**Project-1**

CMPSC472

Mark Vernachio

10/23/24

**1. Project Description**

The project implements a file processing system that counts word frequencies across multiple files. The system is designed to use both **multiprocessing** and **multithreading** for parallel processing of files. It aims to compare the performance of these two approaches in terms of processing time and resource usage, with results being aggregated and analyzed.

**Problem Definition:**

* **Objective**: Process a set of text files, count the frequency of words in each file, and merge the results. The project will compare the use of **multiprocessing** (separate processes per file) versus **multithreading** (multiple threads within each process).
* **Key Features**:
  + Efficient file processing using **parallelism**.
  + **Inter-process communication (IPC)** to merge results in the parent process.
  + Comparison of performance between multiprocessing and multithreading.
  + Generation of a histogram and top 50 most frequent words.

**2. Structure**

The structure of the code involves the following key components:

* **Main Process**:
  + Forks child processes for each file.
  + Each child process processes one file and uses multithreading to divide the file into chunks for parallel word counting.
  + The results (word counts) are sent back to the parent process using pipes.
* **Child Process**:
  + Each child process uses threads to count word frequencies in file chunks.
  + After processing, the child process sends results (word and frequency) back to the parent process.
* **Parent Process**:
  + Aggregates word counts from all child processes and merges the results.
  + Displays the top 50 most frequent words and their frequencies.
  + Compares performance between multiprocessing and multithreading.
* Important Info:
  + Parent process forks child process, one per file
  + Child processes spawn threads to divide the file into chunks to count the words.
  + The results are sent back to the parent process and filtered so the output is words >= 3.

**3. Instructions**

I utilized VirtualBox and Ubuntu Linux to run my code. I also used Vagrant to connect these so that I could run my code using Windows PowerShell. You can also run the code using GCC, but I decided to use this method as I learned it in another class and it was already set up for me to use.

**GCC implementation**:

* Install GCC and pthread support if not already available.
* Place all required text files (bib, paper1, paper2, etc.) in the same directory as the code.

**Compilation**:

* Compile the C code using:
  + gcc -pthread -o main main.c

**Execution**:

* Run the program by executing:
  + ./main

**Output**:

* The system will display the top 50 most frequent words across all files along with their frequencies.
* The system will also provide time comparison metrics between multiprocessing and multithreading.

**Vagrant Implementation**:

* Install Vagrant
* Download the Project file on github for easy implementation
* Install VirtualBox and Ubuntu current versions

**Compilation:**

* Open Windows PowerShell and move to file location.
* Run “Vagrant up” to initialize the virtual machine environment.
* Then run “Vagrant ssh” to connect to the virtual machine environment.
* After compiling the code using gcc -o main main.c.

**Execution:**

* ./main

**Output**:

* The system will do the same as before, displaying the top 50 most frequent words across all files along with their frequencies.
* The system will also provide time comparison metrics between multiprocessing and multithreading.

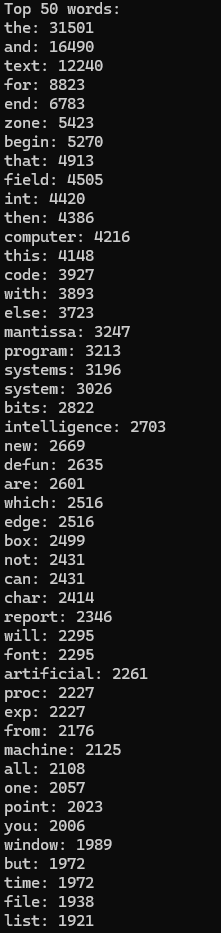
**4. Verification of Code Functionality**

* **Histogram:**

A graph of a person with blue squares

Description automatically generated with medium confidence

* + Provides the most frequent word counts, generated using matplotlib in python
* **Top 50 Words:**



**5. Performance Comparison:**

* **Implementation**
  + For my implementation, I used time comparisons for both multiprocessing and multithreading. After executing the code, it displays the total time taken for the combination of all processes compared to the time taken for multithreading.
  + In most scenarios, the timing is around 0.5-0.6 seconds to run through the code for each method, with multithreading normally slightly being faster than multiprocessing.
* **Timing**
  + **#**include <time.h> to track the amount of time used

**6. Results:**

* **Observations:**
  + Multiprocessing creates separates processes, which have higher memory usage. This is because the processes use separate memory spaces.
  + Multithreading works within a single process with shared memory. Which allows memory to not be used to the same extent, but context switching produces limitations, which makes it much more complicated to work with.
* **Performance:**
  + Multiprocessing works better when utilizing larger files, and while I could not exactly determine why multithreading works faster, I can assume the reason is due to the files we are utilizing varying in size.
  + Multithreading is more efficient but just not in the case of utilizing larger files due to shared memory between the threads.
* **Improvement**:
  + If we were to use a hybrid model it could introduce faster speeds.