CS 5523 Operating Systems Project 1A

Implementing Word Count using C Language

A Project by Venkat Rahul Dantuluri fpj626 @01518869

Code Design:

The file named wordcount.c contains the source code to my implementation of counting the number of valid words in a given file. This program takes the input file and prints the top recurring words in decremented fashion. The output comprises of the word followed by the number of times it has occurred in the file. The code is written in C language and compiled using standard gcc compiler that comes with every Ubuntu distribution. For in-depth debugging I used GDB debugger.

argv[0] – Name of the executable (a.k.a "wordcount")
argv[1] – Fully qualified name of the input file containing the words
argv[2] – Number of lines to printed in the output fashion as described above.

If these input arguments are not provided the program will exit printing the appropriate message.

If these above arguments are provided appropriately, the code will try open the file. If successful, the code will proceed to create a virtual address for the content of the file. For this I used to mmap system call. During the progress of code execution until this state, if there are any exceptions or errors in any system calls, we simply print the appropriate error message and exit.

If no errors occur, we have a chunk of addressable file content in the virtual address. We split this chunk among the number of threads we create. The number of threads is defined as a macro in the code for ease of configuration. i.e I can change the macro and recompile the code to achieve a higher or lower thread count.

After splitting the chunk among the given number of threads each thread will now execute the "runner" method independently to count the number of valid words in the chunk of file allocated to it. Once a valid word is found each thread tries to access a common list data structure to store the valid word (if valid word is new) or update its occurrence count (if valid word already present in the list). This list data structure is shared among all the threads and hence is protected by a mutex semaphore protects the data structure from race conditions.

At the end, once all the threads have finished updating the list with all the valid words identified in their chunk, we sort the list in decreasing order of occurrences. For this I implemented the mergesort algorithm. I chose the Merge-sort algorithm because of its ease of implementation (which is a priority when dealing with pointers) and because of its run-time complexity which is O(n log n). Point to note: The main thread is halted until all the threads have exited after successful execution. Only then will the main thread proceed with sorting the list.

I used standard pthread library to implement multi-threading. The code is clearly commented on how this multithreading is done using pthread_create call.

Now we will discuss the algorithm used to find the valid words.

1) Iterate the character pointer to the first valid character.
2) Continue iterating until you receive a ' ' or a '\n'. i.e an empty space or a newline.
3) In course of finding a new line or empty space, if an invalid character is identified make the "validword" flag false.
4) Once you reach empty space or new line, check the validword flag. If true then we identified a valid word. Now add it to list.
5) Else repeat steps 1 through 5 until end of file.

Performance Comparison:

I used clock_t type and clock() present in the time.h header to find out the number of clock cycles taken to execute the program. Then I divided that amount with Number of clock cycles per seconds defined by the CLOCKS_PER_SEC constant to compute the execution time in seconds. This execution time is printed

Number of thread	10MB input file	50MB input file	100MB input file
1 (single thread)	26.801830 sec	212.394390 sec	633.193784 sec
2	22.960124 sec	140.033147 sec	460.65856 sec
8	19.4345 sec	217.593991 sec	711.8751 sec

This table shows a comparative picture of the execution times of the same program on 3 input files given for testing. These input files are 10, 50 and 100 MD respectively. The table also compares the performance of program using higher thread count. To neutralize the cache effects I averaged the execution times. This execution time is machine dependent and may not be accurate to the point on other machines.

By default the program runs with 2 threads. If you want to change the number of threads we need to make changes to 'NUM_OF_THREADS' macro definition in code and recompile the source file using the make file provided.

Despite using multithreading we may not observe expected increase in throughput because I am doing sorting at the end. Sorting is a time consuming process. And despite n log n runtime of merge-sort we still have higher execution times.

Clearly from the above tabulated values we understand that simply adding more threads wont help. In my program two threads execute a huge file much better than 8 threads.