# Advanced Database Organization - Fall 2023
# CS 525 - 04/05
# Programming Assignment I: Storage Manager
# Due: Monday, September 18th 2023 by 11:59pm

---

## 1. Task

The goal of this assignment is to implement a simple storage manager - a module that is capable of reading blocks from a file on disk into memory and writing blocks from memory to a file on disk. The storage manager deals with pages (blocks) of fixed size (`PAGE_SIZE`). In addition to reading and writing pages from a file, it provides methods for creating, opening, and closing files. The storage manager has to maintain several types of information for an open file: The number of total pages in the file, the current page position (for reading and writing), the file name, and a POSIX file descriptor or `FILE` pointer. In your implementation you should implement the interface described below. Please commit a text file `README.txt` that (shortly) describes the ideas behind your solution and the code structure. Comment your code!

## 2. Interface

The interface your storage manager should implement is given as a header file `storage_mgr.h`. The content of this header is shown below. Two additional headers `dberror.h` and `test_helpers.h` define error codes and constants and macros used in the test cases.

```
01 | #ifndef STORAGE_MGR_H
02 | #define STORAGE_MGR_H
03 |
04 | #include "dberror.h"
05 |
06 | /************************************************************
07 |  *                    handle data structures                *
08 |  ************************************************************/
09 | typedef struct SM_FileHandle {
10 |   char *fileName;
11 |   int totalNumPages;
12 |   int curPagePos;
13 |   void *mgmtInfo;
14 | } SM_FileHandle;
15 |
16 | typedef char* SM_PageHandle;
17 |
18 | /************************************************************
19 |  *                    interface                             *
20 |  ************************************************************/
21 | /* manipulating page files */
22 | extern void initStorageManager (void);
23 | extern RC createPageFile (char *fileName);
24 | extern RC openPageFile (char *fileName, SM_FileHandle *fHandle);
25 | extern RC closePageFile (SM_FileHandle *fHandle);
26 | extern RC destroyPageFile (char *fileName);
27 |
28 | /* reading blocks from disc */
29 | extern RC readBlock (int pageNum, SM_FileHandle *fHandle, SM_PageHandle memPage);
30 | extern int getBlockPos (SM_FileHandle *fHandle);
31 | extern RC readFirstBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
32 | extern RC readPreviousBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
33 | extern RC readCurrentBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
34 | extern RC readNextBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
35 | extern RC readLastBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
36 |
37 | /* writing blocks to a page file */
38 | extern RC writeBlock (int pageNum, SM_FileHandle *fHandle, SM_PageHandle memPage);
39 | extern RC writeCurrentBlock (SM_FileHandle *fHandle, SM_PageHandle memPage);
40 | extern RC appendEmptyBlock (SM_FileHandle *fHandle);
41 | extern RC ensureCapacity (int numberOfPages, SM_FileHandle *fHandle);
42 |
43 | #endif
```

2.1. **Data structures.** The page size is hard-coded in the header file `dberror.h` (`PAGE_SIZE`). Each of the methods defined in the storage manager interface returns an integer return code also defined in `dberror.h` (`RC`). For details see return codes below.
The methods in the interface use the following two data structures to store information about files and pages:

- `File Handle SM_FileHandle`
  - A file handle `SM_FileHandle` represents an open page file. Besides the file name, the handle stores the total number of pages in the file and the current page position.
  - The current page position is used by some of the read and write methods of the storage manager.
  - For example, `readCurrentBlock` reads the `curPagePos=th` page counted from the beginning of the file. When opening a file, the current page should be the first page in the file (`curPagePos=0`) and the `totalNumPages` has to be initialized based on the file size.
  - Use the `mgmtInfo` to store additional information about the file needed by your implementation, e.g., a POSIX file descriptor.

    **Hint**: You should reserve some space in the beginning of a file to store information such as the total number of pages.

    **Hint**: Use `mgmtInfo` to store any bookkeeping info about a file your storage manager needs.

    ```
    typedef struct SM_FileHandle {
      char *fileName;
      int totalNumPages;
      int curPagePos;
      void *mgmtInfo;
    } SM_FileHandle;
    ```

- `Page Handle SM_PageHandle`
  - A page handle is a pointer to an area in memory storing the data of a page.
  - Methods that write the data pointed to by a page handle to disk or read a page from disk into the area of memory pointed to by the page handle require that the handle is pointing to an previously allocated block of memory that is at least `PAGE_SIZE` number of bytes long.

    ```
    typedef char* SM_PageHandle;
    ```

2.2. **File Related Methods.**

- `createPageFile`
  - Create a new page file `fileName`. The initial file size should be one page. This method should fill this single page with '\0' bytes.
- `openPageFile`
  - Opens an existing page file. Should return `RC_FILE_NOT_FOUND` if the file does not exist.
  - The second parameter is an existing file handle.
  - If opening the file is successful, then the fields of this file handle should be initialized with the information about the opened file. For instance, you would have to read the total number of pages that are stored in the file from disk.
- `closePageFile`, `destroyPageFile`
  - Close an open page file or destroy (delete) a page file.

2.3. **Read and Write Methods.** There are two types of read and write methods that have to be implemented: Methods with absolute addressing (e.g., `readBlock`) and methods that address relative to the current page of a file (e.g., `readNextBlock`).

- `readBlock`
  - The method reads the block at position `pageNum` from a file and stores its content in the memory pointed to by the `memPage` page handle.
  - If the file has less than `pageNum` pages, the method should return `RC_READ_NON_EXISTING_PAGE`.
- `getBlockPos`
  - Return the current page position in a file
- `readFirstBlock`, `readLastBlock`
  - Read the first respective last page in a file
- `readPreviousBlock`, `readCurrentBlock`, `readNextBlock`
  - Read the current, previous, or next page relative to the `curPagePos` of the file.
  - The `curPagePos` should be moved to the page that was read.
  - If the user tries to read a block before the first page or after the last page of the file, the method should return `RC_READ_NON_EXISTING_PAGE`.
- `writeBlock`, `writeCurrentBlock`
  - Write a page to disk using either the current position or an absolute position.
- `appendEmptyBlock`
  - Increase the number of pages in the file by one. The new last page should be filled with zero bytes.
- `ensureCapacity`
  - If the file has less than `numberOfPages` pages then increase the size to `numberOfPages`.

2.4. **Return codes.** The header file `dberror.h` defines several error codes as macros. As you may have noticed, the storage manager functions all return an `RC` value. This value should indicate whether an operation was successful and if not what type of error occurred. If a method call is successful, the function should return `RC_OK`. The `printError` function can be used to output an error message based on a return code and the message stored in global variable `RC_message` (implemented in `dberror.c`).

## 3. Source Code Structure

You source code directories should be structured as follows:

- Put all source files in a folder `assign1` in your git repository
- This folder should contain at least
  - the provided header and `C` files
  - a make file for building your code `Makefile`. This makefile should create a binary from `test_assign1` from `test_assign1_1.c` which requires `dberror.c` and all your `C` files implementing the `storage_mgr.h` interface
  - a bunch of `*.c` and `*.h` files implementing the storage manager
  - `README.txt`: A text file that shortly describes your solution

  Example, the structure may look like that:

```
git
        assign1
                README.txt
                dberror.c
                dberror.h
                storage_mgr.c
                storage_mgr.h
                test_assign1_1.c
                test_helper.h
                Makefile
```

## 4. Test cases

We have provided a few test case in `test_assign1_1.c`. You makefile should create an executable `test_assign1` from this `C` file. You are encouraged to write additional tests. Make use of existing debugging and memory checking tools. However, usually at some point you will have to debug an error. See the `Programming Assignment: Organization` file information about debugging