Principle of Database Systems - Project 1: Buffer Manager

18/03/2018

Southern University of Denmark

Authors: Sebastian Eklund Larsen & Kasper Kromann Nissen

Mails: selar16@student.sdu.dk, kanis16@student.sdu.dk.

Overall status

The program was implemented by implementing a buffer manager and a clock replacement policy. After the required methods were implemented, the program was tested with BMTest, and it passed successfully.

In order to implement the clock replacement policy, we created our own algorithm in the method pickVictim, and inserted state changes in the methods which get called by the buffermanager.

The other major component, namely the buffer manager itself, was implemented by the completion of the most prominent of its methods:

- pinPage
- unpinPage
- freePage
- flushPage

We implemented the methods according to their descriptions in the skeleton project. We also made use of our understanding we had gathered from reading the pages in book.

All of the methods we implemented were checked through by the both of us in a step-by-step fashion.

File descriptions

In order to implement a functioning buffermanager, we made changes to the following two files in the skeleton-project folder

Clock.java:
Implementation of a clock replacement policy.

BufMgr.java:
Implementation of the missing methods, such as pinPage and flushPage.

Division of labor

We distributed the work by having each member work on implementing a method, and once done with said method start on implementing the next unimplemented method. We encountered problems with the pinPage method which meant that one of us was stuck on it, and only after having both members look through the program were we able to fix it.

Sebastian Larsen:
Clock, bufmgr.freePage, bufmgr.getNumUnpinned, bufmgr.flushAllPages, bufmgr.flushPage.

Kasper Nissen:
bufmgr.getNumBuffers, bufmgr.pinPage, bufmgr.unpinPage, bufmgr.pinPageFound, bufmgr.pinPageSetup.

Besides the distribution of methods by the author above, an author tag has been added to the comments of each of the implemented methods in the source code files.

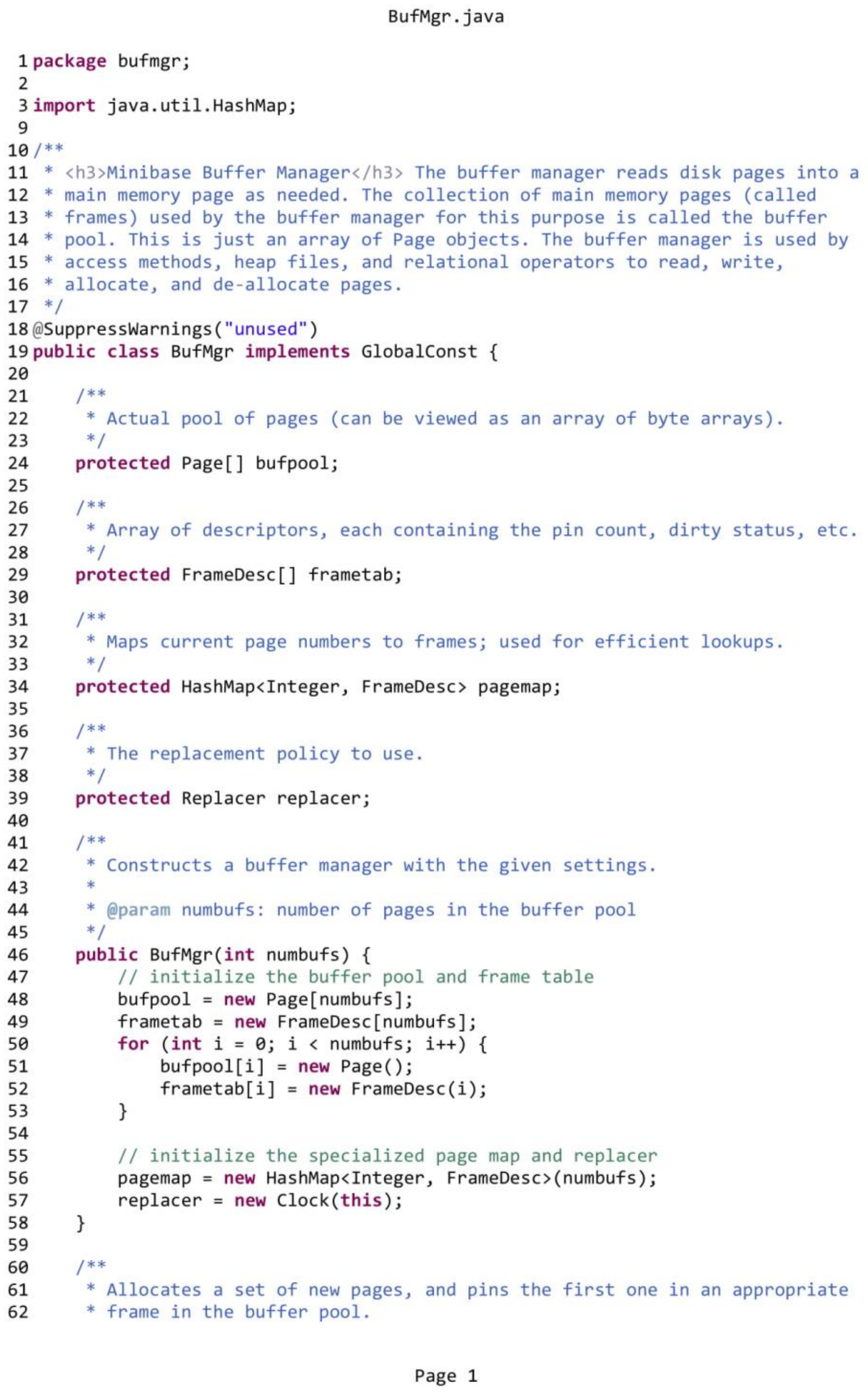
Appendix

Test Output

The code was tested by running BMTest, giving the following output:

```
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!
```

Code:



BufMgr.java

```java
package bufmgr;

import java.util.HashMap;

/**
 * <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.
```

Page 1

```java
     *
     * @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;
    }

    /**
     *
     * @author Sebastian Larsen
     *
     * 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 != null) {
            if (fdesc.pincnt > 0) {
                throw new IllegalArgumentException();
            }
            fdesc.pageno.pid = INVALID_PAGEID;
            pagemap.remove(pageno.pid);
            replacer.freePage(fdesc);

        }
        Minibase.DiskManager.deallocate_page(pageno);
    }

    /**
     * @author Kasper Nissen
     *
```

BufMgr.java

```java
     * 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 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) {

        // attempt to retrieve the FrameDesc from the pagemap
        FrameDesc desc = pagemap.get(pageno.pid);

        // if succesfull
        if (desc != null) {
            // increases the pincnt on the found page,
            // as well as ensuring the other necessary methods are called
            pinPageFound(page, skipRead, desc);
            replacer.pinPage(desc);
            return;
        } else {
            // find the next page to replace
            int victim = replacer.pickVictim();

            // if there is no replaceable pages
            if (victim == -1) {
                throw new IllegalStateException();
            }

            // replace the victim page with the new page
            desc = frametab[victim];

            // if a valid page is found at the victim index // TODO
            if (desc.pageno.pid != INVALID_PAGEID) {
                // remove the valid page from the pagemap
                pagemap.remove(desc.pageno.pid);

                // if dirty
                if (desc.dirty) // write the page to disk
                {
                    Minibase.DiskManager.write_page(desc.pageno, bufpool[victim]);
                }
            }
```

Page 3

BufMgr.java

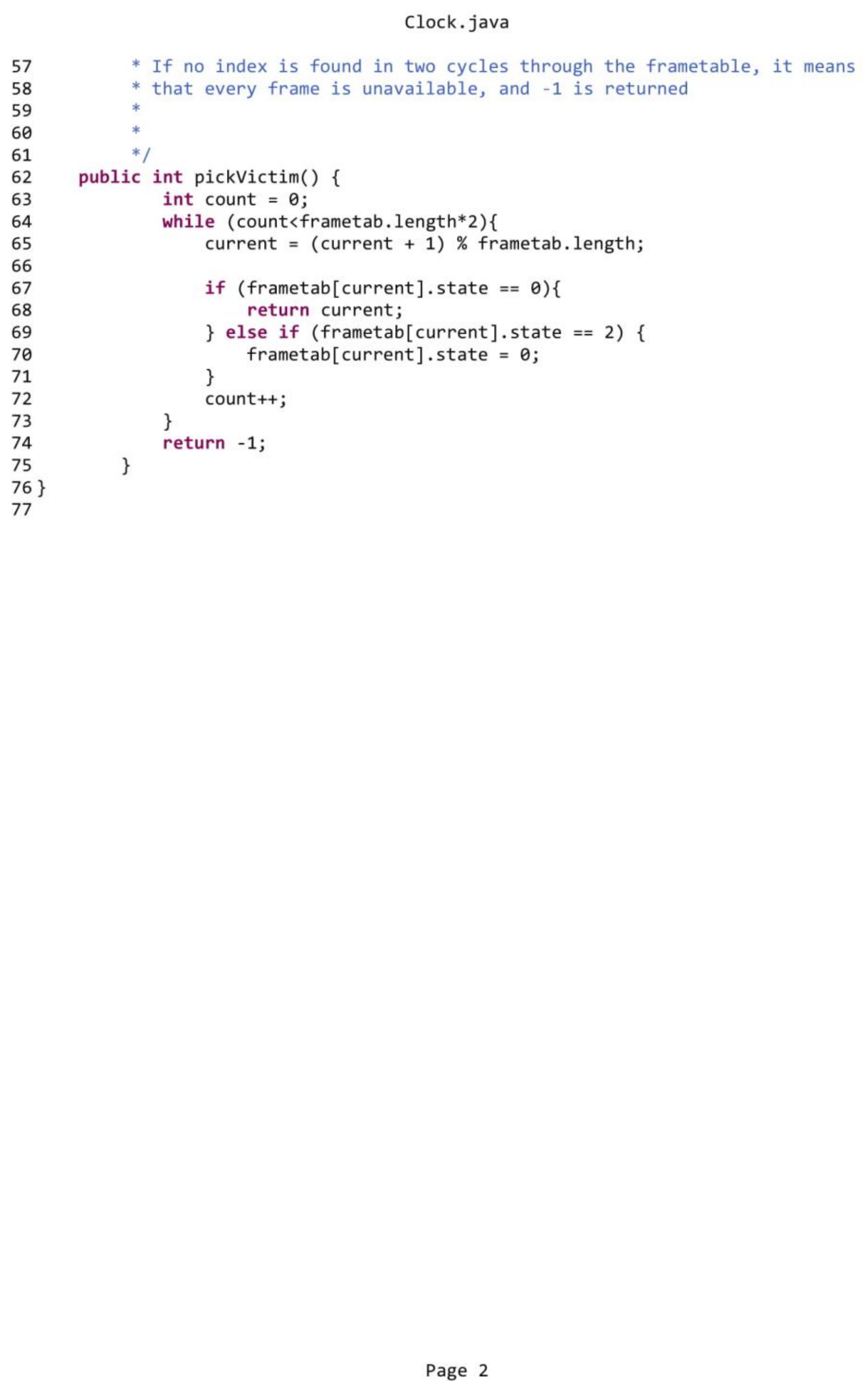
```java

            pinPageSetup(pageno, page, skipRead, desc, victim);
            replacer.pinPage(desc);
        }
    }

    /**
     * @author Kasper Nissen
     *
     * If a page already in the pagemap is to be pinned, this method will be
     * called. It ensures that the pincnt is incremented and that the correct
     * replacer flags are set.
     *
     * @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)
     * @param desc the FrameDesc object that holds the FrameDesc tied to the
     * page found
     */
    private void pinPageFound(Page page, boolean skipRead, FrameDesc desc) {
        if (skipRead == PIN_MEMCPY) {
            throw new IllegalArgumentException("invalid argument, birch");
        }

        // pins the page
        desc.pincnt++;
        page.setPage(bufpool[desc.index]);
    }

    /**
     * @author Kasper Nissen
     *
     * ** used in pinPage ** ensures that page is initialized with the right
     * data, according to the value of skipRead.
     *
     * @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)
     * @param desc the FrameDesc object that holds the FrameDesc tied to the
     * page found
     * @param victim the page to be replaced, determined by the replacement
     * policy in use
     */
    private void pinPageSetup(PageId pageno, Page page, boolean skipRead, FrameDesc desc, int victim) {
        // if skipRead == PIN_MEMCPY
        if (skipRead) {
            // copy from memory
            bufpool[victim].copyPage(page);
        } // if skipRead == PIN_DISKIO
```

Page 4

```
        else {
            // read from disk
            Minibase.DiskManager.read_page(pageno, bufpool[victim]);
        }

        desc.pincnt = 1;
        page.setPage(bufpool[victim]);
        pagemap.put(pageno.pid, desc);
        desc.pageno.pid = pageno.pid;
    }

    /**
     * @author Kasper Nissen
     *
     * 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 IllegalArgumentException {

        if (!pagemap.containsKey(pageno.getPID())) {
            throw new IllegalArgumentException();
        }

        FrameDesc desc = pagemap.get(pageno.getPID());

        if (desc.pincnt == 0) {
            throw new IllegalArgumentException();
        }

        desc.pincnt--;

        if (dirty == UNPIN_DIRTY) {
            desc.dirty = true;
        }
        replacer.unpinPage(desc);
    }

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

        FrameDesc fdesc = pagemap.get(pageno.pid);

        if (fdesc.dirty) {
            Minibase.DiskManager.write_page(pageno, bufpool[fdesc.index]);
            fdesc.dirty = false;
        }
    }

```

```java
    /**
     *
     * @author Sebastian Larsen
     *
     * Immediately writes all dirty pages in the buffer pool to disk, skipping
     * the pages which have INVALID_PAGEID
     */
    public void flushAllPages() {
        for (int i = 0; i < frametab.length; i++) {
            if (frametab[i].pageno.pid != INVALID_PAGEID) {
                flushPage(frametab[i].pageno);
            }
        }
    }

    /**
     * @author Kasper Nissen
     *
     * Gets the total number of buffer frames.
     */
    public int getNumBuffers() {
        return frametab.length;
    }

    /**
     *
     * @author Sebastian Larsen
     *
     * Gets the total number of unpinned buffer frames.
     */
    public int getNumUnpinned() {
        int unpin_count = 0;
        for (int i = 0; i < frametab.length; i++) {
            if (frametab[i].pincnt == 0) {
                unpin_count++;
            }
        }
        return unpin_count;
    }

} // public class BufMgr implements GlobalConst

```

```java
package bufmgr;
        /**
         *
         * @author Sebastian Larsen
         *
         * The clock replacement policy looks at the frametable as a clock
         * and loops through the indices evauluating them based on their state.
         *
         * The state of an index can be one of the following 3
         * 0 = available : the index can be replaced
         * 1 = pinned : the index cannot be replaced
         * 2 = prevpinned : should be set to available instead of being replaced when picked.
         */
public class Clock extends Replacer{

        private int current;

    protected Clock(BufMgr bufmgr) {
        super(bufmgr);
        current = -1;
    }

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

    }

    @Override
    public void freePage(FrameDesc fdesc) {

            fdesc.state=0;

    }

    @Override
    public void pinPage(FrameDesc fdesc) {
        fdesc.state=1;

    }

    @Override
    public void unpinPage(FrameDesc fdesc) {
        if(fdesc.pincnt == 0){
                    fdesc.state=2;
                }

    }

    @Override
        /**
         *
         * pickVictim tries to pick a frame which can be replaced.
         * Loops through the frametable upto 2 times looking for a frame to be replaced.
         * For each frame it reaches it checks if its state is available(0), in which the index is returned,
         * or if the state is prevpinned(2), in which case it is set to available.
```



Clock.java

```
         * If no index is found in two cycles through the frametable, it means
         * that every frame is unavailable, and -1 is returned
         *
         *
         */
    public int pickVictim() {
            int count = 0;
            while (count<frametab.length*2){
                current = (current + 1) % frametab.length;

                if (frametab[current].state == 0){
                    return current;
                } else if (frametab[current].state == 2) {
                    frametab[current].state = 0;
                }
                count++;
            }
            return -1;
        }
}
```

Page 2