Advanced Database Organization – Spring 2025

CS 525

Assignment 1: Storage Manager

Due: Monday, February 3rd 2024

Team Members

- Kori Kamoljonov - A20594489 – 33.3% contributed
- Chaitanya Datta Maddukuri - A20568393 - 33.3% Contributed
- Vamshi Krishna Cheeti - A20582646 - 33.4% Contributed

1.Task

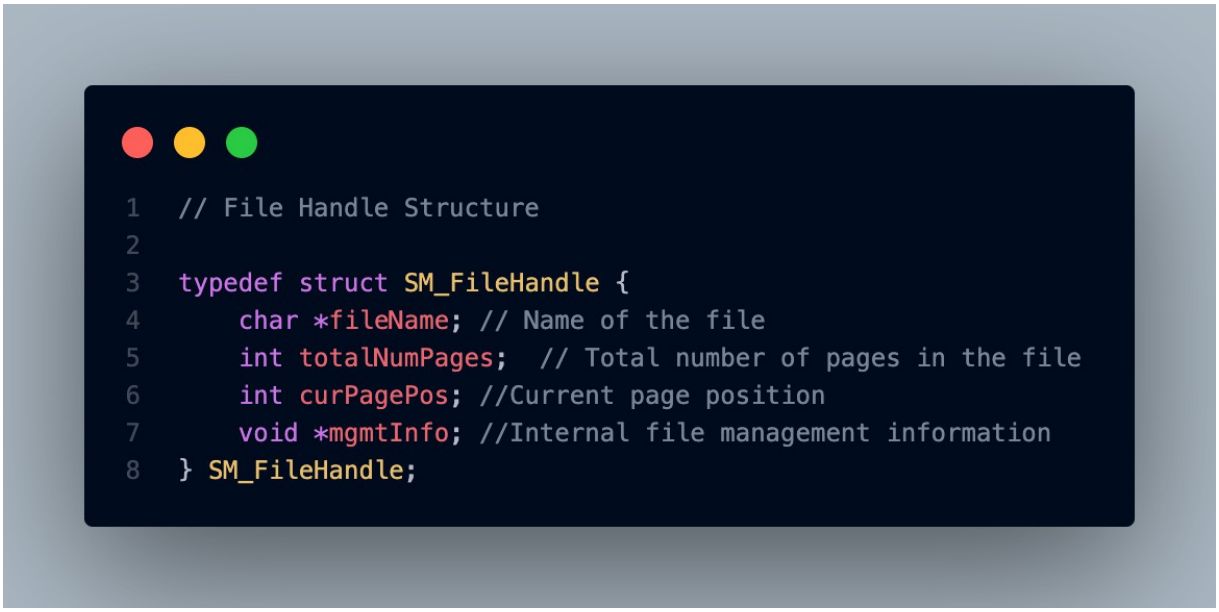
The Storage Manager provides a low-level interface for managing a simple page-based file system. It includes functionality for creating, opening, closing, and deleting page files, as well as reading and writing blocks of data within those files. The Storage Manager ensures proper page alignment and allows for dynamically extending files as needed.

2. Interface

```c
#ifndef STORAGE_MGR_H
#define STORAGE_MGR_H

#include "dberror.h"
#include <stdio.h>

// File Handle Structure

typedef struct SM_FileHandle {
    char *fileName; // Name of the file
    int totalNumPages;  // Total number of pages in the file
    int curPagePos; //Current page position
    void *mgmtInfo; //Internal file management information
} SM_FileHandle;

typedef char *SM_PageHandle;

// Storage Manager Function

void initStorageManager(void);

// File Handling
RC createPageFile(char *fileName);
RC openPageFile(char *fileName, SM_FileHandle *fHandle);
RC closePageFile(SM_FileHandle *fHandle);
RC destroyPageFile(char *fileName);

// Reading Blocks
RC readBlock(int pageNum, SM_FileHandle *fHandle, SM_PageHandle memPage);
int getBlockPos(SM_FileHandle *fHandle);
RC readFirstBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);
RC readPreviousBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);
RC readCurrentBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);
RC readNextBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);
RC readLastBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);

// Writing Blocks
RC writeBlock(int pageNum, SM_FileHandle *fHandle, SM_PageHandle memPage);
RC writeCurrentBlock(SM_FileHandle *fHandle, SM_PageHandle memPage);
RC appendEmptyBlock(SM_FileHandle *fHandle);
RC ensureCapacity(int numberOfPages, SM_FileHandle *fHandle);

#endif

```

2.1. **Data Structures** The interface relies on two key data structures for managing file and page information.

- **SM_FileHandle:**
    - Represents a file handle for a page file, which stores metadata such as the file name, total number of pages, the current page position, and internal management information required for handling file operations.



```c
// File Handle Structure

typedef struct SM_FileHandle {
    char *fileName; // Name of the file
    int totalNumPages;  // Total number of pages in the file
    int curPagePos; //Current page position
    void *mgmtInfo; //Internal file management information
} SM_FileHandle;
```

- **SM_PageHandle:**
    - Represents a pointer to a page-sized memory block used for reading and writing operations. This pointer provides direct access to data stored within a specific page of a file, enabling modifications and retrieval of page contents efficiently.

```c
typedef char *SM_PageHandle;
```

2.2. File-RelatedMethods

- **initStorageManager:**
  - Initializes the storage manager. This function currently prints a message indicating initialization.

- **createPageFile:**
  - Creates a new page file with one empty page initialized to zero bytes.

  Parameters: fileName (name of the file to create)

  Returns: RC_OK on success, RC_FILE_NOT_FOUND if creation fails.

- **openPageFile:**
  - Opens an existing page file for reading and writing operations. This function performs the following steps:
  - Attempts to open the specified file in read-write mode.
  - Sets up the SM_FileHandle structure with the file name, total number of pages (calculated based on the file size), and initializes the current page position to the beginning of the file (page 0).
  - Links the internal file pointer to the mgmtInfo field in the file handle for subsequent operations.

  Parameters:
  - fileName (name of the file to open).
  - fHandle (file handle to initialize with file details).

  Returns:
  - RC_OK on success.
  - RC_FILE_NOT_FOUND if the file cannot be opened or does not exist.

- **closePageFile:**
  - Closes an open page file. This function performs the following actions:
  - Closes the file associated with the mgmtInfo field in the SM_FileHandle.
  - Clears any internal pointers or resources allocated to the file handle, ensuring data integrity before closing.

  Parameters:
  - fHandle (file handle of the file to close).

  Returns:
  - RC_OK on success.

- **destroyPageFile:**

  Deletes a page file from disk. This function permanently removes the file from the file system, ensuring no residual data remains.

  Uses the standard file removal function to delete the file.

  Parameters:

  1. fileName (name of the file to delete).

  Returns:

  2. RC_OK on success.
  3. RC_FILE_NOT_FOUND if the file does not exist or cannot be deleted.

- **Read Methods**

  - **readBlock:**
    - Reads a specific block from the file into memory. This operation:
    - Moves the file pointer to the position corresponding to the specified page number.
    - Reads PAGE_SIZE bytes from the file into the provided memory buffer (memPage).
    - Updates the current page position in the SM_FileHandle structure.

    Parameters:

    - pageNum (page number to read).
    - fHandle (file handle of the file).
    - memPage (buffer to store the read data).

    Returns:

    - RC_OK on success.
    - Appropriate error codes if the operation fails.

  - **readFirstBlock:**

    - Reads the first block (page 0) of the file. Internally, this calls readBlock with page number 0.

    Returns:

    - RC_OK on success.

- **readPreviousBlock:**
    - Reads the block before the current block in the file.This operation ensures the requested page is within valid boundaries.
    - Internally calls readBlock with the current page position minus one.

  Returns:

  4. RC_OK on success.
  5. Error codes if the operation fails.

- **readCurrentBlock:**
    - Reads the block at the current page position in the file.Internally calls readBlock with the current page number from the SM_FileHandle.

  Returns:

  6. RC_OK on success.

- **readNextBlock:**
    - Reads the next block after the current block in the file.
    - Ensures the requested page does not exceed the total number of pages in the file.
    - Internally calls readBlock with the current page position plus one.

  Returns:

  7. RC_OK on success.

- **readLastBlock:**
    - Reads the last block (final page) of the file. Internally calls readBlock with the last page number (total pages minus one).

  Returns:

  - RC_OK on success.

2.4. Write Methods

- **writeBlock:**
  - Writes data to a specific page in the file. This function:
  - Moves the file pointer to the position corresponding to the specified page number.
  - Writes PAGE_SIZE bytes from the provided memory buffer (memPage) into the file.
  - Updates the current page position in the SM_FileHandle structure.

  Parameters:
  - pageNum (page number to write to).
  - fHandle (file handle of the file).
  - memPage (buffer containing data to write).

  Returns:
  - RC_OK on success.
  - RC_READ_NON_EXISTING_PAGE if the specified page does not exist

- **writeCurrentBlock:**
  - Writes data to the current page position of the file. Internally calls writeBlock with the current page number from the SM_FileHandle.

  Returns:
  - RC_OK on success.

- **appendEmptyBlock:**
  - Appends an empty block (filled with zero bytes) at the end of the file. This function:
  - Moves the file pointer to the end of the file.
  - Writes a new page of PAGE_SIZE bytes, all initialized to zero.
  - Increases the total page count in the SM_FileHandle.

  Returns:
  - RC_OK on success.

- **ensureCapacity:**

    - Ensures that the file has at least the specified number of pages. This function:

    - Checks the current total number of pages in the file.
    - Appends empty blocks until the file reaches the required capacity.

    Returns:

    - RC_OK on success.

2.5. Return Codes

- **Defined Return Codes (dberror.h):**

  RC_OK (0) - Operation successful
  RC_FILE_NOT_FOUND (-1) - File not found
  RC_FILE_HANDLE_NOT_INIT (-2) - File handle not initialized
  RC_READ_NON_EXISTING_PAGE (-3) - Attempt to read a non-existing page

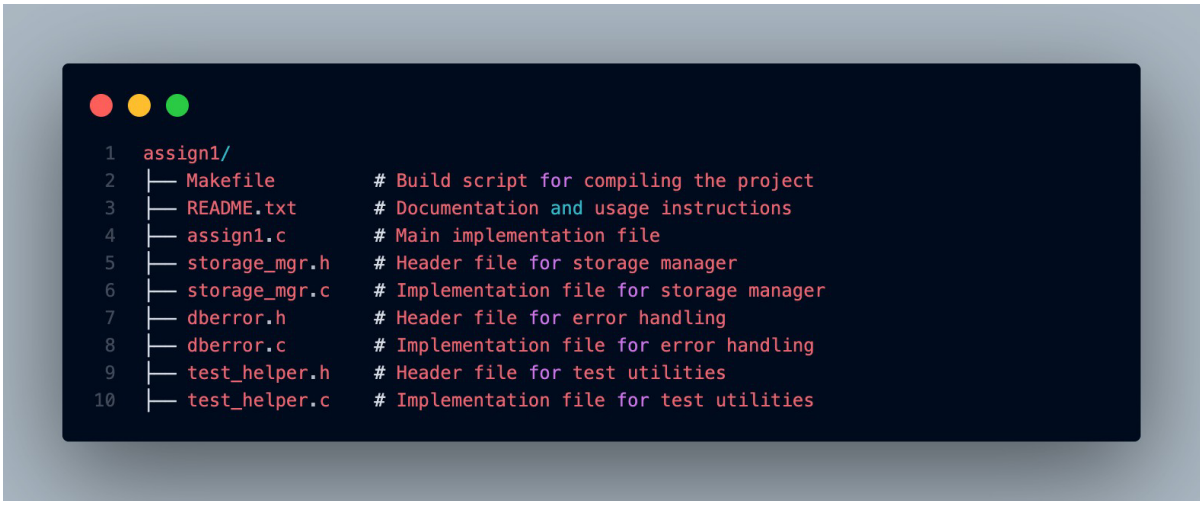
These return codes allow the storage manager to communicate errors effectively.

3. STORAGE CODE ORGANIZATION

- **Directory Structure**

  Your source code should be arranged in the following manner:
  Store all source files within a directory named assign1 inside your Git repository.This directory must contain the following components:
    - The provided header (.h) files and C (.c) files essential for implementation.
    - A Makefile to compile the source code. The Makefile should generate an executable named test from assign1.c, which depends on storage_mgr.h, dberror.c, and other necessary C source files.
    - Various .c and .h files implementing the storage manager functionality.
    - A README.txt file that provides a brief explanation of your implementation.

Example Directory Structure



This structure ensures all necessary components for compilation and execution are properly organized.

3.1. Implementation Notes

To compile and run the code:

1. **Setup the Environment:** Ensure you have a C compiler installed (e.g., GCC).
2. **Organize Files:** Place all source files (storage_mgr.c, dberror.c, test_assign1_1.c) and headers (storage_mgr.h, test_helper.h) in the same directory.
3. **Compile:** Run the **"make"** command in terminal to compile the code:
4. **Cleaning**: Remove any previously compiled files by running the "**make clean**" command in terminal
5. **Execute**: Execute the test cases with the **"./test_17"**command

4.Test Cases

A few test cases have been provided in test_assign1.c. Your Makefile should compile this file and generate an executable named test_assign1.You are encouraged to write additional test cases to ensure the robustness of your implementation.Utilize existing debugging and memory-checking tools to detect potential issues. However, manual debugging may still be necessary at times.