CMPS 181, Spring 2016, Project 1

Due Friday, April 22, 11:55pm on eCommons

Course Project 1 Environment and Submission

[Directions for Projects 2, 3 and 4 will be very similar.]

In project 1, you will implement a simple paged file (PF) manager. It builds up the basic file system required for continuing with projects 2, 3 and 4. The PF component provides classes and methods for managing files and pages in files. In addition, you will implement the first few operations for a record-based file manager (RBFM), which you will continue working on in part 2 of the project. The RBFM is built on top of the paged file system. This document aims at providing you with the necessary information required to start project 1.

Goal

The goal of project 1 is threefold:

- Getting familiar with a C/C++ development environment
- Implement a simple paged file system.
- Implement a few operations of a record-oriented (also known as tuple-oriented) file system.

The detailed description of the project is in the file:
CMPS181_Project1_Introduction.docx

Overview of Steps

1. Development environment
2. Download and deploy the codebase of Project 1
3. Finish the development of Project 1

Detailed Instructions

1. Development environment

- You may develop your code on any system you like using the steps below, but please test that it works on **unix.ucsc.edu**, which is where we will test it. It may simplify things a lot for you if you develop on that machine.

2. Download and deploy the codebase of Project 1

- **Download the codebase of Project 1**

Please download the codebase.zip file onto your own computer, and unzip the file.

- **Deploy the codebase**

Read the readme.txt under ./codebase/.

Go to the codebase, and modify the CODEROOT in makefile.inc properly.

Go to folder "rbf", and type in:

```
make clean
make
./rbftest
```

You will be able to see the output.

3. Finish the development of Project 1

We have seen the results of running the code in codebase. But since the implementation of methods is empty in codebase, you cannot manage any file yet. Please finish the implementation in pfm.cc, as well as the following methods in rbfm.cc (besides the constructor and destructor): 1) insertRecord. 2) readRecord. 3) printRecord. The remaining methods are not required for part 1 of the project; instead you will implement them as part of part 2 of the project. Please write your own test cases to test your code. You are responsible for anticipating other things that might go wrong that we haven't provided public tests for, just as you would be if you were building a DBMS in an industrial setting.

You may find these functions useful:
http://www.cplusplus.com/reference/clibrary/cstdio/

Submission Instructions

The following are requirements on your submission. **Points may be deducted if they are not followed.**

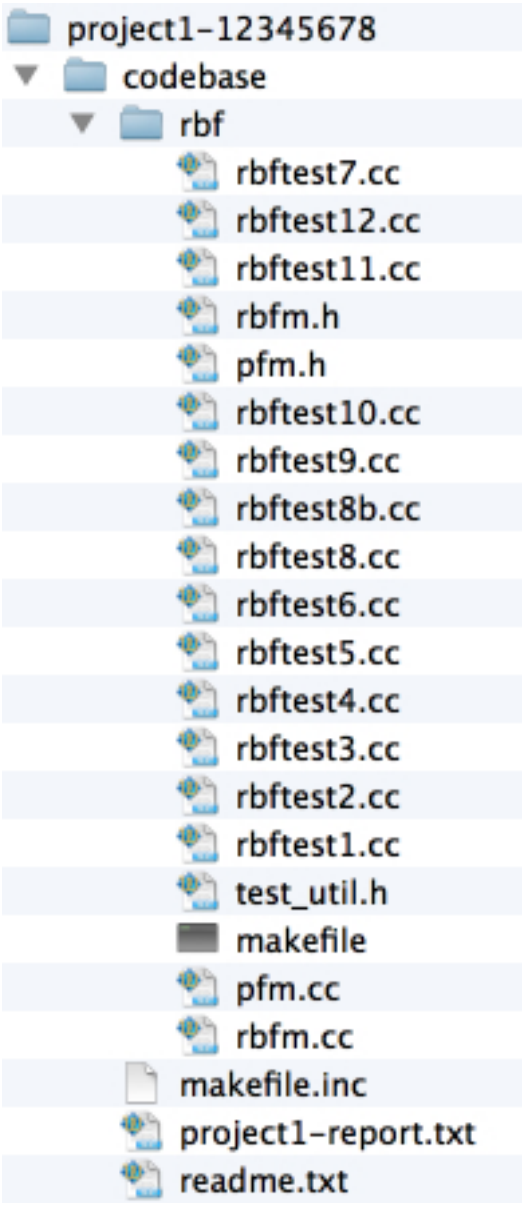
- **Suppress any debug messages that you put in your code. Only the original messages in the test cases should be printed.**
- Write a brief report that briefly describes the implementation of your paged file and record-based file systems. Please submit it as text, not in any other format, with the name project1-report.txt, using the project-report.txt file that's in codebase as the basis for your report.
- You need to submit the source code under the "rbf" folder. Make sure you do a "make clean" first, and do NOT include any useless files (such as binary files and data files). You should make sure your makefile runs properly.

- Please organize your project in the following directory hierarchy:

  project1-*studentID* / codebase / {rbf, makefile.inc, readme.txt, project1-report.txt} where the rbf folder includes your source code and the makefile. (e.g., project1-12345678 / codebase / {rbf, makefile.inc, readme.txt, project1-report.txt})

  [Since teams have multiple students on them, you may use the studentID of one of the students on your team. Only one member of a team should submit each project, preferably the same person for all of the projects.]

  Your folder structure should look like this picture:



- Compress project1-*studentID* into a SINGLE zip file, with the name "project1-*studentID*.zip" (e.g. project1-12345678.zip).
- Put test.sh and the zip file under the same directory. Run it to check whether your project can be properly unzipped and tested. (Use your own makefile.inc and the rbftest.cc when you are testing the script). If the script doesn't work correctly, it's most likely that your folder organization doesn't meet the requirement. Our grading will be automatically done by running the test script. The usage of the script is:

```
./test.sh ''project1-studentID''
```

- **IMPORTANT: Make sure the script works on unix.uccsc.edu, since the CMPS 181 Reader will use that machine to grade the assignment.**
- Upload the zip file "project1-*studentID*.zip" to eCommons. As noted, only one person on each team should do this.

Testing

Please use the code test inside the codebase to test your code. Note that these test cases will be used to grade your project partially, but we also have our own private test cases. This is by no means an exhaustive test suite. You should add more cases to this and test your code thoroughly!

Q & A

- **Q1**: In a case where Page 1 and 3 of the file are written in completely and Page 2 is partially filled and the user wants to append data on page 2. Now if the size of this data that he wants to write is more than the available space on Page 2, then what is the expected action to be taken. Do we just fit in whatever data we can and truncate the rest OR completely disallow the user to make such a write?
  A: AppendPage() always happens to the end of the file, so this scenario can't arise. The number of file bytes affected by each page operation is always PAGE_SIZE. The paged file system layer always deals in pages -- nothing more and nothing less.

- **Q2**: Is it fine if I do the file handling in C++ using the binary mode of read/write?
  A: You should use the binary mode.

- **Q3**: Why is the access specifier of the constructor and destructor of the class PagedFileManager set to be "protected"?
  A: The PagedFileManager is a singleton class, which means only one instance of PagedFileManager is allowed. You cannot instantiate the class by calling its constructor. Instead you should get an instance of the class by calling the

Instance() function of PagedFileManager. The Instance() function has been implemented for you in pfm.cc. The same applies to the RecordBasedFileManager.

- **Q4**: As for pages, if I understand correctly, the Read/Write/AppendPage functions are operating on these files and if you want to write 3rd page (page number: 2) of a file, you'd seek 8K bytes into the file and start writing the data. Is this correct or am I misunderstanding the concept of pages?
  A:
  - Read reads a page that has to exist
  - Append adds a page
  - Write overwrites a page that has to exist.
    To write to the 3rd page of a file, the file should already have at least 2 pages (page numbers: 0,1) that contain valid data. Then you can either append data to 3rd page if it doesn't exist, or overwrite the 3rd page if it already exists. Please do not leave "holes" in files by writing past EOF. We won't allow the case of appending garbage pages to happen.

- **Q5**: Are we supposed to work on heap files? In particular, for inserting a tuple, do we only have to consider about insertion the new tuple at the end of the last page? Or instead, we have to be able to support insertion in wherever free spaces among all the pages?
  A: You are supposed to insert the record on the first page with sufficient space available.

- **Q6**: What's the data format for insertTuple?
  A: The API format for insertRecord is as follows: Suppose you have five fields and their types are varchar(20), integer, varchar(20), real, and string. If a record is ("Tom", 25, "UCSantaCruz", 3.1415, 100), then the format of the record should be: [1 byte for the null-indicators for the fields: bit 00000000] [4 bytes for the length 3] [3 bytes for the string "Tom"] [4 bytes for the integer value 25] [4 bytes for the length 11] [11 bytes for the string "UCSantaCruz"] [4 bytes for the float value 3.1415] [4 bytes for the integer value 100]. Note that integer and real type fields do not have an associated length value in front of them; this is because each of these types always occupy 4 bytes.

  The first part of the input contains n bytes for passing the null information about each of the incoming record's fields. The value n can be calculated by using this formula: ceil(number of fields in a record / 8). For example, in this case, since there are 5 fields, the size of "n" can be calculated by ceil(5/8) = 1. If there are 20 fields, the size will be ceil(20/8) = 3. The left-most bit in the first byte corresponds to the first field. The right-most bit in the first byte corresponds to the eighth field. If there are more than eight fields, the left-most bit in the second byte corresponds to the ninth field and so on.

If a field value is NULL, the corresponding bit in the null bit vector will be set to 1. For example, if we have a record ("Tom", 25, NULL, NULL, 100) whose third attribute and fourth attribute are NULL, the first part contains 00110000 as the bit pattern in one byte. The actual byte representation will be: [1 byte for the null-indicators for the fields: 00110000] [4 bytes for the length 3] [3 bytes for the string "Tom"] [4 bytes for the integer value 25] [4 bytes for the integer value 100]. Note that there are no values to represent NULL values in the actual data. You MUST follow this API format!

NOTE: This API data format is just intended for passing data into the insertRecord(). This does not mean that the internal representation of your record should be the same as this format -- in fact, it almost certainly will not be! On-page record formatting options have been covered in Lecture 2 (starting at slide 8), and this material is also described in Section 9.6 of the textbook. Your project should make good choices based on what you learn in class.