RMIT
UNIVERSITY

Operating Systems Principles

Assignment 1

Guy Seccull (s3785085)

Assignment 1 Report

GitHub Account URL – https://github.com/GuySeccull
GitHub Project URL - https://github.com/GuySeccull/osp_assignment1

Task 1

Task 1 notes and design choices

- Task 1 only requires that the filtering be done to the supplied wordlist, this means that TaskFilter requires both the input and output file name to do this, but all the other tasks only require the input filename. For this reason I've overloaded the TaskFilter function, one takes only the wordlist file, and the other takes both the wordlist and output file.

A - The Combination of coreutils used to generate the equivalent of Task1Filter

To complete the task of filtering words and sorting using the core utilities (with the exception of grep which was allowed according to Ron), I chose to use a combination of grep, sort and uniq to achieve this. First by using regular expressions with grep, I am able to filter out certain characters and word lengths to get only the results I want, then using sort, I can provide the '-k 1.3' flag to sort entries from the 1st word, 3rd letter onwards, then I used uniq with the '-u' flag in order to print out only the unique entries. Although it is possible to only output unique entries with sort via the '-u' flag, by doing this with the '-k 1.3' flag set, it only looks for the uniqueness of words based on the third character onwards so it doesn't work properly, hence why uniq is used. The full command I use is:

```
grep -o -w '[a-zA-Z]\{3,15\}' $file | sort -k 1.3 | uniq -u > $output
```

This command will only include alphabetical characters in the filtered words, but the code of this task and all other tasks can accept any character whatsoever, these are just the simple rules I've chosen.

B – The source of Task1Filter

The source code is included in the zip file.

C - The number of words of length 3 to 15 letters in the data set you end up with.

From the provided dataset at https://www.keithv.com/software/wlist/ I have chosen to use the file 'wlist_match2.txt' as the basis for testing performance. Unfiltered this file contains 586,880 words, and once filtered it will have 536,477 words to sort.

D – The performance data for this task

Statistic	*Data*
Execution time	User time was on average 1800 milliseconds System time was on average 70 milliseconds
Peak memory footprint	315,392
CPU cycles elapsed	5,571,323
Instructions retired	4,934,827

Task 2

Task 2 notes and design choices

- I've chosen to use vectors over arrays in my implementation as I don't know how many words of *n* size are in the file so I cannot declare the array size correctly, meaning I will constantly need to resize it, and vectors make this process much easier.
- Instead of having TaskFilter return a global array/vector of all words, I decided it would be better just to have TaskFilter generate a file from the filtered words, then in map2, open that file and add each word to its corresponding character length vector because the global array/vector has no use everywhere else.
- The files my program generates are numbered 0-12 and not 3-15 based on character count, this is just to make looping easier and my code clearer.
- To sort each file, I'm using the same thing I used from task 1, *sort -k 1.3*
- The way I've designed my reduce function is that it loops through one word from each file, creates a substring from the third character onwards, and compares them against each other iteratively, then the lowest sort order of that loop gets written into the output file and another line is read from the file the lowest sort order word came from.

Performance

Statistic	*Data*
Execution time	User time was on average 6000 milliseconds System time was on average 250 milliseconds
Peak memory footprint	23,105,536
CPU cycles elapsed	6,041,895,401
Instructions retired	11,075,616,964

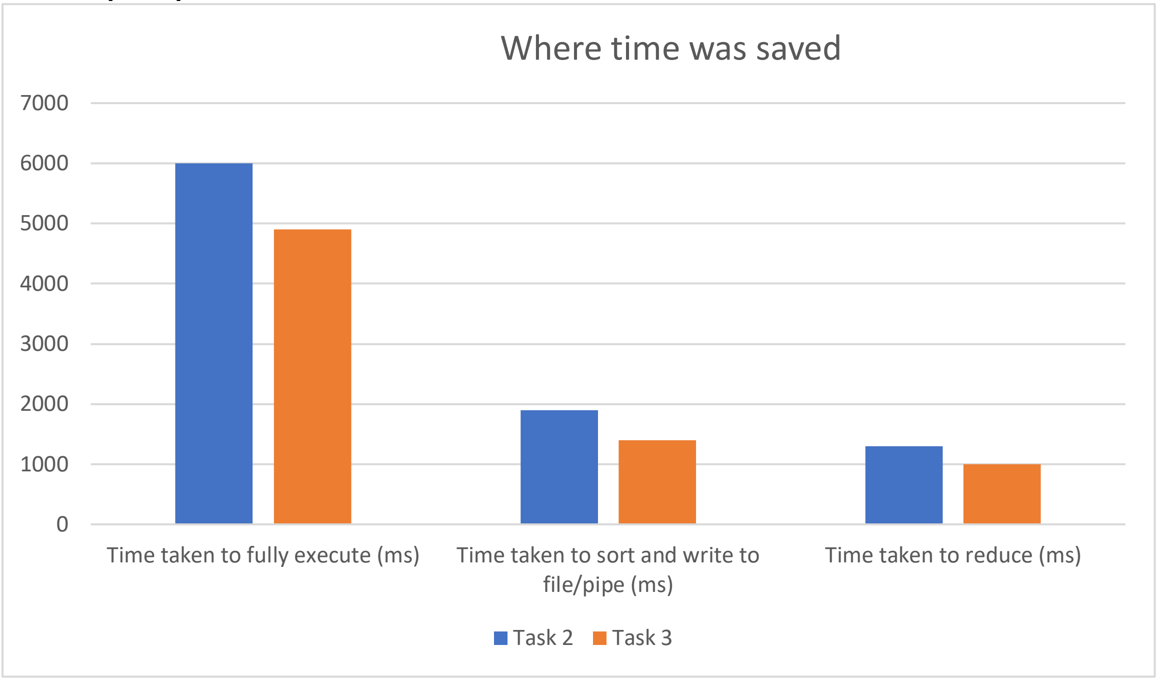
Task 3

Notes for my task 3 implementation

- I have used vectors as opposed to arrays here for the same reason as in task 2.
- For the mapping and reducing parts of this task, I have each section create 13 threads, so map3 will create 13 thread which will first sort, then it will open a pipe and write the sorted words to that pipe, using the character count of that word as the buffer size. Reduce3 will have 13 corresponding threads that read from the pipes and add each word to a vector.
- Since the sort/write threads are tied to the read threads, meaning that the read threads will finish after the write threads because of the way the pipes work, and read threads will always finish last, I haven't used pthread_join() on the sort/write threads, I've only done this on the read threads because if the read threads are finished and all join, then the write threads would also be finished.

- Instead of the C style qsort() function, I have used C++'s sort() which according to https://www.geeksforgeeks.org/c-qsort-vs-c-sort/, says that C++'s sort() is far more efficient.
- Previously the read threads were somehow executing before its corresponding write thread could execute, so instead of acquiring the lock at the start of the sort thread, I have locked the mutex just before the sort thread is created, this prevents any errors with the read thread from occurring.

A - Compare performance and consider where the time was saved or lost.



Looking at the above chart shows that between task 2 and 3, the most time was saved during the sorting and writing to file/pipe period with task 3 taking on average 500 milliseconds less than task 2. The reason for this is most likely due to the multithreading capabilities of task 3, as well as the use of pipes over regular files which made writing and reading from files quicker than a single threaded, normal file solution like task 2. Additionally, about 300 or so milliseconds were saved during the reduce process, this is likely due to the fact that task 3's reduce read words from vectors as opposed to iteratively getting lines from the text files in task 2.

Task 4

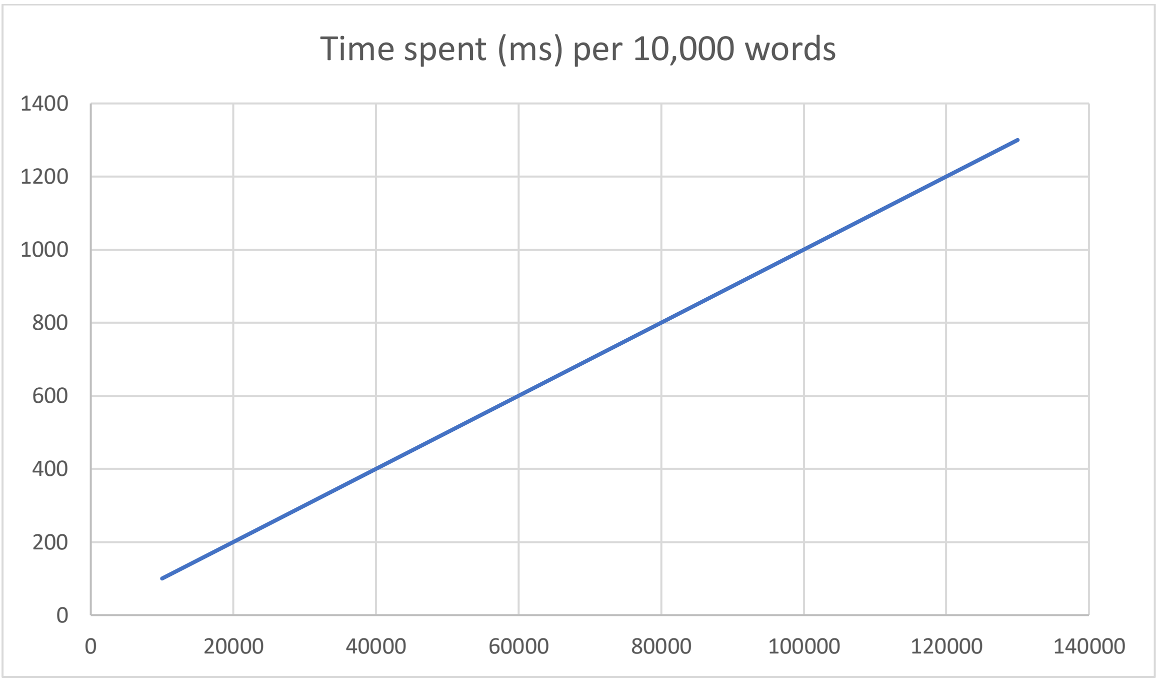
Notes for my task 4 implementation

- For this to work successfully it is vital that this is run on linux with full privileges by running the program with sudo. The RMIT servers only allow priority values (nice values) to be changed between 15 to 19, whereas the full range is -20 to 19. Although the nice values can be set on Macs successfully, they don't actually do anything so this is why it must be done either via WSL or a linux machine/VM other than the RMIT servers.

- After using various wordlists and seeing the ratio of word lengths, I found that the best way to dynamically determine the nice values, is to first figure out the percentage out of 100, that each word length makes up from the total wordlist, multiply that value by 2.6, round it up then subtract it from 19 (the maximum nice value). Word lengths with a low occurrence percentage will be closer to 19, and words that occur more will reach down to -20, where threads with a nice value of 19 are the least prioritized, and threads with -20 are the most prioritized.
- It's important to note that to see any actual benefit from this, a very large wordlist needs to be used (or maybe a less powerful machine) otherwise the difference in thread finish times are already going to be too small for nice values to change this.

A – Where are the threads spending most of their time?
Each individual thread spends most of its time writing and reading to/from the pipes with less, but still a significant amount of time being spent on the sorting process. As for the threads as a whole, the more characters a thread has to deal with (characters being the words per thread multiplied by the word length), the longer it will take, where words of lengths 7 and 8 generally make up the most of the wordlists, ergo taking up the most time of all the threads.

B – Big O representation and justification

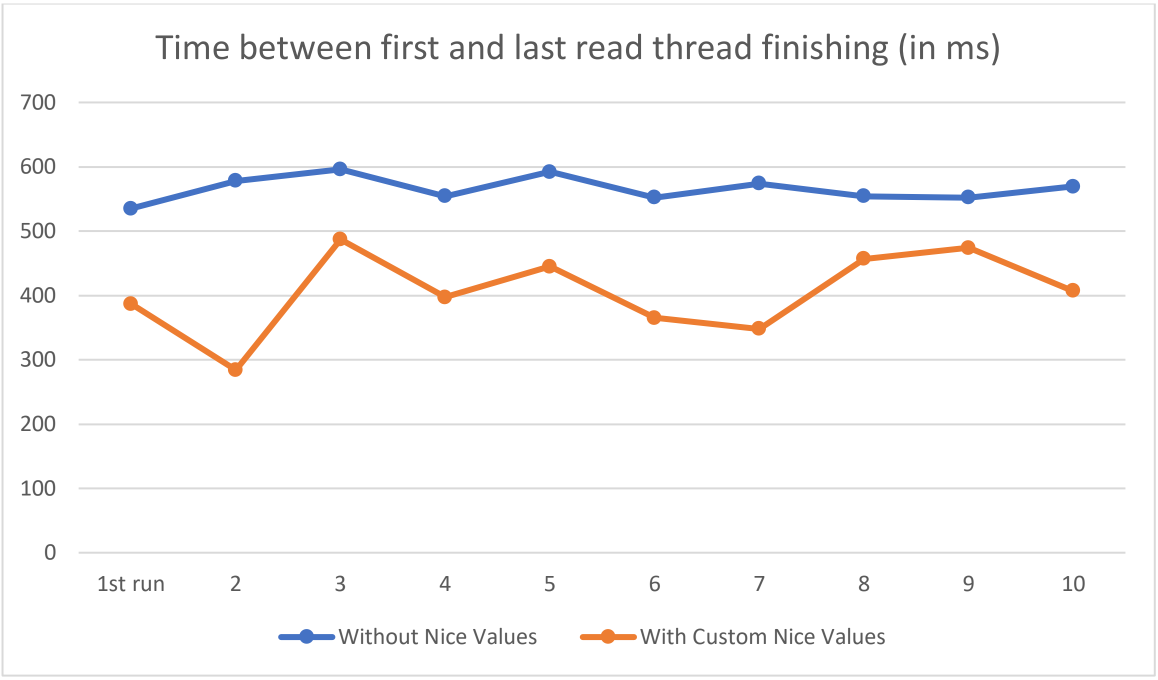


In each sort/write thread it has two main tasks, sort the words, then write to the pipe. C++'s std::sort is O(n logn) which is practically linear, and then writing the words to the pipe is itself linear as it writes words to the pipe at a constant rate, so I figured that the time spent per 10,000 words roughly equates to 100ms.

C - For the larger word lengths the string compare will also take longer. Is that significant?
The answer to this question depends on the wordlist used. If each word length from 3-15 characters long, all had 100,000 words to sort, C++'s std::sort function has a time complexity

of O(n logn) so for words that have more characters in it, the time taken will grow quite linearly and will definitely take longer than the other lengths. This can be significant depending on the number of words used, but as is the case for most wordlists, words with more characters occur less often, so they end up finishing sorting either before or around the same time as more common word lengths.

Performance



The chart above depicts the time difference, in milliseconds, between the first read thread finishing and the last read thread finishing, based on the 10 times I ran my program. As you can see in the chart, without custom nice values the first and last thread would finish on average 550ms apart, whereas with custom nice values, the times were quite sporadic, generally ranging from 300-450ms.

Task 5

A - Give a detailed description of how you would change Task4 in order to make it streamable, but without sorting (so no blocking needed)

In order to modify Task4 to make it streamable, the easiest solution would likely be to remove the multi-threading capabilities as it no longer needs to be multithreaded. Map5 would be replaced with a while loop that continuously reads from std::cin (for C++) until it detects an end of file, which can be done with *while(std::cin)*. Depending on the solution required, map5 can either add the words it reads in based off character size (ignoring any newline or carriage return characters) to an array/vector and then pass a pointer to reduce5, or it can write to a pipe with reduce5 listening on the other end, meaning map5 will need to create a thread.

B - Describe what CPU scheduling algorithm could be used if there was no sort but there was a precedence rule that said that shortest words pass first in reduce5().

One CPU scheduling algorithm that could be used for this rule would likely be round-robin or RR, although this algorithm treats all jobs equally in terms of time, this only means that the shorter words which are shorter jobs, will pass before the longer words/jobs.