**Operating System Principles – Assignment 1**

***Matthew Yamen***

***S3902219***

**Performance Measurement**

The main performance measuring tool used was C++’s chrono library and the high resolution methods, in which I created my own timing utilities that record a function’s execution time, which is very accurate. I chose this over linux perf (as my main tool) and other tools because it gave a more accurate depiction of my algorithm’s runtime, since there were some snippets of code that were solely for maintenance and to clean up resources, which should not be included in the execution time. My timing utilities allowed me to measure on a per-unit basis (the unit being a function), and they are generic, making them very flexible and versatile. However, perf was used as a supplementary performance measurement tool, as it provides far more details than just the time. The third and final tool I utilised was linux top, which was very useful in visually checking if the thread priorities were being correctly set.

**Note #1:** for **all** tests I will be using wlist\_match1.txt as the dirty file that will be cleansed.

**Note #2:** all performance (time) measurements have been average over 10 attempts to ensure better accuracy.

**Optimisations**

**Using C++ Sets over C++ Vectors**

Iterating over a C++ vector (std::vector) is a lot quicker than iterating over a C++ set (std::set). So, if you chose a vector implementation for the clean words list, and you are iterating over it a lot, then good you made a choice, as the same code with a set implementation will be around 1.6x (approx.) slower. Satisfying the duplicate elements criterion and the sorting with a set is, however, a lot more concise and convenient (possibly faster too then calling std::sort and std::unique/std::erase for the vector implementation - haven't tested this though, so it is pure speculation and you should take it with a grain of salt), but if you are iterating over the clean words list at least twice in your code, then the vector implementation will be quite a bit quicker. You can do a hybrid of both, and use one where the other is weaker, but if you can't be bothered, I'd go for the vectors. To test this, I altered tasks 1-5, so that they would use sets. I had already measured the performance using the vectors, and so all I needed to do was benchmark with the set implementation, which required significant changes.

After some research, the reasoning behind it is due to the way the data structures are stored internally; vector items are contiguously stored, whereas set items are node-based and non-contiguous. The advantage of contiguously stored data is that it allows pre-fetching on the cache. Essentially, the data can be stored in the cache, and read from it, rather than RAM, which is much faster. However, sets cannot do this pre-fetching on the cache, as there is no guarantee on where the elements will be (with contiguous elements, they are guaranteed to be next to one another), so it is forced to read from RAM.

I tried both implementations and this was my experience. Here is a comparison of the times **(average out of 10)** for Task 2:

* **Set Implementation >>** *Time taken in seconds: 2.561764s.*
* **Vector Implementation >>** *Time taken in seconds: 1.577395s.*

The other tasks showed very similar results, and thus, they have been omitted from this document for the sake of brevity. Overall, I ended up wasting time refactoring a lot of code to try this potential optimisation, though, it was nonetheless a great learning experience.

**Using C API regex over C++ standard library regex**

Interestingly, the C++ std library regexes (std::regex) were around 4-5 times slower than the C API regular expressions. Swapping the C++ ones for the C ones drastically reduced the time of my task 1 filtering function from (at the time) an average (taken out of 10 runs) of 5.5 seconds to 1.2 seconds. After some research, I discovered that the reason C++’s regular expressions are so much slower is because the regex pattern is parsed and compiled at runtime, rather than at compile time. The latter is done, however, with C regular expressions.

For Tasks 2-4, the merge function seems to be the bottleneck, as it slows down everything else.

**Task 1**

I decided on the following filtering rules:

1. Each word must have a length of 3 to 15 (inclusive of both bounds).
2. Each word can only contain alphabetical characters (a-z, ignoring casing, since all words are lower-case by default – to include upper-case letters, we simply do a-zA-Z).
3. The list of words is to be sorted on the third character and onwards.

The first **two** **rules** were enforced via the following sub-regular-expressions: {3,15} and [a-z]. Utilising other meta characters to ensure the first two rules are properly enforced, we arrive at the regular expression “^[a-z]{3,15}$”, which roughly translates, in English, to all strings of length between 3 and 15 (inclusive), such that it begins and ends in letters of the English alphabet.

To enforce these filtering rules, I employed the following coreutils tools:

* grep – for filtering the dirty file using regular expressions (enforced my rule),
* uniq – to remove all duplicates,
* sort – to sort on the third character onwards (this is done before uniq),
* output redirection – to redirect to the clean file.

|  |  |
| --- | --- |
| Bash Script Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 2.44s |

|  |  |
| --- | --- |
| C++ Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 0.64s |

**VALGRIND IF TIME.**

**Task 2**

Despite the specification stating that we should read data from the clean file, I will skip the overhead of calling task\_filter and writing the data to a file, then reading it again, and simply call task\_filter and have it return the list of clean words (in-memory). This ensures that the algorithm used in Task 1 is consistent with the algorithm used in this task, and thus, eliminates bias that might affect the performance results.

Here is the performance data for this task.

|  |  |
| --- | --- |
| C++ Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 0.97s |

**Task 3**

Did not wait for the 13 FIFO files to be open for writing by map3 to perform the reduction step, as this is an unnecessary bottleneck, and will only slow down the code. It makes much more sense for the files to write and read in any order, and when one fifo has both a read and write thread attached, then it begins and that thread continues. Reduce need not go strictly after map.

Here is the performance data for this task.

|  |  |
| --- | --- |
| C++ Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 1.17s |

**Task 4**

Optimisation 1: Put threads that aren't writing to sleep.

This optimisation is useless, because when a thread is waiting for I/O in THIS CASE (i.e., in the case of the named pipes, it is waiting for the other end to be opened), it is put to sleep and does not use any CPU.

Optimisation 2: Prioritise threads with bigger lists to process.

Result: Likely made it slower on my laptop. This is not a good form of optimisation, because all the threads have a nice value of 0 by default anyway, and since I do not have superuser privileges on the teaching servers, I cannot use negative nice values to increase the priority. This is a problem because, we can only add positive nice values, which means some threads will be prioritised lower than other threads that belong to totally different processes (since all processes/threads default to 0 nice value and PR 20), and so, our program is actually getting less CPU time than if we had left the nice values alone (to default to their minimum). This would only be effective if we could use negative nice values, so that we can prioritise the heavier threads WITHOUT the worry of other unrelated threads having a higher priority over our ones.

Here is the performance data for this task.

|  |  |
| --- | --- |
| C++ Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 1.21s |

**Task 5**