Project 1 for CMPS 181: Implementing a Paged File Manager

Deadline: Thursday, April 18, 2019, 11:59 pm, on Canvas.

Introduction

In Project 1, you will implement a very 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 (RBF) manager (which you will continue working on in part 2 of the project) on top of your basic paged file system. This document aims at providing you with the necessary information required to start project 1.

For all four CMPS 181 projects, you will be working in teams of 2 or 3 people, where at least one person on each team is familiar with C++. We'll form teams during the first week of classes, and we hope that teams will last throughout the entire term. Only one person on a team needs to submit each assignment, but all team members are responsible for ensuring that the intended submission takes place on-time. Other team members will be listed in the project report file.

Goal

The goals of Project 1 are:

- Get (re)familiar with a C/C++ development environment.
- Implement a simple paged file system.
- Implement a few operations of a record-oriented (a.k.a. tuple-oriented) file system.

The detailed description of the project is in the CMPS181_Project1_Description file.

Overview of Steps

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

Detailed Instructions

1. Download and deploy the codebase of Project 1

- **Download the codebase of Project 1**

Please download the codebase.zip file onto your own computer. Unzip the file. Note that almost all of the files are tests that you won't have to modify.

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

./rbftest1

You will be able to see the output.

2. Finish the development of Project 1

We have seen the results of running the code in codebase. But, since the implementation of the methods is empty, you cannot manage any files 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 Project 1; instead you will implement them as part of Project 2. 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 will likely 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 short report to briefly describe the implementation (key design choices) of your paged file and record-based file system layers. Please supply your report as a text file named *project1_report.txt*, not a PDF, Word Document, or other non-text format. Do not use RTF format. Format of the report is described in the project1_report.txt file. **All team members** should be listed, not just the submitter.

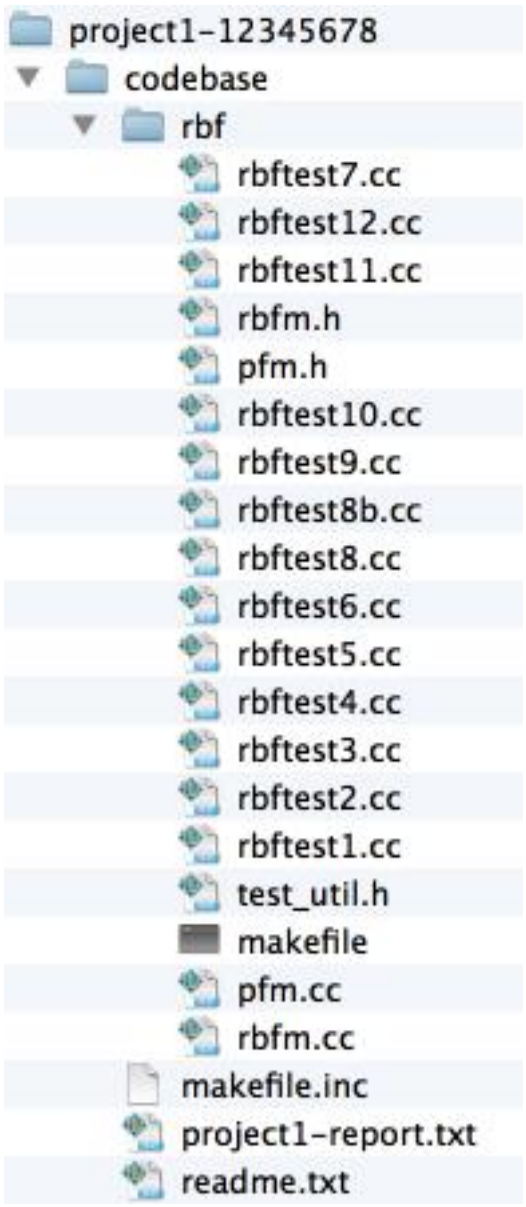
- Submit the source code under the "rbf" folder. Make sure you do a "**make clean**" first, and do NOT include any useless files (such as binary files and data files). You must make sure your makefile runs properly!

Please organize your project according to 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 should 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 named "project1-*studentID*.zip" (e.g., project1-12345678.zip).

- Put the test.sh file we provided and the zip file under the same directory. Run the test.sh script to verify that your project can be properly unzipped and tested (using your own makefile.inc and rbftest.cc when you are testing the script). If the script doesn't work correctly, it's likely that your folder organization doesn't meet the above requirements. Our grading will be done by automatically done by running the script. The usage of the script is:

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

- **IMPORTANT: You may develop your code on your own computer, or on any other system, but you need to make sure that your script works on unix.ucsc.edu, as testing and grading of your submission will be done there. Remember to suppress any debug messages that you put in your code. Only the original messages in the test cases should be printed.**

- Submit the zip file "project1-*studentID*.zip" to Canvas (e.g., project1-12345678.zip). Only one student on each team needs to do this, but your report should identify all team members, and your team number. That requires that you FTP your zip file from unix to your computer so that you can post it on Canvas. Please ask the course T.A., Saloni Rane, for help with that if needed.

Testing

Please test your code using the public test cases inside the codebase. Note that these public test cases will be used to grade your project, but for future projects, we may 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!

Grading Rubrics

Full Credit: 100 points

1. Follow the submission requirements (5%)
   - Code is structured as required (e.g., the directory structure).
   - Makefile works.
   - Code compiles and links correctly.

   Please make sure that your submission follows these requirements and can be unzipped and built by running the script we provided. If we can't run your project based on these instructions on unix.ucsc.edu, you'll lose points.

2. Project Report (10%)
   - Fill in each section of the report (project1_report.txt) in the codebase directory.
   - Clearly document the design of your project (internal record format and page format).

3. Implementation details (15%)
   - Implement all the required functionalities.
   - Implementation should follow the basic requirements of each function, e.g., that file should be created on disk.

4. Pass the provided test suite. (65%)
   - Each of the 13 tests is graded as pass/fail. There are no partial points for each test case.

5. Valgrind (5%)
   - You should not have any memory leaks. A valgrind check should not fail.

Q & A

- **Q1**: Can we make changes to the makefile provided?
  A: You have to make sure that we can build your submitted project in command line by using 'make'. If you don't add any new source files to the project, you shouldn't need many modifications to our makefile. Here is a tutorial for the 'make' tool. You can refer to it if you do decide to change the makefile.

- **Q2**: Consider a case where Page 3 of the file is full but Page 2 is partially filled and the user wants to append data. Now, if the size of the data that he or she wants to write is more than the available space on Page 2, 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 from making 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.

- **Q3**: Is it fine if I do the file handling in C++ using the binary mode of read/write?
  A: You should definitely use the binary mode!

- **Q4**: 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.

- **Q5**: As for pages, if I understand correctly, the Read/Write/AppendPage functions are operating on these files, and if you want to write the 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 must already exist.
  - "Append" adds a new page at the end of the file, increasing the number of pages in the file by one. Append is needed when you're inserting a record into a file, but there's no room for that record in any of the existing pages of the file.
  - "Write" overwrites a page that must already exist.
    To write to the 3rd page of a file, then if the 3rd page already exists, you can overwrite it. However, if the 3rd page doesn't exist and the file has 2 pages (page numbers: 0,1) that contain valid data, then you can Append a new page (page number 2), and then Write to that new 3rd page.

    Please do not leave "holes" in a file by writing past EOF. We won't allow you to append garbage pages that aren't in the file.

- **Q6**: Since I need to change the path of codebase in makefile.inc to test the project, do I need to change it back when I submit the zip file?
  A: No, you don't need to change back, but you need to instead make sure the path is **relative** so that the test.sh script will also work on another machine.

- **Q7**: When inserting a tuple, do we only have to consider insertion of the new tuple at the end of the last page? Or do we have to be able to support insertion in whatever free space may exist among all the current pages?
  A: You should first try to insert the record on the last (currently existing) page. If that fails, you should then try to find the first page with sufficient space available (e.g., looking from the beginning of the file). If none exists, then (and only then) should you append a new page to hold the new tuple.

- **Q8**: What's the data format for data being passed to insertRecord?
  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 occupies 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 will be covered in lecture, and your project should make good choices for what it does based on what you learn in class.)

- **Q9**: Can we assume that a record can fit on a page (i.e., the size of a record < the predefined page size)?
  A: Yes. You can assume that a record can fit on a page.