ASSESSMENT 1 REPORT

Operating Systems Principles

Jordan Hacking | s3723766
S3723766@student.rmit.edu.au

Contents

Task 1 (Filter)

This task was a baseline for the rest of the assessment, creating the definition of what a word is allowing for the creation of a bash function and C++ function to filter an inputted text document with a single word on each line.

Definition of a word

The definition of what the filter classifies as a word is as follows:

- A word must be between 3 and 15 (inclusive) characters long
- A word must only be lowercase characters
- A word must not have any numbers
- A word cannot contain any special characters
- A word cannot have a hyphen separating segments
- A word cannot contain more than 2 consecutive characters

Using this definition of a word we can denote the validity of the following words:

- do (length 2: invalid)
- dog (length 3: valid)
- happy (length 5: valid)
- uncharacterised (length 15: valid)
- uncharacteristically (length 20: invalid)
- happy (all lowercase: valid)
- SAD (all uppercase: invalid)
- mistake! (contains special character: invalid)
- football (all lowercase: valid)
- to-do (contains hyphen separating words: invalid)
- match5 (contains number: invalid)
- peTer (not all lowercase: invalid)
- good (2 consecutive characters: valid)
- keeper (3 consecutive characters: invalid)

Shell function

With this definition of a word, we can devise a shell bash script (**Task1.sh including in submission**) using grep, pipe, and sort to filter an input file and output only words which are by the definition above, valid.

```
#!/bin/bash
if [ ! $# -eq 2 ]
then
    echo "Missing file, usage: 'filter.sh INPUT_FILENAME OUTPUT_FILENAME'"
    exit 0
fi

if [ -e $1 ]
then
    grep -E "^[a-z]{3,15}$" $1 | grep -v -E "([a-z])\1{2}" | sort -u > $2
else
    echo "Input file does not exist, usage: 'filter.sh INPUT_FILENAME
    OUTPUT_FILENAME'"
```

```
fi
```

The first if statement in script is just checking the presence of both the input parameters, it wasn't stated as a requirement so I will skip over it. The second if statement is checking for the presence of the input file, then proceeds to perform a few unix commands, all brought together through the pipe ('|') command then streamed into the output file using ">".

1. **grep -E "^[a-z]{3,15}$"**
    a. This command is using grep to extract the lines of valid length, using -E allows grep to act like its counterpart egrep and accept extended regular expression (regex). The expression "^[a-z]{3,15}$" is a simple regex for any word with length between 3 and 15 inclusive that is only the range of characters between **a** and **z** (only lowercase characters in the alphabet).
2. **grep -v -E "([a-z])\1{2}"**
    a. as mentioned above grep -E has been explained, however, grep -v allows us to invert the regex, meaning match everything that is **NOT** in the regex. The regex ([a-z])\1{2} means any occurrence of the same letter consecutive more than twice. The string baaa would match in this expression, but because we're using the invert, it will **NOT** be included in the grep output.
3. **sort -u**
    a. sort -u simply sorts the file based on the first character but ignoring duplicate entries.

C++ Function

Both C++ files are included in the submission

Task1.cpp (entry point)

```
#include <iostream>

int TaskFilter(const std::string& input, const std::string& output);

int main(int argc, char * argv[]) {
    if(argv[1] == nullptr || argv[2] == nullptr) {
        std::cout << "Invalid usage: './Task1 INPUTFILE OUTPUTFILE'" <<
std::endl;
        return 0;
    }

    std::cout << "Using input file: " << argv[1] << std::endl;

    //If TaskFilter returns false, there was an error filtering
    if(!TaskFilter(argv[1], argv[2])) { return 0; }

    std::cout << "Filtering complete, file '" << argv[2] << "' created!" <<
std::endl;
}
```

TaskFilter.cpp (filtering function)

```
#include <sys/stat.h>
#include <unistd.h>
#include <fstream>
#include <iostream>
#include <algorithm>
#include <string>
```

```
#include <set>

/*
    checks a file exists and is accessible
*/
bool check_filename (const std::string& name) {
  struct stat buffer;
  return (stat (name.c_str(), &buffer) == 0);
}

/*
    This function take a string and checks if the length if valid,
    if a string is less than 3 or greater than 15 false is returned as it
is not valid
    if not true is returned
*/
bool valid_length(const std::string string) {
    if(string.length() < 3 || string.length() > 15)
        return false;
    return true;
}

/*
    This function takes in a string and checks it only contains lowercase
    characters
*/
bool only_lowercase_chars(const std::string string) {
    return string.find_first_not_of("abcdefghijklmnopqrstuvwxyz") ==
std::string::npos;
}

/*
    This function takes a string and loops through each character
    and checks if the next 2 letters are the same character
    if true then the string is invalid as it contains more than 2 conseq
    chars
*/
bool conseq_chars(std::string string) {
    for(int i = 0; i < (int)string.length() - 2; i++)
        if(string[i] == string[i+1] && string[i] == string[i+2])
            return true;
    return false;
}

/*
    TaskFilter takes in an input file and output file
    exported function for the file (entry point)
*/
int TaskFilter(const std::string& input, const std::string& output) {
    //Check if the input file is present and accessible, if no print usage
    if(!check_filename(input)) {
        std::cerr << "File '" << input << "' not found, usage: './Task1
INPUTFILE OUTPUTFILE'" << std::endl;
        return 0;
    };

    //Declare the input and clean files
    std::ifstream InputFile(input);
    std::ofstream CleanFile(output);
```

```
    //By using a set to contain the readLines we are ensuring they're no
    duplicate entries
    std::set<std::string> readLines;

    for(std::string curLine; std::getline(InputFile, curLine);) {
        //If the string is not between 3 and 15 (inclusive), skip
        if(!valid_length(curLine)) { continue; }

        //If the string is not only lowercase chars ([A-Z][0-9] and special
      chars, skip)
        if(!only_lowercase_chars(curLine)) { continue; }

        //If the string contains more than 2 conseq character, skip
        if(conseq_chars(curLine)) { continue; }

        //Pushing the string to the set handles the uniqueness of each line
        readLines.insert(curLine);
    }

    //Push all the lines that got inserted into the set to the clean file.
    for(std::string string: readLines) {
        CleanFile << string << "\n";
    }

    //Close the input and outfile file streams
    InputFile.close();
    CleanFile.close();

    return 1;
}
```

Performance Metrics

Throughout this report performance will be monitored using the '*time*' command from Unix, time is used by invoking the time command followed by the executable and any other parameters, for example:

time Executables/Task1 Wordlists/wlist_match1.txt Output.txt 1>Outputs/stdout.txt 2>Outputs/stderr.txt

will return three key metrics about the execution of Task1.

- **Real**: total execution time from invoking to termination of the executable
- **User**: the total CPU time taken up by the executed process
- **Sys**: the total CPU time taken up by the system on behalf of the executed process

```
jordan@DESKTOP-T33I6LN:~/s3723766_OSP_A1$ time Executables/Task1 Wordlists/wlist_match1.txt Output.txt 1>Outputs/stdout.
txt 2>Outputs/stderr.txt

real    0m0.434s
user    0m0.403s
sys     0m0.030s
```

For the reporting we will be mainly using the '*real*' time taken by the process from start to finish.

Shell

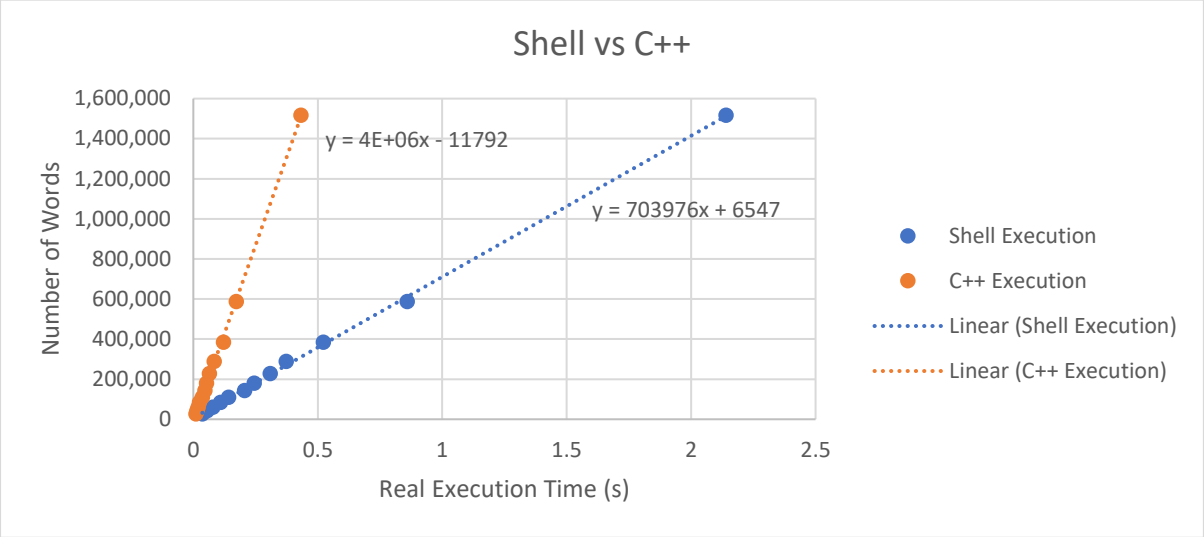
Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	2.14	708878
2	586,880	0.859	683213
3	384,111	0.522	735844
4	288,773	0.373	774190
5	227,779	0.309	737148
6	180,801	0.245	737963
7	143,317	0.206	695713
8	110,542	0.142	778464
9	83,922	0.11	762927
10	59,950	0.079	758860
11	42,896	0.056	766000
12	26,680	0.036	741111
		Average words per second:	**740025.9167**

C++

Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	0.432	3511571
2	586,880	0.172	3412093
3	384,111	0.121	3174471
4	288,773	0.084	3437773
5	227,779	0.065	3504292
6	180,801	0.053	3411339
7	143,317	0.047	3049297
8	110,542	0.036	3070611
9	83,922	0.025	3356880
10	59,950	0.019	3155263
11	42,896	0.014	3064000
12	26,680	0.01	2668000
		Average words per second:	**3,234,632.50**

Data Analysis

Using the table's above we can plot both performance metrics on a scatter plot to see how they match each other, and the trend lines they produce.



As you can see the C++ Execution time (orange) is much quicker than the shell (blue), as and as the number of words increase the C++ implementation hovers at around ~460% quicker.

We can use the trendline equations to predict how many words each method could filter when given 10 seconds.

C++: $\left((4 * 10^6) * 10\right) - 11792 = 39{,}988{,}208.00$

Shell: $(703976 * 10) + 6547 = 7{,}046{,}307.00$

The difference between the two is remarkable, C++ is able to process 32,941,901.00 more words in 10 seconds than the shell equivalent and is working (over 10s) 467.1% quicker.

Task 2 & 3 (Forking vs Threading)

In this task we were to implement a map and reduce sorting method using two different process/threading techniques. '*Fork*' is a utility function which creates a child process within the caller (parent), the child process runs concurrently to the parent, in this implementation the parent creates 13 child processes and waits for them to complete before reducing the outputs into one sorted file. '*pThread_create*' creates concurrent threads within the parent process, similarly, to fork 13 mapping threads are created, but also 13 reducing threads are created (as well as the 2 threads to initiate mapping and reducing).

Other than the different concurrency methods, the fork implementation is also using basic .txt file outputs for mapping state, whereas the thread-based implementation is using FIFO files and piping data from one mapping thread to the matching reduce thread before combining and sorting.

Raw Data

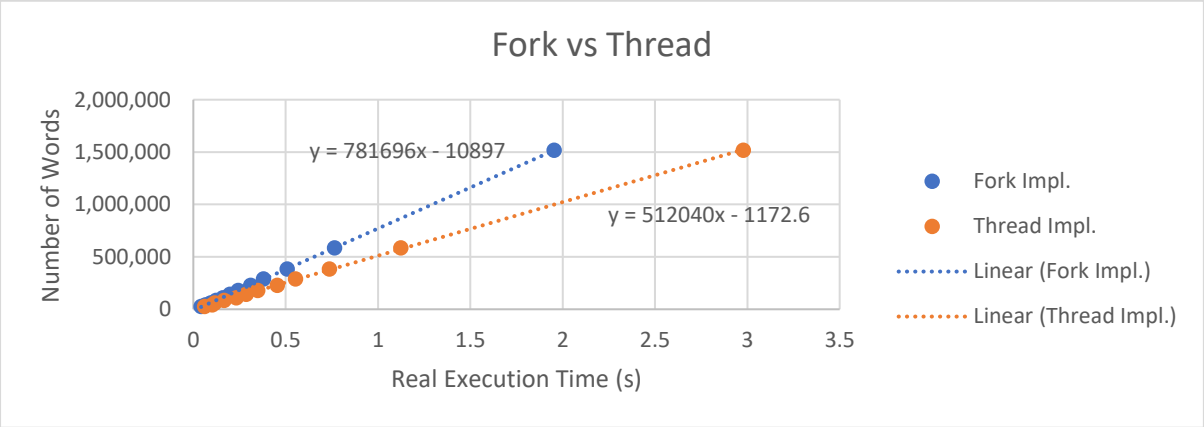
Fork Real Execution Time

Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	1.954	776355
2	586,880	0.764	768167
3	384,111	0.508	756124
4	288,773	0.38	759928
5	227,779	0.31	734770
6	180,801	0.243	744037
7	143,317	0.198	723823
8	110,542	0.159	695232
9	83,922	0.123	682292
10	59,950	0.093	644623
11	42,896	0.066	649939
12	26,680	0.042	635238
		Average words per second:	**714,210.67**

Thread Real Execution Time

Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	2.978	509401
2	586,880	1.124	522135
3	384,111	0.737	521181
4	288,773	0.553	522193
5	227,779	0.454	501715
6	180,801	0.349	518054
7	143,317	0.286	501108
8	110,542	0.233	474429
9	83,922	0.167	502526
10	59,950	0.117	512393
11	42,896	0.102	420549
12	26,680	0.061	437377
		Average words per second:	**495,255.08**

Data Analysis



As shown by the scatter plot above, the trendline for the fork implementation is much steeper than the trendline for the thread implementation, this mean that as more and more words are added to the file the fork implementation will outperform the threading counterpart. This proportionality is also shown in the raw data and the average words each method can process per second, the forking implementation being able to process approximately 220,000 more words per second.

Using the trendline equations we can also predict the performance of each implementation given a set time limit and see which method can process the most, given the steeper nature of the fork implementation's trend line we should be expecting it to continue to outperform.

Using a 60s base (meaning we are calculating how many words can be processed in 60 seconds)

Fork: $(781696 * 60) - 10897 = 46{,}890{,}863.00$

Thread: $(512040 * 60) - 1172.6 = 30{,}850{,}173.01$

As expected over the 60s the fork implementation continues to beat the threading implementation, it seems no matter how long the two-process run for the fork implementation will always be approximately 50% quicker than the threading implementation.

Task 4 (Thread Optimisation)

In this task we were to implement a map and reduce sorting method using two different process/threading techniques. '*Fork*' is a utility function which creates a child process within the caller (parent), the child process runs concurrently to the parent, in this implementation the parent creates 13 child processes and waits for them to complete before reducing the outputs into one sorted file. '*pThread_create'* creates concurrent threads within the parent process, similarly, to fork 13 mapping threads are created, but also 13 reducing threads are created (as well as the 2 threads to initiate mapping and reducing).

Other than the different concurrency methods, the fork implementation is also using basic .txt file outputs for mapping state, whereas the thread-based implementation is using FIFO files and piping data from one mapping thread to the matching reduce thread before combining and sorting.

Raw Data

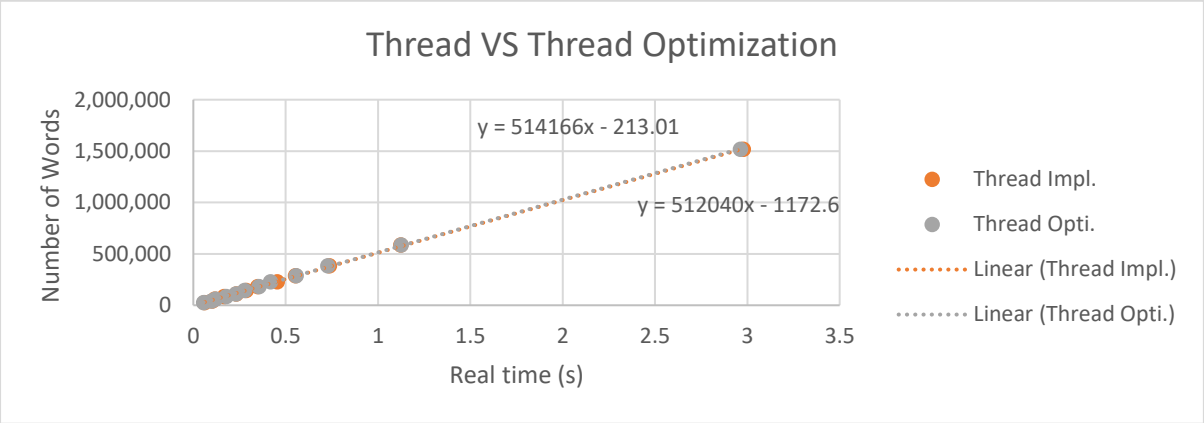
Thread Real Execution Time

Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	2.978	509401
2	586,880	1.124	522135
3	384,111	0.737	521181
4	288,773	0.553	522193
5	227,779	0.454	501715
6	180,801	0.349	518054
7	143,317	0.286	501108
8	110,542	0.233	474429
9	83,922	0.167	502526
10	59,950	0.117	512393
11	42,896	0.102	420549
12	26,680	0.061	437377
		Average words per second:	**495,255.08**

Thread Optimized Real Execution Time

Wordlist	Number of words	Time taken (*real*) in seconds	Words per second
1	1,516,999	2.963	511980
2	586,880	1.126	521207
3	384,111	0.729	526901
4	288,773	0.556	519375
5	227,779	0.418	544925
6	180,801	0.356	507867
7	143,317	0.278	515528
8	110,542	0.23	480617
9	83,922	0.179	468837
10	59,950	0.12	499583
11	42,896	0.099	433292
12	26,680	0.055	485090
		Average words per second:	**501,266.83**

Data Analysis



As shown by the scatter plot above, the optimized threads are slightly better performing than the non-optimised, over the 12 word lists the optimized threads are able to get approximately 6,000 more words done per second.

Using the trendline equations we can also predict the performance of the two threading methods and deduce the better performing method based on a 60 second interval, given that we have given the optimized threads a higher priority we should be expecting to see it outperform.

Using a 60s base (meaning we are calculating how many words can be processed in 60 seconds)

Threading: $(512040 * 60) - 1172.6 = 30{,}721{,}227.40$

Threads Optimized: $(514166 * 60) - 213.01 = 30{,}850{,}173.01$

As expected over the 60s the optimized threading implementation continues to beat the non-optimized threading implementation, it seems no matter how long the two-process run for the optimized implementation will always be approximately 1% quicker than the threading implementation.

Additional Remarks

The majority of the threads time is spent in the mapping phase and waiting/sending data through to the reduce pipes. With some further optimizing of the piping methods, it could be possible to improve the performance more.

To calculate the BigO notation of the code, I split each step into their own individual notation, mapping came out to have a notation of $O(n + n^2 + n^2)$. Reduce came out to be $O(n + n + n + n^2)$

Taking the largest, the BigO notation of the Task 3 and 4 implementation is $Big\ O\ (n^2)$

As the string length would increase, the comparator function which takes the sub string from character 3 onwards would definitely take longer as the number of characters to compare to would increase, increases of small amounts wouldn't have a great impact on performance, however, if they were large increases say 200 letter words, I would definitely expect a performance hit.

Closing Thoughts

The task outlined was to when given a dirty word list, clean it and sort it based on the third character, the first step was to define what my process deemed “a word” then create the shell (bash) and C++ functions to filter this data, the C++ implementation proved 460% quicker than the shell implementation.

We then went forward to using a map and reduce methodology to sort the new clean file created in task1 and sort it based on the third character and using fork to create 13 child processes to map each of the word lengths into their own text file and then reduce (merge sort) them into one output file, the same logic was implemented using threads and FIFO files (pipes). The data showed that the fork’ing process was quicker than the threading by approximately %50. With some tweaking in task 4 we were able to squeeze another 1% of performance using the ‘*nice*’ thread priority settings.