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 hit some bugs we that don't understand and are handing in this part of the project so we can get feedback and fix the current issues when we hand it Project 3.

Division of Labor

We worked on the project either sitting together at the university or at home remotely working together and spliting tasks when possible.

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.
Pickvictim gets the index for the first unpinned page, and returns -1 if all pages in the pool are pinned.

Implementation

Freepage First checks if the page is pinned. If its not, it then deallocates the page from disk.

```
    public void freePage(PageId pageno) throws IllegalArgumentException {
        FrameDesc fdesc = pagemap.get(pageno.pid);

        if (fdesc.pincnt > 0) {
            throw new IllegalArgumentException("The page is pinned.");
        }
        Minibase.DiskManager.deallocate_page(pageno);
    }
```

pinpage

```
    public void pinPage(PageId pageno, Page page, boolean skipRead) {
```

First we check if the page is already in the bufferpoll if it is we increment the pin counter

```
                FrameDesc fdesc = pagemap.get(pageno.pid);
        if (fdesc != null) {

                    // Validate the pin method
                        if (skipRead == PIN_MEMCPY && fdesc.pincnt > 0)
                            ↪ throw new IllegalArgumentException(
                    "Page pinned; PIN_MEMCPY not allowed"
            );
            // Increment pin count, notify the replacer, and wrap the
                ↪ buffer.
                        fdesc.pincnt++;
            replacer.pinPage(fdesc);
            page.setPage(bufpool[fdesc.index]);
            return;
        } // If in pool
```

If it isnt we pick a victim, and if there isnt any victims we throw an IllegalStateException.

```
        // Select an available frame
        int frameNo = replacer.pickVictim();
    // If no pages are unpinned, then throw an exception telling that.
        if (frameNo < 0){
            throw new IllegalStateException("All pages pinned."
                ↪ );
    }
```

If we have a non pinned frame we write this page to disk if its dirty.

```
    // Pick the frame that is not pinned.
        fdesc = Minibase.BufferManager.frametab[frameNo];
        // If the frame was in use and dirty, it should write it to
            ↪ the disk.
        if( fdesc.pageno.pid != INVALID_PAGEID) {
                pagemap.remove(fdesc.pageno.pid);
                if(fdesc.dirty) {
                    Minibase.DiskManager.write_page(
                        ↪ fdesc.pageno, bufpool[frameNo
                        ↪ ]);
```

And if it isnt dirty we copy or read the new page into the bufferpool and update the pagemap.

```
        //read in the page if requested, and wrap the buffer
        if(skipRead == PIN_MEMCPY) {
            bufpool[frameNo].copyPage(page);
        } else {
            Minibase.DiskManager.read_page(pageno, bufpool[
                ↪ frameNo]);
        }
        page.setPage(bufpool[frameNo]);

        //update the frame descriptor
    fdesc.pageno.pid = pageno.pid;
    fdesc.pincnt = 1;
    fdesc.dirty = false;

    // Pin the page and put the updated page in the pagemap.
    pagemap.put(pageno.pid, fdesc);
    replacer.pinPage(fdesc);
```

unpinpage

```
  public void unpinPage(PageId pageno, boolean dirty) throws
    ↪ IllegalArgumentException {
```

First we check if the page is pinned. if its not we throw an exception.

```
    FrameDesc fdesc = pagemap.get(pageno.pid);
    if (fdesc == null) throw new IllegalArgumentException(
        "Page not pinned;"
    );
```

If its in the buffpool we decrement the pagecounter by 1, and update the pagemap with the new information.

```
        if (dirty){
            flushPage(pageno); fdesc.dirty = false;
        }
        // Decrement the pin count, since the page is pinned by one less.
            ↪ Also unpin the page and update the page in the
        // pagemap.
        fdesc.pincnt--;
        pagemap.put(pageno.pid, fdesc);
        replacer.unpinPage(fdesc);
```

flushpage

```
        public void flushPage(PageId pageno) {
```

Pickvictim is implemented to return the index for the first element with pincnt 0. and if all elements are in use it returns -1 to indicate this.

```
        @Override
        public int pickVictim() {
        // Finds the first element in the frametab array, where pin count
            ↪ is equal to zero and returns it.
                for (int i = 0; i < Minibase.BufferManager.frametab.length;
                    ↪ i++) {
            if(Minibase.BufferManager.frametab[i].pincnt == 0) {
                return i;
                        }
                }
                // If no pages has zero pins, then it returns -1.
        return -1;
```

Testing

From the testing we've done the programs gets into a neverending loop pin/unpin loop at SystemCatalog = new Catalog(false) in Minibase.java line 79 (my file with some print statements for debugging)

Conclusion

Appendix

Pickvictim

```
        @Override
        public int pickVictim() {
        // Finds the first element in the frametab array, where pin count
            ↪ is equal to zero and returns it.
                for (int i = 0; i < Minibase.BufferManager.frametab.length;
                    ↪ i++) {
            if(Minibase.BufferManager.frametab[i].pincnt == 0) {
                return i;
                        }
                }
                // If no pages has zero pins, then it returns -1.
        return -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;

    private boolean debugvalue = false;

    /**
     * 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
        System.out.println("trying␣to␣pin␣the␣first␣page");
        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++) {
                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 before return.
     *
     * @param pageno identifies the page to remove
     * @throws IllegalArgumentException if the page is pinned
     */
    public void freePage(PageId pageno) throws IllegalArgumentException {
        FrameDesc fdesc = pagemap.get(pageno.pid);

        if (fdesc.pincnt > 0) {
            throw new IllegalArgumentException("The␣page␣is␣pinned.");
        }
        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.
     * <p>
     * (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 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)
     * @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) {
        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
                FrameDesc fdesc = pagemap.get(pageno.pid);
        if (fdesc != null) {

                        // Validate the pin method
                            if (skipRead == PIN_MEMCPY && fdesc.pincnt > 0)
                               ↪ throw new IllegalArgumentException(
                        "Page pinned; PIN_MEMCPY not allowed"
            );
            // Increment pin count, notify the replacer, and wrap the
               ↪ buffer.
                            fdesc.pincnt++;
            replacer.pinPage(fdesc);
            page.setPage(bufpool[fdesc.index]);
            return;
                } // If in pool

                // Select an available frame
                int frameNo = replacer.pickVictim();
        // If no pages are unpinned, then throw an exception telling that.
                if (frameNo < 0){
                        throw new IllegalStateException("All pages pinned."
                            ↪ );
        }

        // Pick the frame that is not pinned.
                fdesc = Minibase.BufferManager.frametab[frameNo];
                // If the frame was in use and dirty, it should write it to
                   ↪ the disk.
                if( fdesc.pageno.pid != INVALID_PAGEID) {
                                pagemap.remove(fdesc.pageno.pid);
                                if(fdesc.dirty) {
                                        Minibase.DiskManager.write_page(
                                           ↪ fdesc.pageno, bufpool[frameNo
                                           ↪ ]);
                                }
                        }

                //read in the page if requested, and wrap the buffer
                if(skipRead == PIN_MEMCPY) {
                        bufpool[frameNo].copyPage(page);
                } else {
                        Minibase.DiskManager.read_page(pageno, bufpool[
                           ↪ frameNo]);
                }
                page.setPage(bufpool[frameNo]);

                //update the frame descriptor
        fdesc.pageno.pid = pageno.pid;
        fdesc.pincnt = 1;
        fdesc.dirty = false;

        // Pin the page and put the updated page in the pagemap.
        pagemap.put(pageno.pid, fdesc);
        replacer.pinPage(fdesc);
        }

    /**
     * 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
        ↪ otherwise
     * @throws IllegalArgumentException 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);
        //Checks if page is dirty.
        // First check if the page is unpinned
        FrameDesc fdesc = pagemap.get(pageno.pid);
        if (fdesc == null) throw new IllegalArgumentException(
                "Page␣not␣pinned;"
        );
        // If dirty, it should write the the page to the disk and then tell
           ↪ that the page is not dirty anymore.
        if (dirty){
            flushPage(pageno); fdesc.dirty = false;
        }
        // Decrement the pin count, since the page is pinned by one less.
            ↪ Also unpin the page and update the page in the
        // pagemap.
        fdesc.pincnt--;
        pagemap.put(pageno.pid, fdesc);
        replacer.unpinPage(fdesc);
        //unpin page.

        return;

    }

        /**
         * Immediately writes a page in the buffer pool to disk, if dirty.
         */
        public void flushPage(PageId pageno) {

            // Check if page is unpinned
                FrameDesc fdesc = pagemap.get(pageno.pid);

                // If it is pinned, it cannot flush the page and thus must
                    ↪ return.
                if (fdesc == null)  {return;}
        if (debugvalue) System.out.println("fdesc = " + fdesc.index);

                // If the page exists, it should be written to the disk.
        if( fdesc.pageno.pid != INVALID_PAGEID) {
            // Since it is being written to the disk, it shouldn't be in
                ↪ the pagemap anymore.
            pagemap.remove(fdesc.pageno.pid);
            if (fdesc.dirty) {
                Minibase.DiskManager.write_page(fdesc.pageno, bufpool[fdesc
                    ↪ .index]);
            }
        }
    }

        /**
         * 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);
        }
    }

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

        /**
         * Gets the total number of unpinned buffer frames.
         */
        public int getNumUnpinned() {
            // Using a loop, this checks the state of each frame. Each time
                ↪  an unpinned frame is found, "j" is incremented.
        // In the end "j" is returned, as that must be the total amount of
            ↪ unpinned buffer frames.
            int j = 0;
        for (int i = 0 ; i < Minibase.BufferManager.frametab.length; i++ )
            ↪ {
            if (0 != Minibase.BufferManager.frametab[i].state) j++;
        }
        return j;
    }

} // public class BufMgr implements GlobalConst
```