

# CS 564

## The Minirel I/O Layer

The lowest layer of the Minirel database systems is the I/O layer. This layer allows the upper level of the system to create/destroy files, allocate/deallocate pages within a file and to read and write pages of a file. This layer consists of two classes: a file (`class File`) and a database (`class DB`) class. Let us start with a description of the DB class. We will provide you with an implementation of this layer.

```
class DB {
public:
    DB();                // initialize open file table
    ~DB();               // clean up any remaining open files

    const Status createFile(const string & fileName) const;
                        // create a new file
    const Status destroyFile(const string & fileName) const;
                        // destroy a file release all
space
    const Status openFile(const string & fileName, File* & file);
                        // open a file
    const Status closeFile(File* file);
                        // close a file

private:
    OpenFileHashTbl    openFiles;
                        // hash table mapping files to file handles
};
```

The task of the DB class is to maintain a table of all files that are open. Each file corresponds to a relation (or an index) and is implemented as an OS (UNIX) file. If a file that has already been opened (possibly by another query), then the DB class detects this (by looking in the `openFiles` table) and just returns the file handle (a pointer of type `File*`) without actually opening the UNIX file again. In this way (and in other similar situations that you will encounter later) the DBMS can maintain tight control over the objects that it uses.

The following is a description of what each method in the DB class does.

### **const Status createFile(const string & fileName) const**

Creates a new UNIX file called `fileName` in the current working directory. Returns OK if no errors occurred, BADFILE if `fileName` is empty, FILEEXISTS if a file with this name already exists and UNIXERR if a Unix error occurred.

### **const Status destroyFile(const string & fileName) const**

Destroys the file named `fileName`. An open file cannot be destroyed. Returns OK if no errors occurred, BADFILE if `fileName` is empty, FILEOPEN if the file is open and UNIXERR if a Unix error occurred.

### **const Status openFile(const string & fileName, File\* & file)**

Opens the file named `fileName` and returns a pointer to the corresponding `File` object. First checks if the file is already open. If so, then a pointer to the file object that has already been created is returned and a reference count (inside the `File` object) is incremented. Otherwise the UNIX file is actually opened and used to initialize a new `File` object. The `fileName` and the pointer to this `File` object are inserted into the `openFiles` hash table. Returns OK if no errors occurred, BADFILE if `fileName` is empty and UNIXERR if a Unix error occurred.

#### **const Status closeFile(File \*file)**

This closes the file pointed to by `file`. If after decrementing, the reference count for the file becomes 0, the corresponding UNIX file is closed, the entry in the `openFiles` table removed and the file object is deleted. Returns OK if no errors occurred, BADFILEPTR if file is NULL and UNIXERR if a Unix error occurred.

```
class File {
    friend DB;

public:
    const Status allocatePage(int& pageNo);
                                // allocate a new page
    const Status disposePage(const int pageNo);
                                // release space for the page
    const Status readPage(const int pageNo,
                          Page* pagePtr) const;
                                // read page from file
    const Status writePage(const int pageNo,
                          const Page* pagePtr);
                                // write page to file
    const Status getFirstPage(int& pageNo) const;
                                // return pageNo of first page

private:
    File(const string & fname);           // initialize file object
    ~File();                             // deallocate file object

    static const Status create(const string & fileName);
    static const Status destroy(const string & fileName);

    const Status open();
    const Status close();

    const Status intread(const int pageNo,
                        Page* pagePtr) const;           // internal file read
    const Status intwrite(const int pageNo,
                        const Page* pagePtr);           // internal file write

    string fileName;           // The name of the file
    int openCnt;               // # times file has been opened
    int unixFile;              // unix file stream for file
};
```

The `File` class implements the DBMS abstraction of a `File` by providing a wrapper around the file system facilities provided by the OS. Many of the functions of this class (the private ones) are to be called only by the `DB` class and should not be called directly by the upper layers (which should use only the public methods). Hence the `DB` class is a *friend* of the `File` class. A (somewhat strange) example of this are the

constructor and destructor methods of the File class which are private because a File object can only be created and destroyed by a DB object. The main thing that you should remember is that a File should never be constructed, destructed, created, destroyed, opened or closed directly. These should only be done by calling the appropriate functions of the DB class. This is an example of how a well-designed DBMS is implemented in terms of *layers*.

We now describe the public methods of the File class. As a good object-oriented programmer, that is the only part of the File class that you should really be concerned with.

**const Status allocatePage(int & pageNo)**

Allocates a disk page for the current file and returns the page number in `pageNo`. Returns OK if no errors occurred and UNIXERR if a UNIX error occurred.

**const Status disposePage(const int pageNo)**

Releases the page `pageNo`. This page is added to the free list. Returns OK if no errors occurred, BADPAGENO if `pageNo` is not a valid page and UNIXERR if there is a Unix error.

**const Status readPage(const int pageNo, Page\* pagePtr) const**

Reads page `pageNo` from disk into the memory address specified by `pagePtr`. Returns OK if no errors occurred, BADPAGENO if `pageNo` is not a valid page, BADPAGEPTR if `pagePtr` is not a valid address and UNIXERR if there is a Unix error.

**const Status writePage(const int pageNo, const Page\* pagePtr) const**

Writes page `pageNo` from the address specified by `pagePtr` to disk. Returns OK if no errors occurred, BADPAGENO if `pageNo` is not a valid page, BADPAGEPTR if `pagePtr` is not a valid address and UNIXERR if there is a Unix error.

**const Status getFirstPage(int& pageNo) const**

Returns the (physical) page number of the first page of the file. This will be useful in the second part of the project. Returns OK if no errors occurred, UNIXERR if a Unix error occurred.

Both the DB and the File class can be found in the files `db.C` and `db.h`

## Error Handling

We have defined a class called Error in the files `error.h` and `error.C`. You can create an instance of this class (e.g. `Error err;`) and then print error messages from any method of any class by invoking `err.print(status)`. As you develop new classes you should add new error codes and messages in the Error class. Be sure to check the return codes of each function that you call and make sure that all functions return some status. ALL error messages should be printed using the Error class.

---