

Project 1
Database Management Systems (DM556)



**UNIVERSITY OF
SOUTHERN DENMARK**

Mark Jervelund (Mjerv15) Troels Petersen (trpet15)
IMADA

February 28, 2017

Overall Status

The overall status of of project is that we

Division of Labor

Specification

We were tasked with implementing the following functions for the bufmgr.java
freePage, pinPage, unpinPage, flushPage, flushAllPages, getNumBuffers, getNumUnpinned and pick victim.

freePage should deallocate a page from disk.

Pinpage should pin a page by incrementing the pincnt by 1, or by loading it into the bufferpool if it isnt in the bufferpool already.

Unpinpage should unpin a page, flush it to disk if its dirty and reduce the pincount by 1.

Flushpage should save a page to disk if dirty.

Flushpages should write all pages to disk if they're dirty.

getNumBuffers gets the amount of buffers.

getNumUnpinned gets the number of unpinned pages.

Design

Flushpages was implementing using Flushpage as they're almost doing the same and this reduces the amount of duplicate code.

Implementation

Pickvictim

```

1      @Override
      public int pickVictim() {
          // TODO Auto-generated method stub

5          if (Minibase.BufferManager.frametab == null) {
              // System.out.println("frametab null");
          }

          for (int i = 0; i < Minibase.BufferManager.frametab.length;
              ↪ i++) {
10         if (Minibase.BufferManager.frametab[i].pincnt == 0) {
              // System.out.println("Victim is " + i);
              return i;
          }
          return -1;
15     }

```

Testing

Conclusion

Appendix

Pickvictim

```

1      @Override
      public int pickVictim() {
          // TODO Auto-generated method stub

5          if (Minibase.BufferManager.frametab == null) {

```

```

//          System.out.println("frametab  null");
    }
        for (int i = 0; i < Minibase.BufferManager.frametab.length;
            ↪ i++) {
            if (Minibase.BufferManager.frametab[i].pincount == 0) {
10 //          System.out.println("Victim is = " +i);
                return i;
            }
        }
        return -1;
15     }
}

bufmgr.java

1 package bufmgr;

import java.util.HashMap;

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

10 /**
 * <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
15 * access methods, heap files, and relational operators to read, write,
 * allocate, and de-allocate pages.
 */
@SuppressWarnings("unused")
public class BufMgr implements GlobalConst {
20     /** Actual pool of pages (can be viewed as an array of byte arrays)
 * ↪ . */
    protected Page[] bufpool;

    private boolean debugvalue = false;

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

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

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

35     /**
 * Constructs a buffer manager with the given settings.
 *
 * @param numbufs: number of pages in the buffer pool
 *
40     */

```

```

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

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

55 /**
 * Allocates a set of new pages, and pins the first one in an
 *   ↪ appropriate
 * frame in the buffer pool.
 *
 * @param firstpg
60     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
65     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) {
70     // allocate the run
    PageId firstid = Minibase.DiskManager.allocate_page(
        ↪ run_size);

    // try to pin the first page
    System.out.println("trying_to_pin_the_first_page");
75     try {pinPage(firstid, firstpg, PIN_MEMCPY);}
        catch (RuntimeException exc) {
            System.out.println("failed_to_pin_the_first_page.");
            // roll back because pin failed
            for (int i = 0; i < run_size; i++) {
80                 firstid.pid += 1;
                Minibase.DiskManager.deallocate_page(firstid);
            }
            // re-throw the exception
            throw exc;

85         }
        // notify the replacer and return the first new page id
        replacer.newPage(pagemap.get(firstid.pid));
        return firstid;
    }

90 /**
 * Deallocates a single page from disk, freeing it from the pool if
 *   ↪ needed.
 * Call Minibase.DiskManager.deallocate_page(pageno) to deallocate
 *   ↪ the page before return.
 *

```

```

95      * @param pageno
      *         identifies the page to remove
      * @throws IllegalArgumentException
      *         if the page is pinned
      */
100  public void freePage(PageId pageno) throws IllegalArgumentException
      ↪ {
      if (pageno.pid != -1) {
          Minibase.BufferManager.flushPage(pageno);
      }
      Minibase.DiskManager.deallocate_page(pageno);
105  }

  /**
   * Pins a disk page into the buffer pool. If the page is already
   * ↪ pinned,
   * this simply increments the pin count. Otherwise, this selects
   * ↪ another
110  * 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
115  * shouldn't be in the buffer pool. Throw an
   * ↪ IllegalArgumentException if so. )
   *
   *
   * @param pageno
   *         identifies the page to pin
120  * @param page
   *         if skipread == PIN_MEMCPY, works as an input param
   * ↪ , holding the contents to be read into the buffer pool
   *         if skipread == PIN_DISKIO, works as an output param,
   * ↪ holding the contents of the pinned page read from the disk
   * @param skipRead
   *         PIN_MEMCPY(true) (copy the input page to the buffer
   * ↪ pool); PIN_DISKIO(false) (read the page from disk)
125  * @throws IllegalArgumentException
   *         if PIN_MEMCPY and the page is pinned
   * @throws IllegalStateException
   *         if all pages are pinned (i.e. pool exceeded)
   */
130  public void pinPage(PageId pageno, Page page, boolean skipRead) {
      if (debugvalue) {
          System.out.println("pinpage_called_with_pageid_" + pageno.pid + "_
          ↪ Skipread_" + skipRead + "and_page_" + page.toString());
      }
      //first check if the page is already pinned
135      FrameDesc fdesc = pagemap.get(pageno.pid);
      if (fdesc != null) {

          //Validate the pin method
          if (skipRead == PIN_MEMCPY && fdesc.pincnt > 0)
              ↪ throw new IllegalArgumentException(
140      "Page_pinned;_PIN_MEMCPY_not_allowed"

```

```

    );
    //increment pin count, notify the replacer, and wrap the buffer
    ↪ .
        fdesc.pincnt++;
    replacer.pinPage(fdesc);
145     page.setPage(bufpool[fdesc.index]);
    return;
    } // if in pool

    // select an available frame
150     int frameNo = replacer.pickVictim();
    if (frameNo < 0){
        throw new IllegalStateException("All_pages_pinned."
            ↪ );
    }

    // System.out.println(frameNo);
155 // System.out.println("skipread = " + skipRead);
    //fdesc.pageno.pid = frameNo;
    //Minibase.BufferManager.frametab[frameNo] = fdesc;

    fdesc = Minibase.BufferManager.frametab[frameNo];

160     if( fdesc.pageno.pid != INVALID_PAGEID) {
        pagemap.remove(fdesc.pageno.pid);
        if(fdesc.dirty) {
            Minibase.DiskManager.write_page(
                ↪ fdesc.pageno, bufpool[frameNo]
                ↪ );
165         }
    }

    //read in the page if requested, and wrap the buffer
    if(skipRead == PIN_MEMCPY) {
        bufpool[frameNo].copyPage(page);
170     } else {
        Minibase.DiskManager.read_page(pageno, bufpool[
            ↪ frameNo]);
    }
    page.setPage(bufpool[frameNo]);
    // if (debugvalue) {System.out.println("Pageno = " + pageno.pid);}
175 //update the frame descriptor

    fdesc.pageno.pid = pageno.pid;
    fdesc.pincnt = 1;
    fdesc.dirty = false;

180     pagemap.put(pageno.pid, fdesc);
    replacer.pinPage(fdesc);

185 }

/**
 * Unpins a disk page from the buffer pool, decreasing its pin
 * ↪ count.
 *
190 * @param pageno
 *         identifies the page to unpin
 * @param dirty

```

```

    * UNPIN_DIRTY if the page was modified, UNPIN_CLEAN
    * ↪ otherwise
    * @throws IllegalArgumentException
195    * if the page is not present or not pinned
    */
    public void unpinPage(PageId pageno, boolean dirty) throws
        ↪ IllegalArgumentException {
    if(debugvalue) {
        System.out.println("unpin_page_called_with_pageid" + pageno.pid
            ↪ + "_Dirty_status_" + dirty);
200    }
    //Checks if page is dirty.
    //first check if the page is unpinned
    FrameDesc fdesc = pagemap.get(pageno.pid);

205    if (fdesc == null) throw new IllegalArgumentException(
        "Page_not_pinned;"
    );
    if (dirty){
        flushPage(pageno);
210        fdesc.dirty = false;
    }
    fdesc.pincnt--;
    pagemap.put(pageno.pid, fdesc);
    replacer.pinPage(fdesc);
215    //unpin page.

    return;

    }

220

    /**
    * Immediately writes a page in the buffer pool to disk, if dirty.
    */
    public void flushPage(PageId pageno) {
225        FrameDesc fdesc = pagemap.get(pageno.pid);
        if (fdesc == null) {return;}
    if (debugvalue) {
        System.out.println("fdesc_=" + fdesc.index);
    }

230    if( fdesc.pageno.pid != INVALID_PAGEID) {
        pagemap.remove(fdesc.pageno.pid);
        if(fdesc.dirty) {
            Minibase.DiskManager.write_page(fdesc.pageno, bufpool[fdesc
                ↪ .index]);
235        }
    }
    }

    /**
240    * Immediately writes all dirty pages in the buffer pool to disk.
    */
    public void flushAllPages() {
        for (int i = 0 ; i < Minibase.BufferManager.frametab.length; i
            ↪ ++ ){
            flushPage(Minibase.BufferManager.frametab[i].pageno);
245    }

```

```
    }

    /**
     * Gets the total number of buffer frames.
250    */
    public int getNumBuffers() {
        return Minibase.BufferManager.bufpool.length;
    }

255    /**
     * Gets the total number of unpinned buffer frames.
     */
    public int getNumUnpinned() {
        int j = 0;
260    for (int i = 0 ; i < Minibase.BufferManager.frametab.length; i++ ){
        if (0 != Minibase.BufferManager.frametab[i].state){ j++;}
    }
    return j;
265    }

} // public class BufMgr implements GlobalConst
```