EECS 281 – Fall 2010

Project 3 – Output for Slow Search Engine

Assigned: Wednesday, October 13
Due: Monday, October 25 at 11:55:00 p.m.

1 Honor Code

As with any project/homework/exam in this class, you must abide by The Engineering Honor Code. Remember, all work submitted should only be work done by you and only you. If you are unsure about the level of allowed collaboration, please consult Professor Jagadish, the GSI, or the IAs.

2 Background

Let's say that your file indexer in Project 2 is really slow, or you're searching through a very large directory, and you don't want to have to wait for it to finish before you see some results. One solution is to take advantage of running multiple processes at once.

In this project, you will simulate part of what a computer might do if you have one process run an intermediate version of the Project 2 indexer, one process get user input, and one process keep track of the results and display them to the screen when asked.

3 Specification

The intermediate indexer process will give you a series of file names, combined with their relevances. There is no particular order in which they will appear. Your job will be to maintain these file names and relevances in a priority queue, and give the user the most relevant result whenever the user process requests a result.

You will run your program with one of the commands

```
./P3Search input.txt
./P3Search
```

Where `input.txt` is the file containing the inputs from the intermediate indexer process and the user input process. (See below) If the file name is omitted, then the input will be read from the console.

4 Input/Output Formatting

The input will be given to you either in the form of a file or from the console. Each line of input will contain either a user process request or an intermediate indexer result.

The intermediate indexer results will be formatted as follows

```
filename relevance
```

Where `filename` is the name of a file, and `relevance` is its relevance.

User requests will be preceded by a `$`, and will

- `print` - Display the most relevant current result and remove it from the priority queue.
- `clear` - Empty the priority queue and start over.
- `save` - Save the current version of the priority queue.
- `revert` - Return to the most recently saved version of the priority queue.
- `history` - Display a list of the changes made since the last save.
- `quit` - Quit.

Sample input might look like

```
file1.txt 45
$ print
file2.txt 13
file3.txt 68
$ print
$ clear
file4.txt 9
$ print
$ print
file5.txt 34
$ print
$ quit
```

The output should be formatted the same as the input. For every request, the output process should remove and display the current most relevant result in the same format:

```
cout << filename << " " << relevance << "\n";
```

The user might request a result when the priority queue is empty. In that case, you should store that request until a result becomes available. If there are any requests stored when the user clears or quits, then for every stored request you should respond with the message:

```
cout << "No results found.\n";
```

This means that
The input file above would then produce the output

```
file1.txt 45
file3.txt 68
file4.txt 9
file5.txt 34
No results found.
```

5 Sorting

The file in the priority queue with the highest relevance should be displayed first. If two files in the priority have the same relevance, then the one that comes earlier in the dictionary (as determined by the C library function `strcmp`) should be displayed first.

Thus the input

```
n.txt 42
q.txt 41
h.txt 42
r.txt 42
$ print
$ print
$ print
$ print
$ quit
```

Would yield the output

```
h.txt 42
n.txt 42
r.txt 42
q.txt 41
```

6 Save, Revert, and History

When the user enters the command `save`, your priority queue should make its current state a checkpoint. If the user later enters `revert`, then the priority

queue should change back to the checkpoint created by the last `save`. If the user enters `history`, then it should display a list of the changes made to the priority queue since the last `save`. (See below)

The commands `save`, `revert`, and `history` depend on your priority queue to efficiently save a history of changes made to the priority queue. You may assume that the `save`, `revert`, and `history` commands will be used very often. This means that simply copying the data in the priority queue will not be efficient enough as a `save` mechanism, especially since you need to efficiently print the history. Therefore, you should find a way to save the revision history in way that is efficient to display and to undo.

Whenever the user enters a command `history`, your program should display a history of all changes made to the priority queue since the last `save`. This should be displayed as follows:

```
cout << action << " file " << filename << " " << relevance << "\n";
```

Where `action` is either the string `"Added"` or `"Removed"`.

7 Clear

When the user enters `clear`, then the priority queue should erase all of the file results that it has stored. It should also erase all of the priority queue's history. The input

```
$ clear
$ history
```

should yield no output.

8 Priority Queue

The priority queue is a huge part of this project. A heap forms a very nice priority queue, so you will be using a heap as the main data structure for this project.

You will implement the heap on top of a complete binary tree class, which will be provided to you. The interface is given in the file `complete_binary_tree.h`, and the implementation is in `complete_binary_tree.cpp`. However, there are some problems with the solution we provide, which are discussed in the next two sections.

9 Buggy Tree

The implementation code provided in `complete_binary_tree.cpp` will cause a segmentation fault when combined with the test case `seg_fault.cpp`.

What you must do for this part:

- Use GDB to find the source of the error. Put the output from GDB into a file called `gdb.txt`.
- Fix the error(s) and test your code to make sure that it works. Continue to debug using GDB.

10 Slow Tree

The code that we have provided you in `complete_binary_tree.cpp` is not as efficient as it could be. Once you have completed most of your project, you should be able to use a profiler to figure out what parts of your code are inefficient.

What you must do for this part:

- Use GProf to determine where your program spends most of its time. Print the results to `gprof.txt`. You will submit this file.
- Fix `complete_binary_tree.cpp` to make that part or parts of your code more efficient. You will submit your final version of `complete_binary_tree.cpp`.

11 Error Checking

There are two types of errors, user errors and indexer errors. For user errors, your program should print an error message to the screen and continue execution. For indexer errors, your program should quit as soon as it sees one.

- **User Errors to Check For**: The following errors should be noticed, but are not fatal errors. When encountered, an error message should be printed, and then your program should continue execution starting at the next line.
  - The user should not try to enter any commands other than `print`, `clear`, `save`, `revert`, `history`, or `quit`. If the user does enter any command other than those, print the following message and continue execution.

    ```
    cout << "Error: Invalid command " << command << ".\n";
    ```
- **User Errors not to Check For**: You may assume that the following errors will not occur.

- A line with a user command will always be formatted as

  ```
  $ command
  ```

  where `command` is one word consisting only of alphanumeric characters. There will be no other input on that line.

- You may assume that the user will enter a `quit` command. However, you may **not** assume that there is no user input *after* that. If there is, then you should simply ignore it. You should not display anything prompted for after a `quit`. This includes errors. The input below should produce no output, because the user quits before any errors occur.

  ```
  $ quit
  $ invalidcommand!
  file92.txt -45
  ```

- **Indexer Errors to Check For**: These errors are fatal errors. If any of these errors are encountered,

  - The intermediate indexer process should not try to give your program the same file twice. Try to come up with an efficient way of doing this for large numbers of files. If this happens, print the following error to the user and exit with code 1.

    ```
    cerr << "Fatal error: Indexer process found file "
         << filename << " twice.\n";
    ```

  - The relevance that the intermediate indexer displays should be a positive integer. If it is not (that is, 0, negative, or not an integer), then display the following error and exit with code 1.

    ```
    cerr << "Fatal error: Invalid relevance for file "
         << filename << ".\n";
    ```

12 Sample Output

Sample #1:

Input:

```
file1.txt 1
file10.txt 10
file4.txt 4
$ print
file5.txt 5
file9.txt 9
$ history
$ clear
$ history
$ print
$ quit
```

Output:

```
file10.txt 10
Added file file1.txt 1
Added file file10.txt 10
Added file file4.txt 4
Removed file file10.txt 10
Added file file5.txt 5
Added file file9.txt 9
No results found.
```

Sample #2:

Input:

```
file1.txt 1
file10.txt 10
file3.txt 3
file4.txt 4
file7.txt 7
$ save
$ print
$ print
$ history
$ revert
$ history
$ print
$ print
$ quit
```

Output:

```
file10.txt 10
file7.txt 7
Removed file file10.txt 10
Removed file file7.txt 7
file10.txt 10
file7.txt 7
```

You will find this project much easier once you understand why these two input files produce the output that they do.

13 File Separation and Makefile

Division of code into separate files that perform different tasks is an essential part of good coding style. For instance, you should have at least one file for each class that you make.

For this project, it is required that you use at least one header file and two `.cpp` files. Because each of you will separate your code differently and use different file names, we also require that you include a makefile, called '`Makefile`' to make the compilation process efficient.

Your makefile must have the following properties:

- It must create a working executable called '`P3Search`'.
- It must create the correct object (`.o`) files.
- When '`make clean`' is run, all object files and the '`P3Search`' executable are removed.

14 SVN Repository

You are strongly recommended to use an SVN repository for this project. If you ever find yourself wishing you could go back in time and undo an edit, you will find that SVN is very helpful.

15 How To Submit Your Project

You can start making submissions to the autograder once you think your program works. The autograder will test your program on its own set of test cases and then report the results back to you. The content of the test cases will not be available for you to see. We highly recommend you also write your own test cases and test your program. Keep in mind that the autograder is on a Red Hat Linux machine (same as the CAEN machines). Although all the code should compile and work on any system, sometimes weird things happen. Thus, we

recommend that you test and compile (and develop) your code in Linux on the CAEN machines and not in other platforms. Do all of your project work with all needed files in some directory other than your home directory. This will be your "submit directory". When you are ready to submit your final version, make sure:

- You have deleted all .o files and all executable(s) and that typing "`make clean`" accomplishes this.
- Your makefile is called `Makefile`.
- Typing the command "`make`" will result in an executable called `P3Search`.
- You have no other files besides what you need.
- Your code compiles and runs correction using version 4.1.2 of the g++ compiler. This is the standard version on the CAEN Linux systems (such as login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 4.1.2 running on Linux.

Turn in the following files:

- all your `.h` and `.cpp` files, including `complete_binary_tree.cpp`
- `Makefile`
- `gprof.txt`
- `gdb.txt`

You must prepare a compressed tar archive (a `.tar.gz` file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:

```
tar czf ../submission.tar.gz *
```

This will prepare a suitable file in the parent of your working directory. Submit your tar ball directly to the autograder at: `https://grader8.eecs.umich.edu`.

You may submit up to **3** times a calendar day to autograder for feedback. For this purpose, a calendar day begins at midnight (12:00:00 am) and ends at 11:59:59 pm Ann Arbor local time. However, there might be delay over the internet between the time you submit and the time the autograder receive it. We recommend that you upload your last submission of the day before 11:55:00 pm. It's unlikely to have a delay larger than five minutes. If you submit more than 3 times, the autograder will acknowledge that you have submitted but will not provide you with feedback. If you do not receive feedback within 30 minutes, do not panic. This delay depends on the load on the autograder. It is likely to be greatest in the hours immediately before the deadline. If you have any problems with the autograder, email Scott Reed at reedscot@umich.edu. Thus, if possible, submit early!

16 Coding Style

The Google C++ Style Guide
(`http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`) is **HIGHLY** recommended for reviewing coding style.

Your coding style will be graded on what warnings the autograder gives you (and how many) and a hand-grading of the code. Some suggestions to improve your score:

- Program Design
  - Use classes!
  - Don't lump everything into one class or one file. If ideas are clearly separate, then put them in separate files.
  - Make use of constructors and destructors (like when deallocating memory).
  - Use multiple .cpp and .h files as appropriate. For example, class declarations should go into the header files (.h) and their subroutines should go into standard code files (.cpp).
  - In general, if your code seems sloppy to you, then it probably is. Try and write code that you and others can understand.
- Use informative and concise comments:
  - Explain what each class does and what each of its member functions do.
  - A 1-2 line explanation is useful for non-standard coding syntax.
  - A useless comment is worse than no comment at all. For example,

    ```
    int temp; // creates temp variable
    ```

    is not helpful.
- Formatting your code:
  - Use indentations when you have if/for/while/general code blocks.
  - Avoid going past 80 characters per line.
- Variable names:
  - Use reasonable and informative names.
  - Using variables "`i`" and "`j`" are acceptable for LCVs (loop control variables), but consider using other names for long-term data.
- Code Reuse: Do not have duplicate code. It is expected that you will organize your project logically and create many helper functions so that you do not have different segments of code doing the same things.

17 Runtime Evaluation

This project will be graded somewhat on speed. We will post benchmark times for selected test cases and will deduct points if you do not meet these times. Specifically, you will lose 10% of the grade for that test case if your program takes between 1 and 3 times as long as the benchmark. You will lose 20% if it is between 3 and 10 times worse, 33% if between 10 and 100 times worse, and all credit if greater than 100 times worse. These points are deducted per test case but are not calculated until after the project is due.

You may use compiler optimization (the O3 flag, for example) in this project.

18 Grading

The grading for Project 2 is out of 100 points:

- 80 points - Test Cases
- 5 points - `gdb.txt`
- 5 points - `gprof.txt`
- 10 points - `complete_binary_tree.cpp`

19 Submission Time and Late Policy

The project is officially due at 11:55:00 on Monday, Oct 25, 2010. You have two free late days in total for this semester. An additional late day will be charged at the cost of 1% of your total course grade. (Each project is 8% of your total courses grade). You can use at most 2 late days on each project regardless of whether they are free late days or charged late days.

20 Hints and Advice

- **Start early!** The earlier you start, the more time you have to write your code, ask questions, and debug.
- **Break the program into its components.** There are several modules that should be disaggregated and coded separately. Write down and determine for yourself what functions, classes, and features you will need.
- Create complex test cases before you start.
- Debug intelligently. Use GDB when useful, and informative print statements when necessary.
- Good Luck!

21 Revisions

Revision 1: October 13 at 10:00 AM.

- Updated grading scheme.
- Updated due date in section on late policy.