Assignment 3: Processes

Due Nov 14, 2018 by 4pm **Points** 10 **Available** after Oct 24, 2018 at 12am

Assignment 3: A Simple Search Engine

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 you are interested in how more sophisticated indexes are created, you might consider taking CSC401.)

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 that creates an index for a directory of files, and writes the index to two files: `index` and `filenames`. We have also created several indexes that you can use for testing, and put them in `/u/csc209h/fall/pub/a3-2018` on teach.cs. The `index` file is binary so you won't be able to read it with a plain text editor.

Your program will use `read_list` (also provided in the starter code) to load an index from the two files into memory. 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.

A3.linkedlist.pdf

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 `filenames` and `freq` arrays are parallel. In other words, the position of a file name corresponds to the position of the count for that file in 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.

Task 0: Getting and reading the starter code

Visit the A3 page in MarkUs to trigger the addition of the starter code to your 209 git repo. Then do a pull on your repo. Read through the rest of this handout carefully and then read over the starter code for the assignment. Pay particular attention to the data structures and file formats and which files are not plain text.

As you mature in your programming ability, we are giving you fewer explicit instructions about how to accomplish some tasks. Like in industry, the interfaces are still carefully specified because it is critical that your new code fits with existing code written by other people. You may not change `freq_list.c` or `freq_list.h`.

Task 1: Finding a word in an index

Your first task is to write a function called `get_word` that looks up a word in an index. Since you may not change `freq_list.c` or `freq_list.h`, this function should go in `worker.c`. Do not change the signature.

The definition for a `FreqRecord` is as follows:

```
        #define PATHLENGTH 128

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

`get_word` will return an array of `FreqRecord` elements for the word. To indicate the end of the valid records, the last record will have a `freq` value of 0 and the `filename` set to an empty string. If the word is not found in the index, `get_word` returns an array with one element where the `freq` value is 0 and `filename` is an empty string.

This diagram shows the two possible arrays that could be returned by `get_word` if it were called on the word `spinach` using the index and the filenames from the previous example. The order of the elements (other than the sentinel value) of the array is up to you. Make sure you understand the behaviour of `get_word` by drawing the results of calling it on the words `pepper` and `salt` using the same index.

FreqRecordArray.pdf

Below is a function that you should use to print an array of `FreqRecord` elements. Notice that it does not take the length of the array as an argument. Make sure you understand why it still works.

```
	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 and add a target to your Makefile to build it.

Task 2: Workers

Next you should complete the `run_worker` function in `worker.c` with the following signature:

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

The `run_worker` function takes as an argument the path to a directory that contains the files (`index` and `filenames`) that 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. Then it will write one final `FreqRecord` with a frequency of zero and an empty string for the filename. 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. You will have to work out how to handle the output from queryone since it is not plain text. So simply displaying it on the terminal isn't going to work.

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 which we will call the *target directory*. If no argument is given, the current working directory is used as the target directory. Put your `main` function 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.

The target directory will contain subdirectories that each hold an index. The number of subdirectories of the target directory determines the number of processes created. You should ignore any regular files within the target directory (including an `index` or `filenames` file should they exist.) You only process the index files for the immediate subdirectories of target (not recursively) and your program should fail gracefully and report an error to standard error if one of the subdirectories does not contain either or both of the index files. You will see that much of this is handled already by the starter code.

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 (using the provided `print_freq_records` function), 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 the target directory
- while(1)
  - read a word from stdin (do **NOT** prompt the user)
  - using pipes, write the word to each worker process
  - while(workers still have data)
    - read one `FreqRecord` from a 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 replaced. (In the event of a tie, you may select either word to keep.) 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`.

Managing the workers

We have left some design decisions up to you including how to keep track of the workers within your query program and how to cycle through reading data from them. If you find it helpful to have a maximum number of workers, you should use the constant `MAXWORKERS` defined in `workers.h` Depending on your design, you may or may not need to use this maximum.

Reading and writing using 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 `FreqRecord` elements 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.

Error checking

Check all return values of system calls and exit with a non-zero return code if your program encounters an error that would not allow it to proceed.

Being A Good Citizen

The first assignment that uses `fork` can be a challenge because error-prone code can generate processes and leave them running indefinitely. You must not leave processes lying around on any of the machines you work on, or the machines will eventually become unusable. **Always**, before you log out, and frequently while you are working, use `ps aux | grep <username>` or `top` to check if you have any stray processes still running. Use `kill` to kill them.

If you notice that other students have processes that have been around for a long time, please post a note to the discussion board with their user names, and ask them politely to clean up their processes.

Finishing Off and Submitting

Because we will run automated tests on your program, it is important that you not print extra information to standard out. In particular, do not print a prompt asking the user for the next word. **Only** use the `printf` statements that we have provided in the starter code.

Commit and push to your 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 (without any argument), it must build the two programs (in addition to the helper programs we gave you) without any warnings. Do **NOT**commit any executables.