * (If one needs to copy the page from the memory instead of reading from
     * the disk, one should set skipRead to PIN_MEMCPY. In this case, the page
     * shouldn't be in the buffer pool. Throw an IllegalArgumentException if so.
     *
     *
     * @param pageno
     *            identifies the page to pin
     * @param page
     *            if skipread == PIN_MEMCPY, works as as an input param, holding
     *            if skipread == PIN_DISKIO, works as an output param, holding t
     * @param skipRead
     *            PIN_MEMCPY(true) (copy the input page to the buffer pool); PIN
     * @throws IllegalArgumentException
     *             if PIN_MEMCPY and the page is pinned
     * @throws IllegalStateException
     *             if all pages are pinned (i.e. pool exceeded)
     */
    public void pinPage(PageId pageno, Page page, boolean skipRead) {
        FrameDesc pageInfo = pagemap.get(pageno.pid);

        //check if page is allready pinned
        if (pageInfo != null ) {
            if (skipRead == PIN_MEMCPY && pageInfo.pincnt > 0) {
                throw new IllegalArgumentException("Page is allready pinned");
            }
            //if the page is allready in the buffer
            //the pin count is incremented, replacer notified and
            pageInfo.pincnt++;
            replacer.pinPage(pageInfo);
            page.setPage(bufpool[pageInfo.index]);
            pagemap.put(pageno.pid, pageInfo);

        }else{

            //find a frame to remove
            int freeFrameNo = replacer.pickVictim();
            //if no frames could be removed
            if (freeFrameNo < 0) {
                throw new IllegalStateException("could not free anything");
            }
            pageInfo = frametab[freeFrameNo];

            //if the frame is valid it must be removed but also checked if its d
            if (pageInfo.pageno.pid != INVALID_PAGEID) {
                flushPage(pageInfo.pageno);
                pagemap.remove(pageInfo.pageno.pid);
            }

            /*if skipRead == PIN_MEMCPY copy the page to the buffer
            * if skipread == PIN_DISKIO read the page from the disk to the buffe
            if (skipRead) {
                bufpool[freeFrameNo].copyPage(page);
            }else{
                Minibase.DiskManager.read_page(pageno, bufpool[freeFrameNo]);
            }

            //update the frame info
            page.setPage(bufpool[freeFrameNo]);
            pageInfo.pincnt = 1;
            pageInfo.pageno.pid = pageno.pid;

            pagemap.put(pageno.pid, pageInfo);
            replacer.pinPage(pageInfo);
        }
    }

    /**
     * Unpins a disk page from the buffer pool, decreasing its pin count.
     *
     * @param pageno
     *            identifies the page to unpin
     * @param dirty
     *            UNPIN_DIRTY if the page was modified, UNPIN_CLEAN otherrwise
     * @throws IllegalArgumentException
     *             if the page is not present or not pinned
     */
    public void unpinPage(PageId pageno, boolean dirty) throws IllegalArgumentEx
        FrameDesc pageInfo = pagemap.get(pageno.pid);
        //checks if the page is pinned
        if (pageInfo == null || pageInfo.pincnt == 0) {
            throw new IllegalArgumentException("page is not pinned");
        }
        //if it is pinned the pin count is reduced and ???
        pageInfo.dirty = dirty;
        pageInfo.pincnt--;
        replacer.unpinPage(pageInfo);
    }

    /**
     * Immediately writes a page in the buffer pool to disk, if dirty.
     */
    public void flushPage(PageId pageno) { //done hopefully
        //checking if the page has any changes, if it has the it is written to d
        if (pagemap.get(pageno.pid).dirty == true){
            Minibase.DiskManager.write_page(pageno, bufpool[pagemap.get(pageno.p
            pagemap.get(pageno.pid).dirty = false;
        }

    }

    /**
     * Immediately writes all dirty pages in the buffer pool to disk.
     */
    public void flushAllPages() { //done
        for (FrameDesc f : frametab){
            if (f.pageno.pid > 0) {
                flushPage(f.pageno);
            }
        }
    }

    /**
     * Gets the total number of buffer frames.
     */
    public int getNumBuffers() { //done
        //returns the bufpool length
        return Minibase.BufferManager.bufpool.length;
    }

    /**
     * Gets the total number of unpinned buffer frames.
     */
    public int getNumUnpinned() { //done
        int count = 0;
        //runs through every frame and increments the count every time an unpinn
        for (FrameDesc f : frametab){
            if (f.pincnt == 0) {
                count++;
            }
        }
        return count;
    }

} // public class BufMgr implements GlobalConst
```

Listing 2: Replacer.java

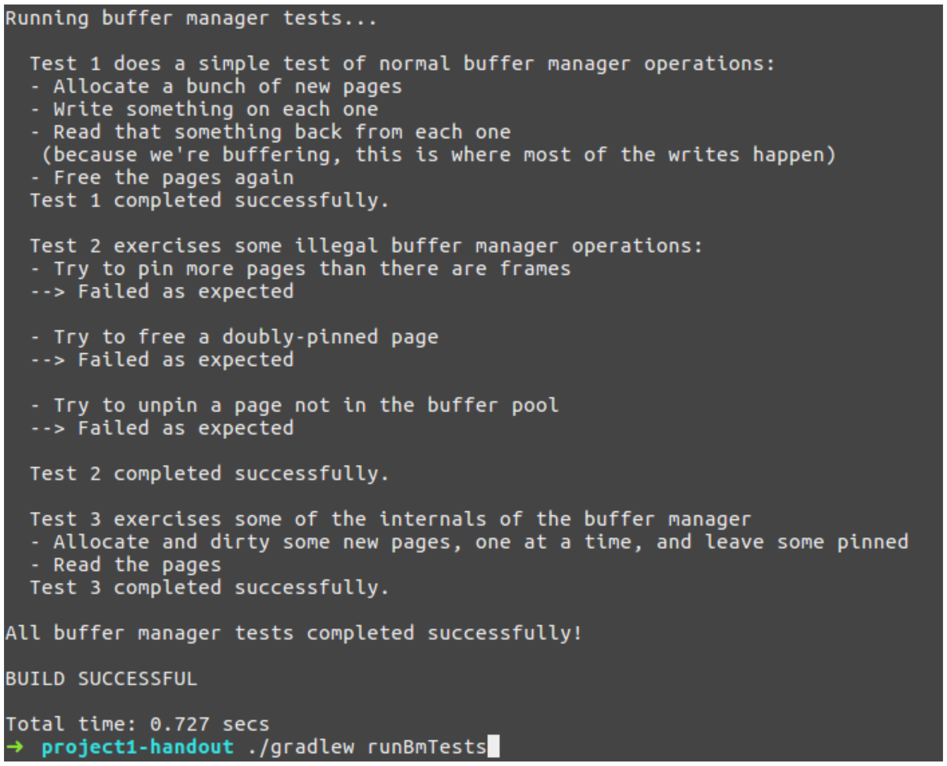
```
package bufmgr;

import global.GlobalConst;

/**
 * Base class for buffer pool replacement policies.
 */
abstract class Replacer implements GlobalConst {

  /** Reference back to the buffer manager's frame table. */
  protected FrameDesc[] frametab;

  // ----------------------------------------------------------------------------

  /**
   * Constructs the replacer, given the buffer manager.
   */
```

```java
  protected Replacer(BufMgr bufmgr) {
    this.frametab = bufmgr.frametab;
  }

  /**
   * Notifies the replacer of a new page.
   */
  public abstract void newPage(FrameDesc fdesc);

  /**
   * Notifies the replacer of a free page.
   */
  public abstract void freePage(FrameDesc fdesc);

  /**
   * Notifies the replacer of a pined page.
   */
  public abstract void pinPage(FrameDesc fdesc);

  /**
   * Notifies the replacer of an unpinned page.
   */
  public abstract void unpinPage(FrameDesc fdesc);

  /**
   * Selects the best frame to use for pinning a new page.
   *
   * @return victim frame number, or -1 if none available
   */
  public abstract int pickVictim();

} // abstract class Replacer implements GlobalConst
```

Listing 3: Clock.java

```java
package bufmgr;

public class Clock extends Replacer{
    //to identify different states of a page
    static int freeState = 1, pinned = 2, unPinned = 3;

    protected Clock(BufMgr bufmgr) {
        super(bufmgr);
        for (FrameDesc f : frametab){
            f.state = freeState;
        }
    }

    @Override
    public void newPage(FrameDesc fdesc) {
        // TODO Auto-generated method stub

    }

    @Override
    public void freePage(FrameDesc fdesc) {
        //free the state of the page
        fdesc.state = freeState;
    }

    @Override
    public void pinPage(FrameDesc fdesc) {
        //set the state of the page to pinned
        fdesc.state = pinned;
    }

    @Override
    public void unpinPage(FrameDesc fdesc) {
        // sets the state of the page to unPinned
        if (fdesc.pincnt == 0) {
            fdesc.state = unPinned;
        }
    }
    int ptr=0;
    @Override
    public int pickVictim() {
        /*run through every frame and if one frame isn't pinned
        *its state is set to free and if the function finds a free frame it is r
        *the list is sun through 2 times to be sure that if a frame was found th
        *pinned it is not freed while still being used*/
        //int ptr=0;
        for(int j=0;j<2;j++){
            for ( int i = 0 ; i < frametab.length; i++ ) {
                    FrameDesc pageInfo = frametab[ptr];
                    if(unPinned == pageInfo.state){
                        pageInfo.state = freeState;
                    }else if(freeState == pageInfo.state){
                        return ptr;
                    }
                    ptr = (ptr+1) % frametab.length;
                }
        }
```

```
            return -1;
        }
}
```



```
Running buffer manager tests...

  Test 1 does a simple test of normal buffer manager operations:
  - Allocate a bunch of new pages
  - Write something on each one
  - Read that something back from each one
   (because we're buffering, this is where most of the writes happen)
  - Free the pages again
  Test 1 completed successfully.

  Test 2 exercises some illegal buffer manager operations:
  - Try to pin more pages than there are frames
  --> Failed as expected

  - Try to free a doubly-pinned page
  --> Failed as expected

  - Try to unpin a page not in the buffer pool
  --> Failed as expected

  Test 2 completed successfully.

  Test 3 exercises some of the internals of the buffer manager
  - Allocate and dirty some new pages, one at a time, and leave some pinned
  - Read the pages
  Test 3 completed successfully.

All buffer manager tests completed successfully!

BUILD SUCCESSFUL

Total time: 0.727 secs
➜ project1-handout ./gradlew runBmTests
```