**Operating System Principles – Assignment 1**

***Matthew Yamen***

***S3902219***

**GITHUB PROJECT LINK:** <https://github.com/Huntermuze/ospassignment1>

**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 |

**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.

|  |  |
| --- | --- |
| Task 2 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.

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

In terms of the performance, compared to Task 2, this task is actually slightly slower, which, after some research and extensive performance testing, I discovered was due to the slight overhead of the thread management. The processes in Task 2 didn’t need any management, and they only ran for a short period of time.

**Task 4**

The priority-scheme for this task was to prioritise the threads with heavier loads over the tasks with lighter loads, so that the tasks would all roughly finish at the same time. The nice values used were in the range -1 to -13, so as to ensure that other unrelated processes (particularly on the teaching servers) do not get priority over my threads, since the default nice value is 0 (giving each process or thread, in the case of linux, a default priority of 20). Of course, a nice value of -19 is not wise as it can starve other processes/threads, and hence, I made sure the upper bound of the nice value was -13, which gives a maximum priority level of 7 (i.e., the threads with the biggest load will have this priority).

The load values (i.e., a quantity determining how much work a thread has to do) were calculated by counting the number of words in each list and considering the length of the words in each list. The length of the words in each list will affect the time spent processing that list, as words with more letters required scans in the sorting rule (i.e., on 3rd character onwards), since a call to sub-string works on a per-character basis, and has a time complexity of O(n), where n is the size of the string. This accumulates very quickly, since each call to the sort comparator requires two sub-string calls, so we now are looking at 2n per comparator call. Taking that further, every two items in a given word list need to have the comparator called on them, making the overall time complexity (in the worst case scenario) for the lists with more characters 2\*n\*(m-1), which is O(m\*n). This is a quadratic time complexity, which is significant. As such, I used both the length of each word in a list and the size of that word list as the two defining factors that will determine a thread’s priority. We simply add the size of the list (we only take the **first two digits, separated by a decimal point, of the list size**, to normalise it)to the size of each word in that list, to get a load value (i.e., for a list of length 5 words, which has 5385 words, the load value is 5 + (5385 / 1000) = **10.3**) We compute this for every list, and then sort that list of computed load values in ascending order. We then start allocating nice values from -13 to -1, where the threads with the bigger load value get the lower nice value (making the priority higher in the sense that it is executed before any of the others).

Interestingly, however, this was not a good optimisation, as it did not make much of a difference, since the threads execute so quickly that the CPU does not care what the priority of the threads are, as they can get done very quickly.

In addition, I attempted to put the threads to sleep when they weren’t writing or reading. However, upon investigation, I learned that when a thread is waiting for I/O (in the case of named pipes, it is waiting for the other end to be opened), it is put to sleep and its CPU usage is absolutely minimal.

The reduce thread was indeed the bottleneck of the entire program, and that was where most of the execution time was spent, since the merging was not exactly efficient. Further, the threads interestingly spent most of their time waiting, because of the excessive pipe calls between the read and writing threads.

Here is the performance data for this task.

|  |  |
| --- | --- |
| Task 4 Performance | |
| Number of 3-15 letter words | 1367007 |
| Time taken (seconds) | 1.21s |

**Task 5**

In my github repo, I changed Task 4 to make it streamable, utilising the same design as that task, however, I soon realised it is very inefficient to complete this task that way, and changed it to the way it is now (feel free to review the repo to see the initial design). To change Task 4, you simply just make one map thread, and have that map thread send the word it has currently read in from the main named pipe (i.e., where the words are being read in) to the reduce thread that corresponds to the thread responsible for handling words of those lengths, and then that reduce thread simply increments a counter in the global array containing the size of each list (as it grows). Then, another thread simply displays the contents of that global array of word counts to stdout, until it stops changing (i.e., all the words have been counted).

The priority-scheduling algorithm would be a good choice, so that the threads containing the word lists where each word is shorter in length could be given a higher priority, and thus, be completed before the words longer in length.