Bryan Lee   ID: 922649673
Github: BryanL43   CSC415 Operating Systems

Assignment 4 – Word Blast

Description:
This program reads the book War and Peace and acquires the top ten most frequent words with six letters or more. It also supports multi-threading, where the user can specify the number of threads in the run options to speed up the frequency counting.

Approach:
To start the design phase, I will break down the beginning of the program into these steps:
1. Initialize a mutex
2. Parse the run options to get the filename and thread count.
3. Open the file and acquire its size.
4. Calculate the buffer size for each thread's buffer.
   a. The thread's buffer will store the fragmented (file size/thread count) file data to be processed by the threads.
5. Read and store the fragmented file data in the thread's buffer.
6. Instantiate the global array to store all the words and their frequencies (the word and frequency will be a structure).

The next part of the program is the thread section. I will first need to create the specified number of threads and a new function, with void pointer return type, for the start_routine parameter of pthread_create. The function will increment the frequency of a word or add a new word to the global array, which stores all the word and frequency structures.

Some planning revisions:
I initially planned to store the data using a linked-list vector instead of a global array; however, this proved disastrous. After carefully weighing the trade-offs and benefits of using a vector versus a global array, I have decided to use the global array approach. I will discuss this in detail in the issue and resolution section.

The thread's function will tokenize the buffer that holds a file segment with the strtok_r function for thread safety and the provided delimiter. Now, I must iterate through the tokens and check if the length is six characters or more. After this, I will need to do three operations:
1. Increment the frequency of a word that is already in the global array.
2. Resize the global array with realloc if it gets full.
   a. I will double the global array every time the array gets filled up.
3. Add a new word to the global array with a frequency of one.

The program's most critical component, the mutex lock, will protect two parts: incrementing the frequency of a word and adding a new word to the global array. These two sections are called the critical section, which all the threads must access.

IMPORTANT: Since the thread will only contend in either of the two sections, I will have to apply two sets of mutex locks. To clarify this essential aspect of the thread's function, I will follow this pseudocode:

```
while (token != NULL) {
        if (the word is six or more letters) {
                if (found word) {
                        mutex_lock;
                        frequency++;
                        mutex_unlock;
                }
                if (word is not found) {
                        mutex_lock;
                        resize global array if needed;
                        Add a new word with a frequency of 1;
                        mutex_unlock;
                }
        }
}
```

By protecting two sections of the code and immediately unlocking the mutex, I can reduce the critical section to its most reduced form and reduce extra points of contention between threads.

The last part of the program will need to join the threads, sort the words and their frequencies in descending order, display the top ten most frequent words with six characters or more, and perform a clean-up: free the buffers, destroy the mutex lock, and close the file.

Note: I will use a quicksort as it is the fastest sorting algorithm for this program.

Issues and Resolutions:
My first issue was creating a data structure to store my WordFreq structure containing a unique word and its frequency. Since I did not know how many unique words I would need to store, I implemented a linked-list vector to store the WordFreqs dynamically. Creating the vector required extra code and memory management, making it prone to errors and memory issues. The vector also did not cooperate reasonably with the presence of a mutex lock. Despite implementing the critical sections accurately, there was no reduction in time for multi-threading. However, I discovered that the error lies in unlocking the mutex, particularly the size of the critical section. I had to move the mutex lock into the vector data structure to reduce the critical section or choose another option.

Example of one of my vector implementations that did not work correctly:

```c
void vectorInit(Vector* vector, int capacity) {
    vector->data = malloc(capacity * sizeof(WordFreq));
    if (vector->data == NULL) {
        perror("Failed to instantiate vector data!\n");
        exit(EXIT_FAILURE);
    }
    vector->size = 0;
    vector->capacity = capacity;
}

void vectorAdd(Vector* vector, const char* word) {
    //Check if the word already exists in the vector
    for (int i = 0; i < vector->size; i++) {
        if (strcasecmp(vector->data[i].word, word) == 0) {
            //Word found, increment the frequency
            vector->data[i].freq++;
            return;
        }
    }

    if (vector->size >= vector->capacity) {
        return;
    }

    //Allocate memory for the new word and copy it
    vector->data[vector->size].word = malloc(strlen(word) + 1);
    if (vector->data[vector->size].word == NULL) {
        perror("Memory allocation for word failed!\n");
        exit(EXIT_FAILURE);
    }
    strcpy(vector->data[vector->size].word, word);
    vector->data[vector->size].freq = 1;
    vector->size++;
}
```

My solution was to replace the vector with a global array to store the WordFreq. The global array approach was more straightforward to implement, reducing the chance of memory mismanagement and the program's complexity. The trade-off is that I had to double the size of the global array (arraySize * sizeof(WordFreq)) when it got full. The constant resizing resolved my initial dynamic storage issue, but it led to some wasted memory space.

My other issue was the size of the critical section. Initially, I decided to wrap my entire frequency increment and add a unique word with a mutex lock. However, this did not improve time performance. An example of my original idea (image on the next page):

```
char* token = strtok_r(buffer, delim, &saveptr);
while (token != NULL) {
    if (strlen(token) >= 6) {

        pthread_mutex_lock(&lock);
        //If word is already in array then increment frequency
        for (int i = 0; i < counterArraySize; i++) {
            if (strcasecmp(counterArray[i].word, token) == 0) {
                counterArray[i].freq++;
                break;
            }
        }
    }
```

Long story short, my oversight on the break statement caused me to misuse the Mutex locks. I forgot that the break statement in the code only exits the for loop, thus continuing the other logic and duplicating a word every iteration. I was able to resolve this by simply adding a check to see if the word is found (as shown in the following image):

```
(strlen(token) >= 6) {
    int found = 0;

    //If word is already in array then increment frequency
    for (int i = 0; i < counterArraySize; i++) {
        if (strcasecmp(counterArray[i].word, token) == 0) {
            pthread_mutex_lock(&lock);
            counterArray[i].freq++;
            found = 1;
            pthread_mutex_unlock(&lock);
            break;
        }
    }

    //If word isn't in array then add it to array with frequency of 1
    if (!found) {
        pthread_mutex_lock(&lock);

        //Resize the counter array when it gets full
        if (counterArraySize == arraySize - 1) {
            arraySize += 500;
            counterArray = realloc(counterArray, arraySize * sizeof(Wor
            if (counterArray == NULL) {
                perror("Failed to resize counter array");
                exit(EXIT_FAILURE);
            }
        }
    }
```
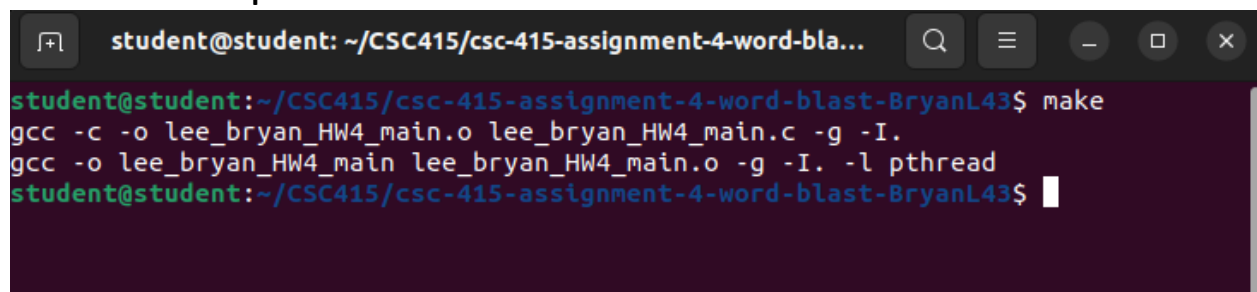
Another critical section is necessary in the code because the frequency increment section will not execute if the word does not exist in the global array. However, as shown in the image, only one critical section will still be executed per token iteration.
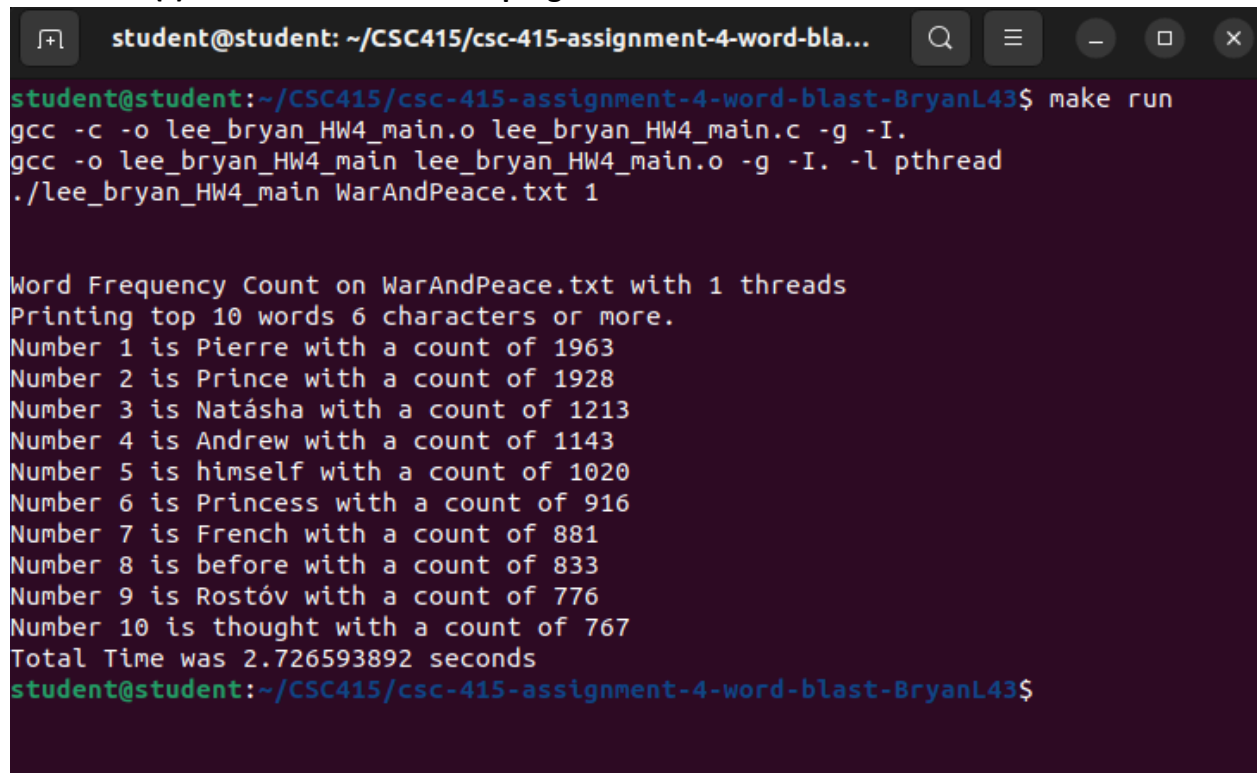
Analysis:
The time difference between using one thread and two threads is 50%. However, there is a lack of performance improvement with three and four threads. Although the time deviates slightly, properly due to thread joining, sorting, or computer specs, they remain the same as two threads. The lack of performance improvement is due to the virtual machine's number of cores. Since we are implicitly told not to change the settings, the number of total processor (CPU) cores is two, which limits the parallel processing ability. Introducing more than two threads to the two cores will not result in parallelism; instead, they will compete against each other for the same resource, causing the operating system to switch between threads. This switching adds overhead and diminishes the potential for performance gains in multithreading.

Screenshot of compilation:

Screenshot(s) of the execution of the program:

```
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$ make run
./lee_bryan_HW4_main WarAndPeace.txt 2


Word Frequency Count on WarAndPeace.txt with 2 threads
Printing top 10 words 6 characters or more.
Number 1 is Pierre with a count of 1963
Number 2 is Prince with a count of 1928
Number 3 is Natásha with a count of 1213
Number 4 is Andrew with a count of 1143
Number 5 is himself with a count of 1020
Number 6 is Princess with a count of 916
Number 7 is French with a count of 881
Number 8 is before with a count of 833
Number 9 is Rostóv with a count of 776
Number 10 is thought with a count of 767
Total Time was 1.600265446 seconds
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$
```

```
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$ make run
./lee_bryan_HW4_main WarAndPeace.txt 4


Word Frequency Count on WarAndPeace.txt with 4 threads
Printing top 10 words 6 characters or more.
Number 1 is Pierre with a count of 1963
Number 2 is Prince with a count of 1928
Number 3 is Natásha with a count of 1212
Number 4 is Andrew with a count of 1143
Number 5 is himself with a count of 1020
Number 6 is princess with a count of 916
Number 7 is French with a count of 881
Number 8 is before with a count of 833
Number 9 is Rostóv with a count of 776
Number 10 is thought with a count of 767
Total Time was 1.760763974 seconds
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$
```

```
Total Time was 1.738705944 seconds
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$ make run
./lee_bryan_HW4_main WarAndPeace.txt 8


Word Frequency Count on WarAndPeace.txt with 8 threads
Printing top 10 words 6 characters or more.
Number 1 is Pierre with a count of 1963
Number 2 is Prince with a count of 1928
Number 3 is Natásha with a count of 1213
Number 4 is Andrew with a count of 1143
Number 5 is himself with a count of 1020
Number 6 is Princess with a count of 916
Number 7 is French with a count of 881
Number 8 is before with a count of 833
Number 9 is Rostóv with a count of 776
Number 10 is thought with a count of 767
Total Time was 1.738921219 seconds
student@student:~/CSC415/csc-415-assignment-4-word-blast-BryanL43$
```