Washington State University Vancouver

Systems Programming - CS 360

Assignment 1

Instructor:
Ben McCamish

Overall Assignment - 100 points

Write a program (in C) targeted at the Linux platform which reads words from one or more files, and prints out a list of the most frequently occurring sequential pairs of words and the number of times they occurred, in decreasing order of occurrence.

Example: If a file contained the text *"This is a nasty assignment and it will take me two weeks to do this nasty assignment. I should probably start on it early. I suspect it will take 300-500 lines of code."*, the word pairs *"nasty assignment"*, *"it will"* and *"will take"* would each be detected as having 2 occurrences, while all the other sequential pairs have only one occurrence.

Marks are shown below. Any requirement marked with "(Required)" will result in a score of 0 if not implemented.

Program Interface (Required)

```
wordpairs <-count> fileName1 <fileName2> <fileName3> ...

Where: count is the integer number of word pairs to print out and fileNameN are
pathnames from which to read words. If no count argument is specified, ALL words
pairs are printed to stdout. (tokens enclosed in angular brackets are optional).
```

1 Specifications and Restrictions

- (10 points) All error output to be printed to stderr. If unrecoverable errors occur, the program exits with a non-zero exit code, otherwise it exits with a zero exit code.
- (40 points) All normal (expected) output (the list of word pairs and number of occurrences) are to be printed to stdout.
- A procedure will be supplied by the instructor which reads successive words from an open file descriptor on the website.
- (30 points) Use a hash table to store and count occurrences of sequences of words. Your hash table must keep track of how full it is and grow itself to a larger size (i.e. more buckets), as needed. Design and code your own data structures and procedures implementing a hash table for the purposes of this assignment. Use the technique known as separate chaining to implement your hash table buckets.
- (10 points) The hash table must evaluate some measure of its search performance. This can be average number of collisions, maximum collisions or some other reasonable measure which you will document in your comments. When this measure exceeds a threshold, your hash table will grow its number of buckets by at least a factor of 3. Growth of the table will be transparent to the code using your hash table module.
- (Required) You will need a hashing function to hash the strings you insert and lookup in your hash table. You can research your own function, or use the one posted on the webpage. If you develop your own function, document your hash algorithm in comments.
- (Required) Use the standard library procedure "qsort()" for sorting, the unix man page should help with how to use it.
- (Required) When your program outputs word pairs and their occurrence counts, output one word pair per line using the format `%10d %s\n"`, where the decimal number is the number of occurrences and the string is the word pair (with one space between the words).
- (Required) Use good coding practices and make your code readable and understandable. Proper and consistent indention is required.

- (Required) Break your code into at least two source files, one for the code implementing the hash table and at least one other for the implementation of word pair counting. Design a good interface for your hash table implementation. The interface should not be specific to this application and must be re-entrant. The interface should be reflected in a header file included by both source programs and containing comments describing how each method (and its parameters) are used. The program using the hash table implementation must not need to comprehend the internal structure of the hash table, it should perform actions solely through the procedure interface declared in the hash table's header file.
- (Required) Your program should manage the heap in such a way as to avoid memory leaks. If data structure components become unused, they must be freed before they become inaccessible. Consider this in the code associated with growing your hash table.
- You are encouraged to use assert() where appropriate.
- Use man pages to find the details of the C library routines you need.
- (10 points) Design your program to be robust, anticipating exceptional data or boundary conditions coming from the user or the data files.
- Do not surf the web for code or solutions to this assignment.
- Think and design first, then implement. To the extent possible, implement and test components first, then integrate and test them together in a bottom up fashion. Start with simple tests and migrate to more strenuous test cases. You may test your program on the file `"gettysburg"`. You can also use a large text file on which to test called `"mobydick.txt"`. There are a few other datasets that you can use to test. Diagramming your data structures and paper simulation of operations on your data structures are highly recommended as part of your design process.
- (Required) Submit a zip containing all of your source files and a Makefile without absolute directory names or derived binary files. Execution of make with no parameters should build the target program `"wordpairs"`. Assume that the environment variable `GET_WORD` is defined as the pathname of a directory which contains directories `"include"` and `"lib"` containing `getWord.h` and `libget.a`, respectively. In your own build environment in the lab, you will want to define `GET_WORD` to be a path to the directory downloaded on the website.
- (Required) While your program may build and execute correctly in other system environments, it must build and execute correctly in the ENCS laboratory's environment.

What to turn in (in a zip on Autolab):

- all source code
- README.txt containing instructions on how to compile and run
- any other files the program might need