CS 214: Systems Programming, Fall 2012
Programming Assignment 4: Search

1 Introduction

In this assignment, you will put everything that you have done together into a simple search tool. For now, your search tool will look much like the `grep` utility.

Your task is to implement a search tool that will load an inverted index produced by your indexer into memory and use it to answer users' search queries. Using the same example from the indexer assignment, if you are given the following set of files:

File Name	**File Content**
boo	A dog name name Boo
baa	A cat name Baa

you would use your indexer to generate the following inverted index and save it to an index file:

"a" → ("boo", 1), ("baa", 1)
"baa" → ("baa", 1)
"boo" → ("boo", 1)
"cat" → ("baa", 1)
"dog" → ("boo", 1)
"name" → ("boo", 2), ("baa", 1)

When you run your search tool, it should read the content of the index file into memory. Then, it should continuously poll for user queries and output the names of the files that contain the search terms. For example, if the user gives the query `dog`, your search tool should output `boo`. If the user gives the query `name`, your search tool should output `boo, baa`.

2 Implementation

Your program must support the following invocation interface:

```
search <inverted-index file name>
```

The first (and only) argument, <`inverted-index file name`>, gives the name of an index file that your search tool should read into memory. For now, you may assume that the entire index file will fit into memory (you will relax this assumption in a future assignment). You should design an in-memory data structure to make the search efficient. (**Hint:** To ease the next assignment, you should design a data structure where inverted lists (e.g., "dog" → ("boo", 1)) can be easily inserted

and removed. This is because you will need to keep a cache of parts of your index in memory when the entire index cannot fit in memory.)

Once `search` has successfully read and process the index file, it should go into a loop asking for queries. It should be able to respond to at least two commands:

sa <term> ...: search for files containing the given terms. A query may contain 1 or more terms. If there are more than 1 term, the search tool should return only files that contain *all* terms in the query. (The query is a "logical and" of all the given terms.)

so <term> ...: search for files containing the given terms. A query may contain 1 or more terms. If there are more than 1 term, the search tool should return any file that contains *any subset* of the terms in the query. (The query is a "logical or" of all the given terms.)

q: the search tool should gracefully shut itself down.

Here is an example session of what using your tool should look like using an index of the `boo` an `baa` files above:

```
$ search my-index.idx
search> sa a dog
boo
search> so a dog
boo
baa
search> so cat dog
boo
baa
search> q
$
```

The `search>` is the prompt of my hypothetical search tool.

As in the last assignment, you should carefully consider all possible exception cases, outline a strategy to deal with them, and implement your strategy. Also, as in the last assignment, your implementation should be divided into modules as it makes sense to do so.

New requirements: You should use valgrind to ensure that your program does not have any memory leaks. If running valgrind with the option `--leak-check=full` on your program leads to any complaints, you must explain why the complaints remain.

In addition, you must use either valgrind or gprof to study the performance of your program. For this assignment, you only need to describe what you learned from the profiling. That is, you do not need to do any optimization based on the profiling.

Notes:

1. For those really behind, this is a chance for a reset. We will provide an indexer that you may use, together with a specification of the format of the index files that it generates. Thus, you can start fresh with your search tool.

If you choose to use our indexer (and are still using it at the end of the class), it will affect your final grade. That is, it will definitely constrain the highest grade that you can earn. However, this might be a better option than dropping or not passing the class.

The above implies that if you finish the search tool, you can still pass the class. This is absolutely true. However, the search tool will be more challenging to build than the indexer. So, if you want to take advantage of the reset opportunity, you need to start working hard immediately!

2. The output format for the indexer specified in the last assignment has a couple of problems:

   (a) It is a little bit of a pain to handle file names that contain spaces when your search tool is reading in an index. You might want to change the format to ease this problem.

   (b) The format is **very** inefficient because a file name is kept as a string everywhere. You can optimize this by have a mapping from file names to unique integer IDs, store this mapping in your index, and then use the IDs in the term count records, as opposed to the file names themselves. You are not required to make this optimization but if you have time, it's a good thing to do.

3 What to turn in

A tarred gzipped file name pa4.tgz that contains a directory called pa4 with the following files in it:

- All the .h and .c files necessary to produce `index` and `search`. If you are using our indexer, then you should state this explicitly in your readme file.
- A `makefile` used to compile and produce `index` and `search`. It must have a target `clean` to prepare a fresh compilation of everything.
- A file called `testplan.txt` that contains a test plan for your search tool. You should include the example files/directories that you index to test your search tool, but keep these from being too large, please. (We might test your program with a very large data set though so don't skip testing your program for scalability. In your test plan, you should discuss the larger scale testing and the results, but you can skip including the data set).
- A `readme.pdf` file that describes the design of your search tool. The writeup should also discuss any complaints from valgrind for memory usage. Finally, the writeup should discuss what you learned from profiling your program (e.g., which function takes the most time and why). Note that starting in this assignment, you no longer need to include a big-O analysis.

As always, your grade will be based on:

- Correctness (how well your code is working),
- Quality of your design (did you use reasonable algorithms),
- Quality of your code (how well written your code is, including modularity and comments),

- Efficiency (of your implementation), and
- Testing thoroughness (quality of your test cases).