**Student name: Mingyuan Chen**

**Student id: s3782493**

**Git url:** [**https://github.com/H3ZHA/COSC1114\_A1**](https://github.com/H3ZHA/COSC1114_A1)

**Task 1.**

1. **grep -x '.\{3,15\}' *file* | sort -u -k1.3 > *Task1.sh***

Use grep to filter out words, **{3,15}** filtered out the words with length less than 3 or more than 15. Then use sort to sorting words, **-k1.3** will ignore first 2 characters and start sorting on the third character onwards, but it only effect when first part of words has more than 2 characters, sort command use space to separate parts by default, which mean this only effect when the words have more than 2 characters before first space, otherwise not effect. Last **-u** will ignore duplicates.

1. The data source is from <https://www.keithv.com/software/wlist/>, use wlist\_match12.txt be test. wlist\_match12.txt, Task1.sh and Cleanfile of Task1filter are provided in submission, these is in folder Task1.
2. If Makefile doesn’t work, use **g++ Main.cpp -o Task1filter.exe -Wall -g** to compile.

Use **./Task1filter.exe DirtyFile Cleanfile** to run program.

The final data set is in folder Task1, called **Task1\_Cleanfile.txt**. The thing need to be aware of is the output of code program(**Task1\_Cleanfile.txt**) is slightly different with output of coreutils program(**Task1.sh**). This is because the sorting logic is different, in my coreutils program command it will ignore punctuation and sort by character behind punctuation, and in code program it will sort by ASCII include punctuation.

Due to specification said ‘The files may contain punctuation and other symbols’, the code program should be better sorting. But due to ability problem, there is no idea to how to improve coreutils program, therefore keep different output here.

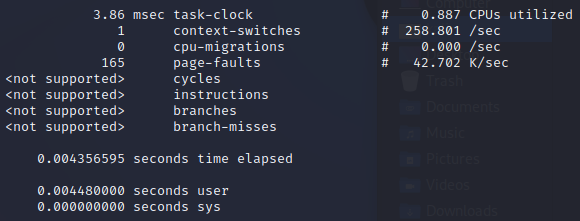
Due to my code program spend too much time, the Graceful Exit limit extend to 30s, change this by change line 72(back of main function).

1. Use **perf stat** to measure performance, the result below, mainly measure time taken, CPUs utilized, CPU migrations and page faults.

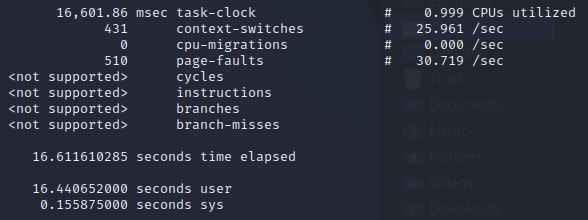
**Warning: The sort function has some problem, it will only detect characters after third character. Such as *we’d* and *he’d* will detected as same word and ignore the back one.**

**Due to my coreutils programs have same problem, this will keep as a feature, not a bug. If need fix this, part in Task5 provides a method.**

***Performance of coreutils program:***



***Performance of code program:***



**Task 2.**

The code of Task2 is modified from Task1, due to modified a lot of content and no longer to supported function of Task1, the source code(**Main2.cpp**), test files and output files of test are provided in folder Task2.

The code of Task2 follows the requirement:

Use same way with Task1 to filter out data(filter out word with length less than 2 or longer than 15);

Use **map2()** separate words to 13 list with their different length and sort each list in independent 13 forks, then output lists as a own file(**words\_3.txt – words\_15.txt**) in these forks;

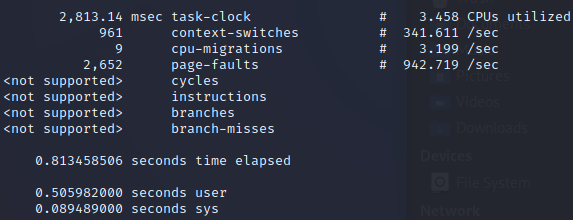
**Reduce2()** open all of these files and read 1 line each time, then write lowest sort order word to final output list and read next word in corresponding list, last output final file(**Task2\_final.txt**) after all word list are write done.

Use **g++ Main2.cpp -o Task2.exe -Wall -g** to compile.

Use **./Task2.exe InputFile OutputFile** to run program.

The final output using **fc Task2\_final.txt Task1\_Cleanfile.txt** to check, is same output with Task1.

The performance of Task2 below:



**Task 3.**

Same as Task2, the code of Task3 modified a lot of content(mainly **main()** and **sort()**), it is no longer to supported Task1 and Task2, the files of Task3 is in folder Task3.

The code of Task3 follows the requirement:

Use same way with Task1 to filter out data(filter out word with length less than 2 or longer than 15), then create **map3()** and **reduce3()** threads by **fork()**;

**map3()** create 13 index arrays and threads separate words to 13 list with their different length, each thread process one length, after done it will send global variable to **reduce3()** that can count number of word in list.;

In each thread of **map3()**, use **sort()** sort list of separated word, due to use **deque** to store data, **qsort()** do not support **deque**. After sort done, each thread will open independent FIFO file(if not exist, create one) to write words in order, once write one word and will wait **reduce3()** read it then write next one, until all words write done.

When **reduce3()** thread created, it will wait the signal of all **map3()** sort list done, because **reduce3()** needs global variable of remain words, **sort()** will filter out duplicates so can only do count after **sort()**.

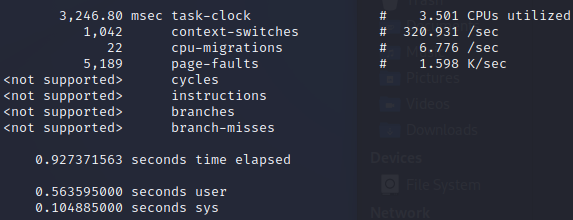
After trigger, it will open 13 FIFO files which **map3()** will open(if not exist, create one). It will read one word in each FIFO file that map3() can write next word. When 13 words read from 13 FIFO file, it will do merge operate like Task2, and read new word from FIFO file with same length until all **map3()** threads done. **reduce3()** use a global variable sent from **map3()** to count remain words.

Use **g++ Main3.cpp -o Task3.exe -Wall -g** to compile.

Use **./Task3.exe InputFile OutputFile** to run program.

The final output using **fc Task2\_final.txt Task3\_final.txt** to check, is same output with Task2.

The performance of Task2 below:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | time taken | CPUs utilized | CPU migrations | page faults |
| Task2 | 0.81 | 3.458 | 9 | 2652 |
| Task3 | 0.93 | 3.501 | 22 | 5189 |

As table show, Task3 has lower run speed, similar CPUs utilized and more CPU migrations and page faults.

Obviously, threads in Task3 need to do more things, therefore Task3 has more CPU migrations and page faults.

Also, either **reduce2()** in Task2 and **reduce3()** in Task3 needs to wait all **map()** threads done sort, can be inferred that time spend of program is mainly spend on sort.

The mainly difference between Task2 and Task3 is Task2 read 13 files once and do merge, Task3 do the real time transmission with **map()** and do merge, however the result show the way of Task3 can not save too much time, this may show more advantage when process lager data.

The reason of Task3 spend more time might be word separation, Task2 separate words once but Task3 do same separation in 13 threads.

**Task 4.**

If is **pThread**, can use **SetThreadPriority()** to assign priority, however the assignment limit to use **pThread**. Therefore here use **fork()** to create thread and use Linux command **renice** to assign thread priority.

**map4()** is based on **map3()**, but do following modify:

Unlike Task3 do separation in threads, will do separation before create new thread like Task2.

After separation, count number of words in each list separate by word length, and sort it by size, the sort index will subtracted in priority(lower number mean higher priority).

When create threads, use system() to call Linux command **renice** to assign thread priority by list size.

Due to need to use sort of c++, the **sort()** function change name to **my\_sort()**.

Delete print progress during **my\_sort()**, it can not work and will affect output of **renice** command.

**reduce4()** will exactly the same with **reduce3()**, did not do any modify.

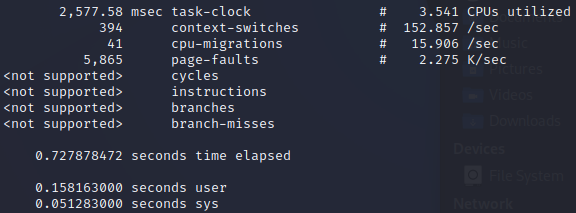
**Warning: renice command need root permission!**

Use **g++ Main4.cpp -o Task4.exe -Wall -g** to compile.

Use **./Task4.exe InputFile OutputFile** to run program.

The final output using **fc Task4\_final.txt Task3\_final.txt** to check, is same output with Task3.

The performance of Task2 below:



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | time taken | CPUs utilized | CPU migrations | page faults |
| Task2 | 0.81 | 3.458 | 9 | 2652 |
| Task3 | 0.93 | 3.501 | 22 | 5189 |
| Task4 | 0.73 | 3.541 | 41 | 5865 |

Task4 has higher run speed, similar CPUs utilized and more CPU migrations and page faults.

Because of priority set, Task4 will has more frequent CPU transfer, thus has more CPU migrations for sure.

Table show Task4 much faster than Task3, even faster than Task2, this mean the inference of Task3 is right: time spend of program is mainly spend on sort. Through different CPU assign can save much time.

In Linux, all new threads should have same priority which is 20 in default, the order of list sorted done in Task3 is 3, 15, 14, 4, 13, 12, 11, 5, 10, 6, 9, 8, 7, and the list done one by one. This is both affect by size of list and the order of threads created, but mainly by size.

And in Task4, the order of list done no significant change, but the list done several by several instead of one by one, though not all threads done at same time but some threads did. The larger list will have lower priority value, which is higher priority that can have more CPU assign, thus Task4 has higher run speed.

The thread creation of Task4 still in order of word length, if change to order of list size may get higher performance.

**Task 5.**

**Warning: Stream server do not outputs words to FIFO randomly, it still in input order, reason below:**

As the sort function feature explain in Task1, “***it will only detect characters after third character. Such as we’d and he’d will detected as same word and ignore the back one.”***

If output words randomly, the order of words will out of order and sort function will ignore different words, thus the output will different with normal order.

If wish to fix this, code of Task5 provided a new function **new\_compare\_2\_words()**, it will compare word in full word when they have same back section(such as **we’d** and **he’d** have same back section **’d**). Use **new\_compare\_2\_words()** to instead **compare\_2\_words()**, the effect of new function has been test(input word in order and randomly).

Do the same thing to fix problem in other Task(Use **new\_compare\_2\_words()** to instead **compare\_2\_words()**). **new\_compare\_2\_words()** only provided in code of Task5, please copy it to other Task.

To enable random output in Task5, comments line 583 and cancel comments of line 584.

The code of Task5 is modified from Task4, it mainly modify **main()** for **stream server** and **map5()**, and modify **TaskFilter()** that no longer open a fixed file to read and filtered words, it will receive a word list and filtered the list.

The priority assign is same as Task4, it not assign by historical ratios but assign by size of word list. Larger list has higher priority.

The program will create 2 threads for **stream server** and **map5()**. Stream server reads input from input stream by **std::cin**, the input stream here are send from **cat** **InputFile.** Then stream server will write words to FIFO file which **map5()** will read it. Stream server will not output words randomly but in input order, reason explain above. Stream server will first write length of word into FIFO file, then second write word in, this is for tell **map5()** what size of data it need to read.

The thread of **map5()** will open the FIFO file that **stream server** will write in, **map5()** once read 2 line, first is length of data it need read, then read word in corresponding size of data. **Map5()** and **stream server** will contact by global variable to know when close the FIFO file.

After **map5()** read all words, it will do exactly the same as Task4: filtered words and create 2 threads for **map4()** and **reduce5()**. The **map4()** is the code from Task4, and **reduce5()** is exactly the same as **reduce4()**;

Use **g++ Main5.cpp -o Task5.exe -Wall -g** to compile.

Use **cat** **InputFile | ./Task5.exe OutputFile** to run program.

The final output using **fc Task4\_final.txt Task5\_final.txt** to check, is same output with Task4.

Due to perf only measure the performance of first command(**cat** **InputFile**) and will not measure command after(**./Task5.exe OutputFile**), and also the assignment does not ask for performance analyze of Task5, thus there will not have performance measurement.