CISS430: Database Systems
Assignment 1

OBJECTIVES

1. Write a simple simulation of a very basic buffer pool management system to understand the buffer pool management system for an RDBMS.

SUBMISSION

- For Q1, I will run your program using

```
g++ *.cpp; ./a.out
```

  You can have as many files as you like for Q1.

1. Archive and compress your a01:

```
tar -cvf a01.tar a01; gzip a01.tar
```

2. Email `yliow.submit@gmail.com` your `a01.tar.gz` from your college email account. The subject line of your email must be `ciss430 a01`.
3. If you have any questions about submission, make sure you talk to me or check with someone in class.

Q1.

The goal is to write a buffer pool management system. But for now, to understand this, we will write a very simple simulation. The real buffer pool management system will be implemented in a future assignment.

Here are some general ideas.

Say you have a file on the harddisk. The file is made up of blocks where each block is, say, made up of 4096 bytes. This means that each read/write from the harddisk to memory is by chunks of 4096 bytes, i.e., by blocks. Even if you read 5 bytes at the beginning of your file, your OS will read the block containing this 5 bytes (i.e., the first block).

Frequently, you do not need to read all the blocks of the file into memory. In fact if the file is huge, that would be impossible.

Say you want to work with the second block of your file – maybe you want to do some reads and some writes. You can of course read and write direct to this block on the harddisk. But disk I/O is slow. (Google and read up on memory hierarchy or check your notes on CISS176, CISS360, etc.) A better way would be to fetch that block into memory and then do your reads and writes of this block in memory. When you are done, you copy the data of this block (in memory) to the block in you harddrive.

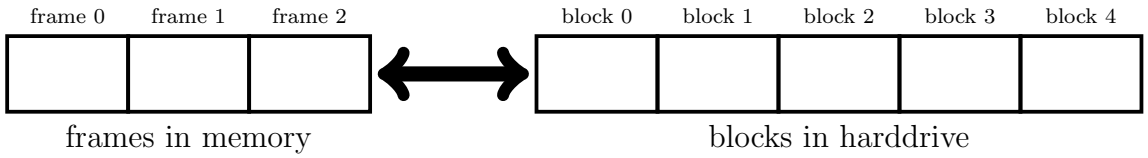
The point is to speed up the reads and writes. The buffer pool management system is basically a system that caches disk pages in memory.

A buffer pool management system brings blocks into memory for reads and writes and then, when necessary, write the blocks in memory to the blocks on the harddisk

When a block from the harddisk is fetched into memory, that 4096 bytes in memory is usually called a frame.

Another term: The term block really refers to a block of data for I/O. Usually I will say page instead of block. A page might be a block or it might be part of a block or ... We do not need to worry too much about this. You can view a page as a block if you like or you can think of it as a virtual block.

Here's a diagram of a scenario where a software manages a buffer pool with 3 frames while working on a store files made up of 5 blocks:



The thing to keep in mind is that the memory is faster than storage, but memory is more expensive than storage. The buffer pool management system usually has fewer frames than the number of blocks it has to work on.

For this exercise, you will assume that the block (or page) size is 4 bytes (otherwise debugging would be a nightmare). Assume that you have a file with 5 blocks (or pages), i.e., this file has 20 bytes. The name of this file is `abc`. So go ahead and create the file `abc` with the following contents (as characters):

```
aaaabbbbccccddddeeee
```

(Just use a text editor.)

We will assume that your buffer pool will only have 3 frames. Each frame can hold 4 bytes as well.

The following is an execution of your program that simulates a simple buffer pool management system. Study it and the comments below very carefully.

```
[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[][][]
option: 0
which page? 1

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][][]
option: 0
which page? 3

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][3:dddd][]
option: 0
which page? 0
```

```
[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][3:dddd][0:aaaa]
option: 0
which page? 3
page 3 is already fetched ... frame id is 1

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][3:dddd][0:aaaa]
option: 0
which page? 1
page 1 is already fetched ... frame id is 0

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][3:dddd][0:aaaa]
option: 0
which page? 0
page 0 is already fetched ... frame id is 2

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][3:dddd][0:aaaa]
option: 1
which page? 3
enter 4 characters: p p p p

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][*3:pppp][0:aaaa]
option: 1
which page? 0
enter 4 characters: q q q q

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][*3:pppp][*0:qqqq]
```

```
option: 0
which page? 4
which frame to remove? 1
frame 1 is dirty ... write to block 3

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[1:bbbb][4:eeee][*0:qqqq]
option: 0
which page? 3
which frame to remove? 0
frame 0 is not dirty ... no write

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[3:pppp][4:eeee][*0:qqqq]
option: 1
which page? 3
enter 4 characters: r r r r

[0] Fetch a page into memory
[1] Write frame
[2] Shutdown
Frames:[*3:rrrr][4:eeee][*0:qqqq]
option: 2
frame 0 is dirty ... write to block 3
frame 2 is dirty ... write to block 0
halt
```

In the above execution, the output

```
Frames:[1:bbbb][*3:pppp][0:aaaa]
```

tells you that the buffer pool management system has already full~~ed~~y used up its three frames.

- Frame 0 holds data for block 1 of the file.
- Frame 1 holds data for block 3 of the file.
- Frame 2 holds data for block 0 of the file.

The * in the output of each frame indicates that the original data has been changed, (We say that the frame is now dirty.) This occurs at frame 1 above which holds data for block 3. If the buffer pool management system shuts down, then the contents of frame 1 must be

copied onto block 3.

MAKE SURE YOUR OUTPUT FOLLOWS THE ABOVE FORMAT. The grading is harsh. I might change the order of the operations or change the data written to the pages, but the format of the output must match the above trial run.

When the above execution ends, the data in `abc` is

```
qqqqbbbbccccrrrreeee
```

Notes on POSIX file I/O is on the next page.

I don't really care how you implement your buffer pool, i.e., I don't care how you implement your 3 frames and the way you remember the block number of each frame, and the $*$ that remembers a frame is dirty, i.e., was modified.

However you must use POSIX file I/O correctly. For instance your read and write should be done in a loop. Also, when you want to put the data from a page into a frame, you should only read a page. When you write a frame to the file, you should only write a frame. For instance if only one frame is dirty, you should only copy that frame to a block on the harddrive – you should NOT copy all the frames to the respective blocks.

ASIDE. Once you're done with the above, you should see that when the number of frames is smaller than the number of pages (which is usually the case), the buffer pool management system frequently needs to decide what frame to remove in order to make way for a new page. The frame to be removed is usually called the **victim**. The algorithm to decide the victim is called the **replacement algorithm**. These terms and most of the general ideas in this question applies to any form of cache system and not just a RDBMS buffer pool management system.

POSIX File I/O

There are many ways to perform file I/O. You have already seen C++ file streams (in CISS245). For this assignment, we will be using POSIX file I/O. For information about POSIX, check wikipedia.

Includes:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <cerrno>
#include <cstring>
```

To open a file for read and write:

```
int fd = open(filename, O_RDWR);
if (fd < 0)
{
    std::cout << "open fail\n";
    std::cout << "errno: " << errno << std::endl;
    std::cout << "strerror: " << strerror(errno) << std::endl;
}
else
{
    std::cout << "open ok" << std::endl;
}
```

If you want to open it for read and write, and create it if the file does not exists, you want to create it:

```
int fd = open(filename, O_RDWR | O_CREAT, 0644);
```

(fd = file descriptor).

To read a file

```
int size = 100;
unsigned char buff[1024];
int result_size;

result_size = read(fd, buff, size);
```

Read 100 bytes into `buff`. Note that `result_size` might be less than 100. If so, you have to keep doing it in a loop until you get all the 100 bytes. If you hit an EOF, `read` will return

0.

To write to the file:

```
int size = 100;
unsigned char buff[1024];
int result_size;

result_size = write(fd, buff, size);
```

Write 100 bytes `buff` into file. Note that `result_size` might be less than 100. If so, you have to keep doing it in a loop until you have written all the bytes.

To move the file's read/write pointer with respect to current position:

```
lseek(fd, 100, SEEK_CUR);
```

In this case, move the pointer by 100 forward. Of course you can move backwards:

```
lseek(fd, -40, SEEK_CUR);
```

-1 is returned if there's an error. To move the file's read/write pointer with respect to beginning-of-file position:

```
lseek(fd, 200, SEEK_SET);
```

lseek returns the pointer position after the move; the type is an int although officially POSIX uses `off_t`. Because of this, using lseek, you can find the current pointer position by not moving from the current position:

```
off_t pos;
pos = lseek(fd, 0, SEEK_CUR);
```

To close the file:

```
close(fd);
```

If there's an error, you can fetch the error message like the above.

You can easily find out more about POSIX file I/O on the web.

As an exercise before doing the assignment question, you should create a simple text file, and using POSIX, open the file, move the file position forward, read a few characters, move the file position backward, read a few characters, move forward, write a few characters to the file, and close the file.

*** SPOILERS *** SPOILERS *** SPOILERS ***

HINTS/SUGGESTIONS

Here are some suggestions/hints:

Variables:

1. Clearly you need to keep an array of frames (which are arrays of `unsigned char` i.e., bytes).
2. You need to know if a frame contains data from a block (or not).
3. For each frame you need to know which block it comes from. For instance frame at index 2 might come from the 3rd block.
4. You also need to know if a frame is dirty (or not).

Therefore you would need (at least):

```
unsigned char frame[NUM_FRAMES][PAGE_SIZE];
bool available[NUM_FRAMES];
int block_index[NUM_FRAMES];
bool dirty[NUM_FRAMES];
```

where `NUM_FRAMES` is number of frames and `PAGE_SIZE` if the number of bytes in a block or a frame. (Clearly you should know by now that you should never hardcode constants.)

You might want to write functions for the various POSIX file operations to simplify your code.

1. For instance the open might result in an error. So you can create an exception class for throwing when open gives an error and then create your own open function that perform POSIX open and throw your exception is necessary. This way, your `main()` won't have the code to check for potential errors.
2. For the read, the operation might be partial, so you need to loop until all requested bytes are read. So you can write your own read function to loop over POSIX read until it's done, and the function also throws an exception if an error occurs.
3. Same for write.

Obviously you want to test these functions on their own.

Once your buffer pool simulation is completely done and passes the tests, if you like, you can also wrap everything up nicely into a class:

```
const int NUM_FRAMES = 3;
const int PAGE_SIZE = 4;
class BuffPoolMgr
{
public:
    // methods to load a block into a frame, save a frame to a block,
```

```
    // write data into frame, etc.
private:
    unsigned char frame[NUM_FRAMES][PAGE_SIZE];
    bool available[NUM_FRAMES];
    int block_index[NUM_FRAMES];
    bool dirty[NUM_FRAMES];
};
```

or to make it flexible, instead of using constants, you can keep all arrays in the heap using pointers:

```
class BuffPoolMgr
{
private:
    int num_frames_;
    int page_size_;
    unsigned char ** frame_;
    bool * available_;
    int * block_index_;
    bool * dirty_;
};
```

or using `std::vector`.

[See LOG for spring 2020 changes.]