Wordstat

Introduction

Purpose

Wordstat is an application designed to display various statistics concerning a text file which it parses. Mainly, it finds all of the unique words matching the following RegEx:

```
((?:[a-zA-Z])(?:[a-zA-Z0-9])*)
```

Then it lists them in lexicographical order, and displays the number of times that particular word has been mentioned and the number of case-sensitive occurrences of said word exist in the text.

Compilation

Compiling Wordstat is simple, `cd` into the source directory and run `make`. The default target shall be built, which is the main wordstat application, and then it is free to be used, or installed with `make install`. Note, that super user permissions are required to install the application properly into `/usr/local/bin`.

Usage

```
wordstat <file>     <file> is the path to the file which is to be parsed
```

Implementation Details

Methodology

Wordstat employs a mixture of a hash table and the POSIX RegEx standard in order to match and store words. The lower case forms of words matched by the regular expression are used as keys in the hash table and their particular occurrence is taken note of. As wordstat consumes more and more words, they are used as keys and statistics about their uniqueness and occurrence are taken note of. At the end, the hash table is searched for all keys and then they are sorted and displayed. The program terminates thereafter.

Hash Table

The hash table is deviously simple, though is powered by a simple hash function which originated from the Lua project. It is modified in order to return positive numbers as hashes in order to be used later to find a spot in the hash table storage array where to place the item. Collisions are taken care of in the standard way, wherein each place in the storage array is actually the head of a linked list, and items which so happen to have the same key are prepended to the linked list. The hash table takes up approximately 4084 Bytes, as it is an array of pointers to linked list heads.

Hash Function

The Lua hashing function is difficult to analyze since it uses the string length as a parameter of its main loop, however in the long run the function performs **O(n)** operations, where **n** is length of the string being hashed.

Hash Table Retrieval

Since the key has to be hashed on every retrieval there are **O(n)** operations performed (where n is the string length) per retrieval of word information, however thereafter lookup is constant time since the exact position in the storage array is well known. In the rare case that there is a collision and multiple items are stored at that location then **m** nodes must be traversed to find the correct node. Therefore **O(m)** operations are performed per retrieval, counting the hashing as a single operation.

Hash Table Store

A hash has to be computed for every key. If there is a collision then the element to be stored is appended to a linked list of nodes at that storage location. Thus, **O(1)** operations are performed, counting the hashing as a single operation.

Linked List

Linked lists are used in two places. The first being inside the hash table storage container, used to solve collisions, and the second as keeping record of the various case-sensitive occurrences of a particular word. Traversing either linked list takes **O(n)** operations, **n** being the number of nodes in the linked list.

Searching/Pattern Matching

Searching and pattern matching are taken care of by the POSIX RegEx implementation which varies among Unix-like systems. This makes it difficult to say precisely what the runtime complexity is for searching, though advanced regex engines have boasted **O(m*n^2)** operations, where **m** is the length of the string and **n** is the length of the regular expression. The regular expression used in this implementation of wordstat is:

```
([[:alpha:]][[:alnum:]]*)
```

During searching, approximately 12kb is used as a buffer of the currently read chunk of a file, and as the leftover bit from words that got cut off from the past fread operation. Thus, wordstat can handle extremely long words, though words exceeding 8kb will cause problems.

Challenges

One of my biggest challenges in building wordstat was getting the correct mindset when dealing with pointers and managing memory. I cannot simply depend on the VM to give me more memory and to persist and free my objects for me. Allocating and cleaning in an intelligent way were also big challenges. Using valgrind helped me tremendously in understanding both what my program was actually doing, in contrast to what I was hoping it would do, and finding sources and solutions to all of my errors and faults.

Unfortunately, I built up wordstat, piece by piece over the course of the assignment. This is expected while learning the ropes of a language however I would have liked to make the separate pieces of the assignment individually, before tying them together. That way, I could reduce redundancy and keep a clear distinction between which part of the program should do what.

Created by: Artem Titoulenko

CS211: Computer Architecture