Assignment 3: A Simple Search Engine

Due Date: 11:59 pm Sunday, March 31, 2019
Worth: 10% of your final grade.

Introduction:

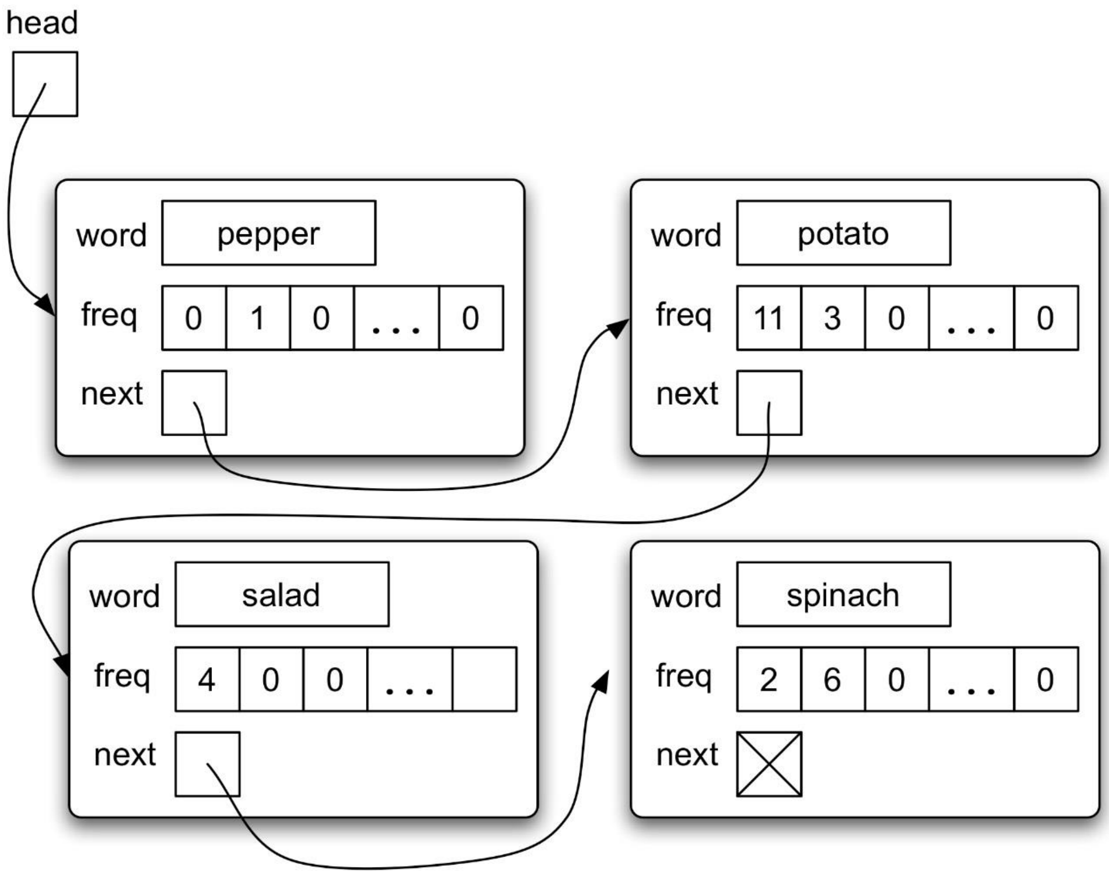
A search engine (like Google) has three main components: a crawler that finds and stores copies of files on the web, an indexer that creates a data structure that is efficient for searching, and a query engine that performs the searches requested by users. The query engine uses the indexes produced by the second component to identify documents that match the search term.

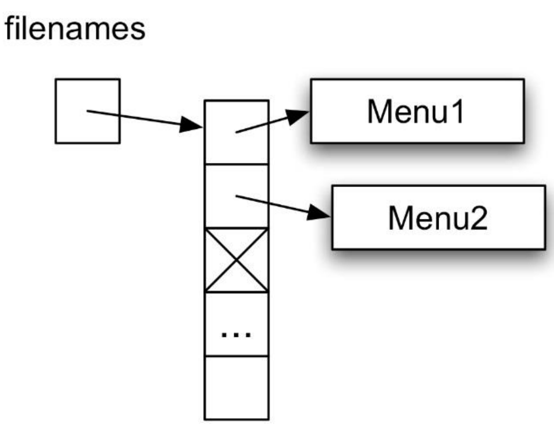
The goal of this assignment is to write a simple parallel query engine. We will assume that the indexes have already been created. For the purposes of this assignment, an index simply tracks word frequency in each document. If Google used a program with a single process to query all of the indexes to find the best matches for a search term, it would take a *very* long time to get the results. For this assignment, you will write a parallel program using `fork` and `pipes` to identify documents that match a search term. (However, you won't likely see any performance difference between using one process and many because the indexes we are giving you are so small.)

Index files

You won't be writing any of the code that builds the indexes themselves. We have given you starter code `[/courses/courses/cscb09w19/nizamnau/a3/]` that creates an index for a directory of files, and writes the two key data structures that represent an index to two files: `index` and `filenames`. Your program will use `read_list` to load an index from the above two files into memory, so it is useful to understand how the data structure works. Words and their frequencies are stored in an ordered linked list. The picture below shows what the linked list looks like.

Each list node contains three elements: the word, an array that stores the number of times the word has been seen in each file, and a pointer to the next element of the list. Another data structure (an array of strings) stores the name of each file that is indexed. The index of a file name corresponds to the index of the `freq` array in a list node. Storing the file names separately means that we don't need to store the name of each file many times.





In the diagram above, four words have been extracted from two files. The two files are called `Menu1` and `Menu2`. The linked list shows that the word `spinach` appears 2 times in the file `Menu1` and 6 times in the file `Menu2`. Similarly, the word `potato` appears 11 times in the file `Menu1` and 3 times in `Menu2`. The words in the linked list are in alphabetical order.

As part of the A3 starter code, we have provided a program called `indexer` that will build the linked list data structure and the string array with the file names and will store them in two files named `index` and `filenames`, respectively. The `indexer` program is implemented in the file `indexer.c` and will be automatically compiled by the provided `Makefile`. When run without arguments `indexer` will index all the files in the current working directory. It also supports a `-d` command line option that you can use to specify a directory where `indexer` should look for files to index (instead of using the current working directory).

We have also created several indexes that you can use for testing, and put them in

`/courses/courses/cscb09w19/nizamnau/a3/testing/big`

on mathlab. Each sub-directory in this folder contains a set of text files and the corresponding `index` and `filenames` files that were obtained from running the `indexer` on the text files in the directory.

The file `index` that stores the linked list data structure is in binary format, so you will not able to look at the file contents and read them. We provide a program called `printindex` (implemented in `printindex.c` in your starter code) that looks for two files named `index` and `filenames`, respectively, builds the index data structure based on these two files and prints the data structure in human readable format. The output of `printindex` will be one line for each word found in the text files covered by the index, followed by the frequency counts for this word for each of the text files.

We put executables for `indexer` and `printindex` in the `/courses/courses/cscb09w19/nizamnau/a3/testing`. For example, if you `cd` into the `big/dir8` directory and call `../../printindex` you will see the contents of the index for the text files in that directory (created based on the index and filenames files in the directory, which we generated for you using the indexer).

Task 1: Find a word in an index

Your first task is to write a function called `get_word` that looks up a given word (provided as an argument to the function) in an index (i.e. a linked list). Since you may not change `freq_list.c` or `freq_list.h`, this function should go in `worker.c`.

`get_word` will return an array of `FreqRecord`s for the word that the function looks up in the index. Each individual `FreqRecord` provides the number of occurrences of the word for a particular file. The array should only contain `FreqRecord`s for files that have at least one occurrence of the word. The definition of a `FreqRecord` struct is shown below and is included in `worker.h`.

```
#define PATHLENGTH 128

typedef struct {
        int freq;
        char filename[PATHLENGTH];
} FreqRecord;
```

The maximum number of valid entries in the array of `FreqRecords` will be the number of files covered by the index. You can assume that the number of files in the index is limited by `MAXFILES` and use this to allocate the right amount of space for your array. To indicate the end of the valid records, the last record will have a `freq` value of 0. If the word is not found in the index, `get_word` returns an array with one element where the `freq` value is 0.

Here is a function that you might use to print an array of records for testing your function:

```
void print_freq_records(FreqRecord *frp) {
        int i = 0;
        while(frp != NULL && frp[i].freq != 0) {
                printf("%d    %s\n", frp[i].freq, frp[i].filename);
                i++;
        }
}
```

After you write `get_word`, be sure to **test** it. Write a main program that calls it and runs it on several sample indexes before you move on. Commit your test file to your repository. We won't be marking it, but we may be checking it if some other part of your code does not work.

Task 2: Workers

`void run_worker(char *dirname, int in, int out)`

The `run_worker` function takes as an argument the name of the directory that contains the index files it will use. It also takes as arguments the file descriptors representing the read end (`in`) and the write end (`out`) of the pipe that connects it to the parent.

`run_worker` will first load the index into a data structure. It will then read one word at a time from the file descriptor `in` until the file descriptor is closed. When it reads a word from `in`, it will look for the word in the index, and write to the file descriptor `out` one `FreqRecord` for each file in which the word has a non-zero frequency. The reason for writing the full struct is to make each write a fixed size, so that each read can also be a fixed size.

We have given you the skeleton of a program in `queryone.c` that calls `run_worker` so that `run_worker` will read from stdin and write to stdout. This will allow you to test your `run_worker` function to be sure that it is working before integrating it with the parallel code in the next section.

Task 3: Now the fun part!

The final step is to parallelize the code so that one master process creates one worker process for each index. Write a program called `query` that takes one optional argument giving the name of a directory. If no argument is given, the current working directory is used. (Your `main` function will be in a file called `query.c`). You will probably find it useful to copy much of the code from `queryone.c` to help you get started.

For example, if we ran query `/courses/courses/cscb09w19/nizamnau/a3/testing/simple`, we would expect the directory `simple` to contain directories that each have an index. The number of subdirectories of the command line argument (or of the current directory, if a commandline argument is not provided) determines the number of processes created.

Each worker process is connected to the master process by two pipes, one for writing data to the master, and one for reading data from the master. The worker processes carry out the same operations as the `run_worker` you wrote (and tested) in the previous step. The master process reads one word at a time from standard input, sends the word to each of the worker processes, and reads the `FreqRecords` that the workers write to the master process. The master process collects all the `FreqRecord`s from the workers by reading one record at a time from each worker process, and builds one array of `FreqRecord`s, ordered by frequency. It prints this array to standard output, and then waits for the user to type another word on standard input. It continues until standard input is closed (using ^D on the command line). When standard input is closed, it closes the pipes to all the workers, and exits.

Details

Here is a high-level overview of the algorithm `query` will implement:

- Initialization (the same as `queryone`)
- Create one process for each subdirectory (of either the current working directory or the directory passed as a command line argument to the program)
- while(1)
  - read a word from stdin (it is okay to prompt the user)
  - using pipes, write the word to each worker process
  - while(workers still have data)
    - read one `FreqRecord` from each worker and add it to the master frequency array
  - print to standard output the frequency array in order from highest to lowest
- The master process will not terminate until all of the worker processes have terminated.

The master frequency array

You will only store the `MAXRECORDS` most frequent records. This means that you need to keep the array sorted, and once you have collected `MAXRECORDS` records, the least frequent records will be removed (or overwritten). This also means that you can allocate a fixed size array for this data. Any helper functions you write to help you manage this array should go in `worker.c`.

Reading and writing the pipes:

- The master process will be writing words to the child processes. How does the child process know when one word ends and the next word begins? The easiest way to solve this problem is to always write and read the same number of bytes. In other words, when the master process writes a word to a child, it should always write the same number of bytes. You can assume that a word is no bigger than 32 characters (see `MAXWORD` in `freq_list.h`). So the master process will write 32 bytes (including the null termination character), and the child process will always read 32 bytes.
- The `FreqRecords` have a fixed size, so the master process knows how to read one record at a time from a worker. The worker process notifies the master that it has no more records to send by sending a `FreqRecord` with a value of 0 for the `freq` field, and an empty string for the `filename` field.

- The master process will read one record at a time from each worker, so you need to keep track of the case when the master has read all of the records from a worker.

Error checking

Check all return values of system calls so that your program will not "crash". Your program should exit with a return code if it encounters and error that would not allow it to proceed.

What to hand in

Add and commit to your SVN repository in the `a3` directory all of the files that are needed to produce the programs `queryone` and `query`. When we run `make` in your `a3` directory, it should build the two programs (in addition to the helper programs we gave you) without any warnings.

Remember to run `svn add` on any source code files that you created. The `svn status` command allows you to confirm that all files have been added and committed to the repository.

Coding style and comments are just as important in CSCB09 as they were in previous courses. Use good variable names, appropriate functions, descriptive comments, indentation and blank lines. Remember that someone needs to read your code.

Please remember that if you submit code that does not compile, it will receive a grade of 0. The best way to avoid this potential problem is to write your code incrementally. For example, the starter code compiles and solves one small piece of the problem. Get a small piece working, commit it, and then move on to the next piece. This is a much better approach than writing a whole bunch of code and then spending a lot of time debugging it step by step.