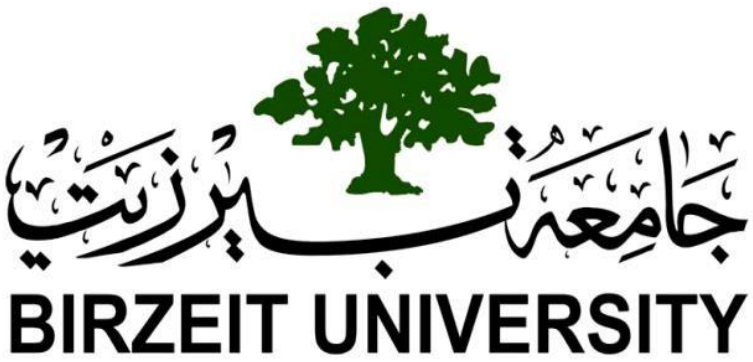


Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Operating System Concepts – ENCS3390

Project #1

Multi- Processing and Multi- Threading Programming

Prepared by: Khaled Abu Lebdeh

ID Number: 1220187

Instructor: Dr. Yazan Abu Farha

Section: 1

Date: November 25, 2024

Abstract

Programmers used to write sequential codes. However, large or heavy programs may take long execution time even if they implemented in optimal algorithms.

Sequential code does not take the full advantage of multi- core systems. On the contrary, there are different approaches use multi- core systems to increase the total throughput, like Multi- Processing and Multi- Threading.

Table of Contents

Contents

Procedure

1- Computer Environment

- The computer used to execute programs has (intel core i5, 8th Gen) processer. Processer's base (guaranteed) speed is 1.8 GHz. However, the average speed while execution was about 3.2 GHz.
- A virtual machine with "*Linux System*" was used with 5-cores and 4 GB memory.
- Programs were written in "*C- Language*" and ran within Linux terminal.

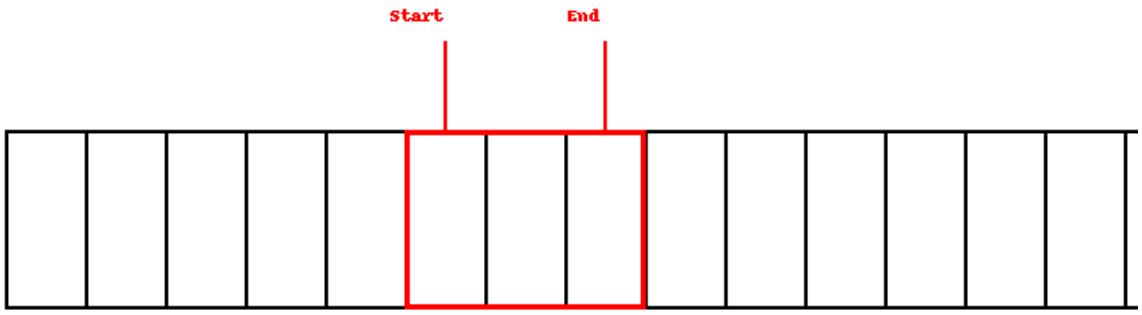
2- Multiprocessing and Multithreading Requirements

2-1 Multiprocessing Achievement

The algorithm used was to load words into an array of strings. Then, processes were created using **fork**() API, each process was given a unique number named as "process_num" which determines start and end indices of words array the process must work on.

start= (number of array cells)*(process_num)/(number of total processes created)

end= (number of array cells)*(process_num + 1)/(number of total processes created)



A shared memory was created using **shm_open**() and **mmap()** APIs to allow processes to communicate and store results gotten from all processes. Shared memory took form of a two- dimensional array, each process found the

frequency of each word in its data slot, and stored the result in the shared memory as a struct (a word and its frequency). Each process wrote on one row of the 2-D array only, to avoid race condition.

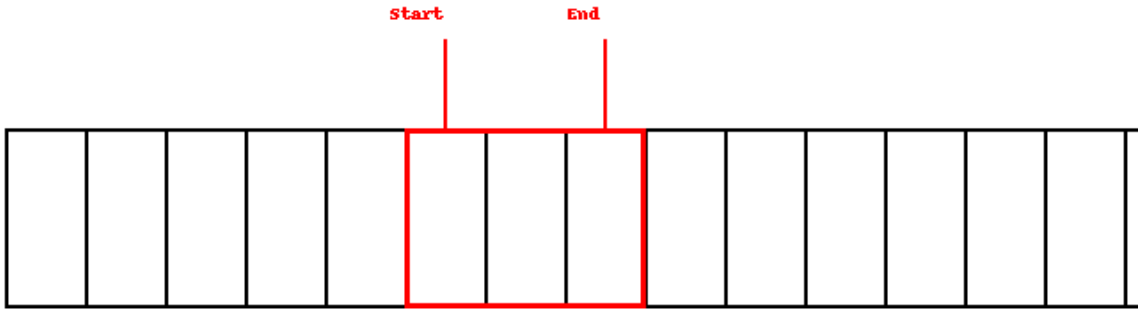
Children processes used **exit**() API after finishing execution, to terminate. While children processes hadn't finished executing, the parent process was waiting for them using **wait**() API. After that, the parent merges children's results using a *Binary Search Tree* (BST) data structure to find total frequency of each word.

Parent process used Min- Heap data structure to find top 10 frequent words in the BST containing all words with their frequencies. For each tree node, we enqueued the word into the heap if its frequency is larger than min frequency in heap. So, the heap then contained the top 10 frequent words among all words!

APIs used in Multiprocessing	Description
fork()	Creates child process.
exit()	Process uses it to terminate.
wait()	Parent process uses it to wait for children processes.
shm_open() and mmap()	Creates shared memory.
memset()	To set initial values, it was used in the code to reset shared data to zeros.
malloc()	Allocates memory during runtime.
free()	Deallocates memory allocated during runtime.

2-2 Multithreading Achievement

The algorithm used was to load words into an array of strings. All variables and arrays needed in threads were declared in the global scope. Then, an array of *pthread_t* type (from *pthread.h* library) was declared to hold threads identifiers (thread_ID). Threads were created in a loop using **pthread_create**() API and given a unique number "thread_num" which determines start and end indices of words array the thread must work on. Another loop was needed to wait for threads' termination using **pthread_join**() API.



Each thread executed a function usually named as runner(), passed as an argument through **pthread_create**() API. The runner() functions called user-defined functions needed to count words within thread's data slot. Runner function initialized temporary array to hold counted frequency words in its data slot, to avoid writing on global data and causing race condition. Binary Search Algorithm was used for optimal words search. After all threads had terminated using **pthread_exit**() API, the total frequency was merged into the global data array using BS. Then a Min- Heap was used to find top 10 frequent words among the merged array (similar to one in Multiprocessing).

- AI was used to add BS algorithm to the code

APIs used in Multithreading	Description
pthread_create()	Creates thread.
pthread_exit ()	thread uses it to terminate.
pthread_join()	Wait for threads to terminate.
malloc()	Allocates memory during runtime.
free()	Deallocates memory allocated during runtime.

3- Results and Performance of approaches used

The task is to find top 10 frequent words in enwik8 dataset. We found results using three different approaches, Naive approach (a program that does not use any child processes or threads), Multiprocessing approach (a program that uses multiple child processes running in parallel) and Multithreading approach (a program that uses multiple joinable threads running in parallel).

Output:

```
1 | the    --   1060930 times
2 | of     --   593543 times
3 | and    --   416437 times
4 | one    --   411559 times
5 | in     --   372213 times
6 | a      --   326320 times
7 | to     --   316319 times
8 | zero   --   264789 times
9 | nine   --   250239 times
10| two    --   192535 times
```

Approach	Execution Time
Naïve	45 minutes
Multi- processing (2 processes)	27 minutes
Multi- processing (4 processes)	16.19 minutes
Multi- processing (6 processes)	15.1 minutes
Multi- processing (8 processes)	12.2 minutes
Multi- threading (2 threads)	28 seconds
Multi- threading (4 threads)	16.7 seconds
Multi- threading (6 threads)	16.3 seconds
Multi- threading (8 threads)	15.2seconds

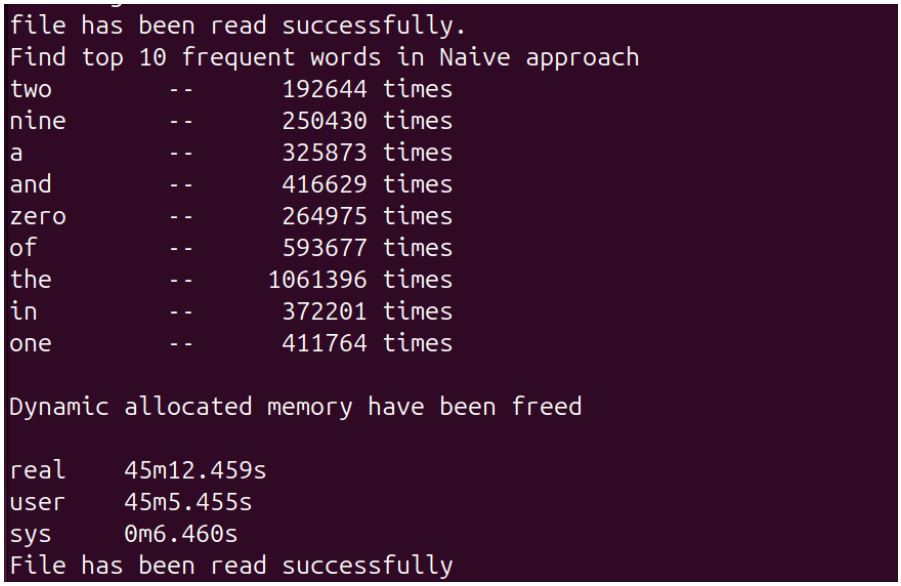


Figure 1 : Naïve Approach's output

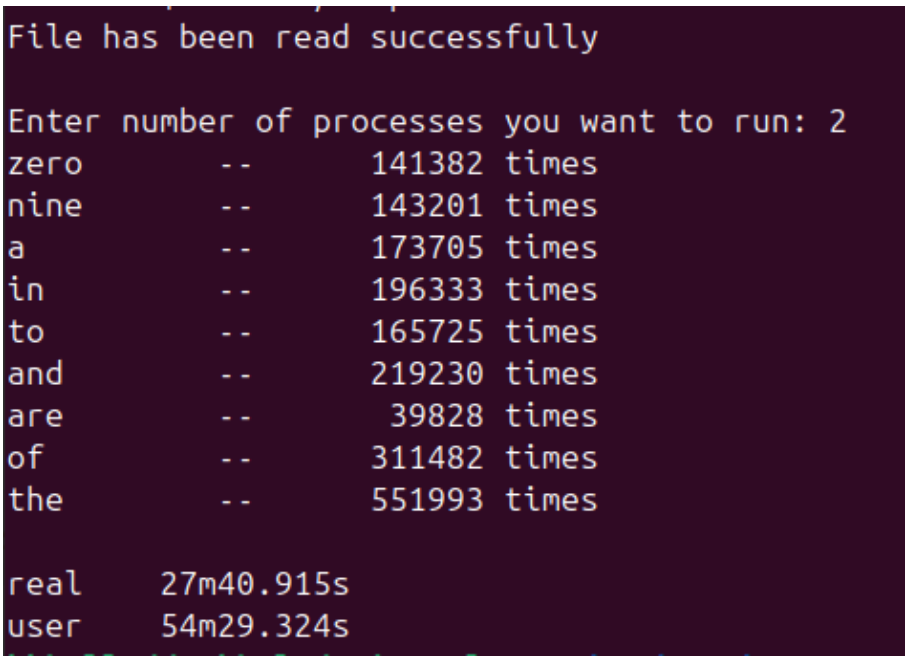


Figure 2 : Multiprocessing Approach's output (2 processes)

Figure 3 : Multiprocessing Approach's output (4 processes)

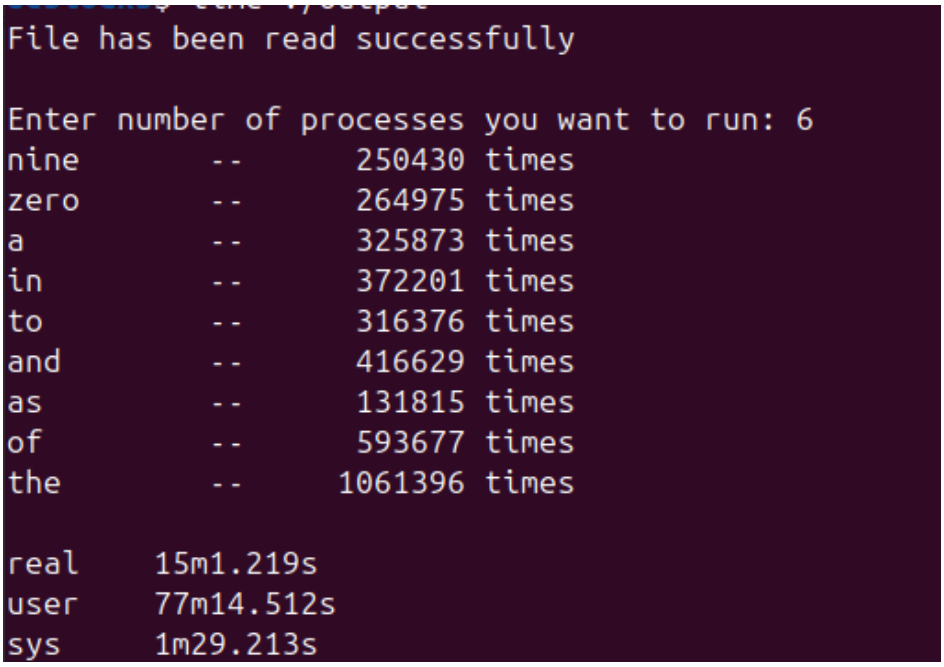


Figure 4 : Multiprocessing Approach's output (6 processes)

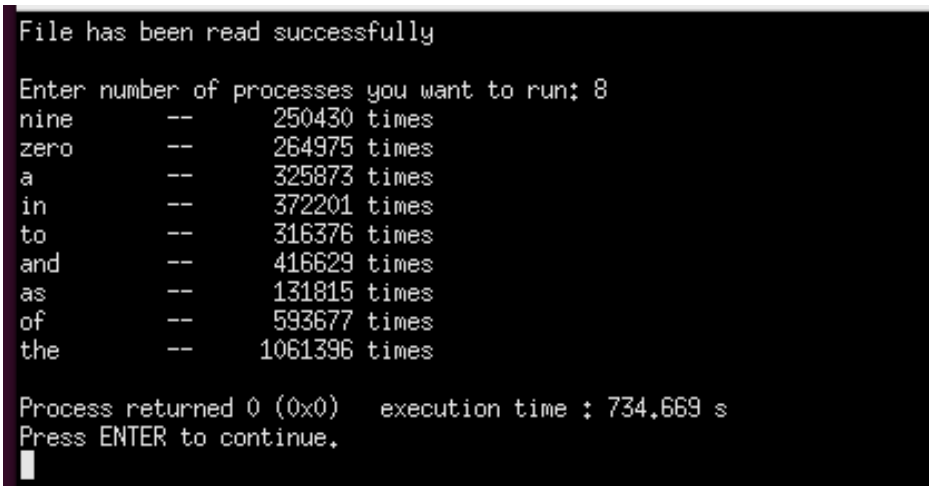


Figure 5 : Multiprocessing Approach's output (8 processes)

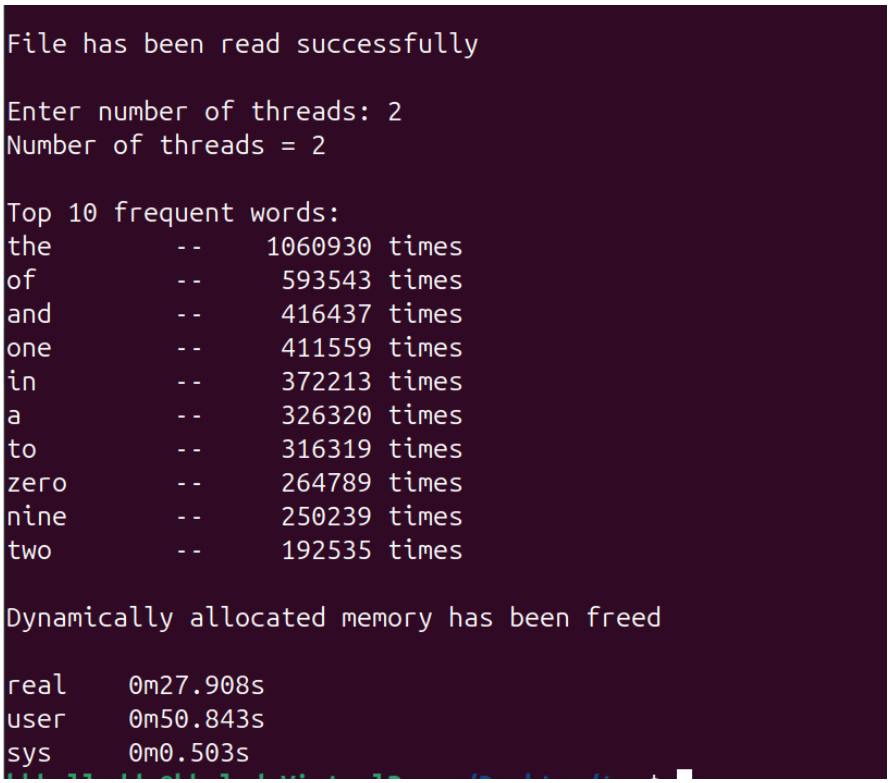


Figure 6 : Multithreading Approach's output (2 threads)

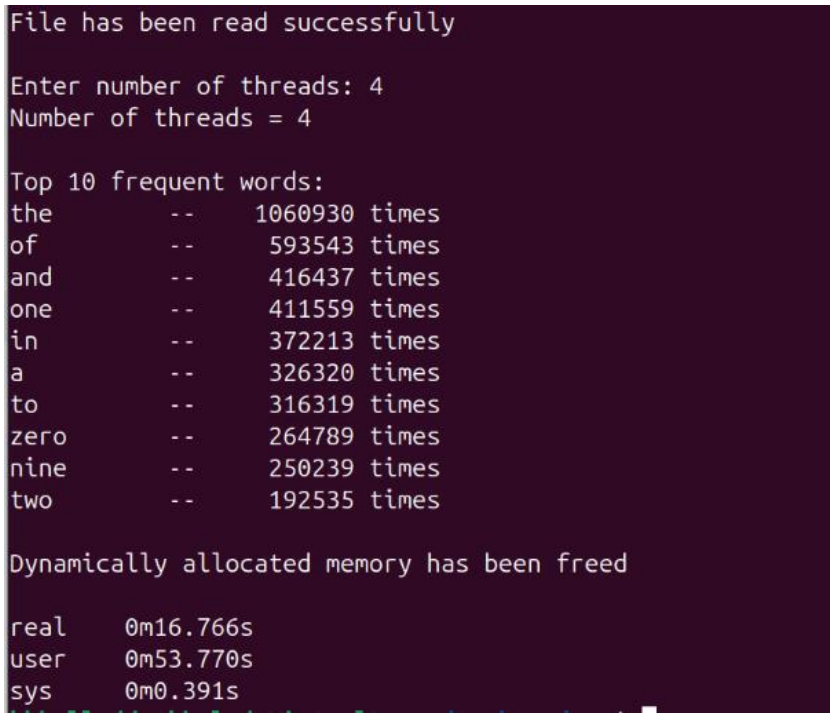


Figure 7 : Multithreading Approach's output (4 threads)

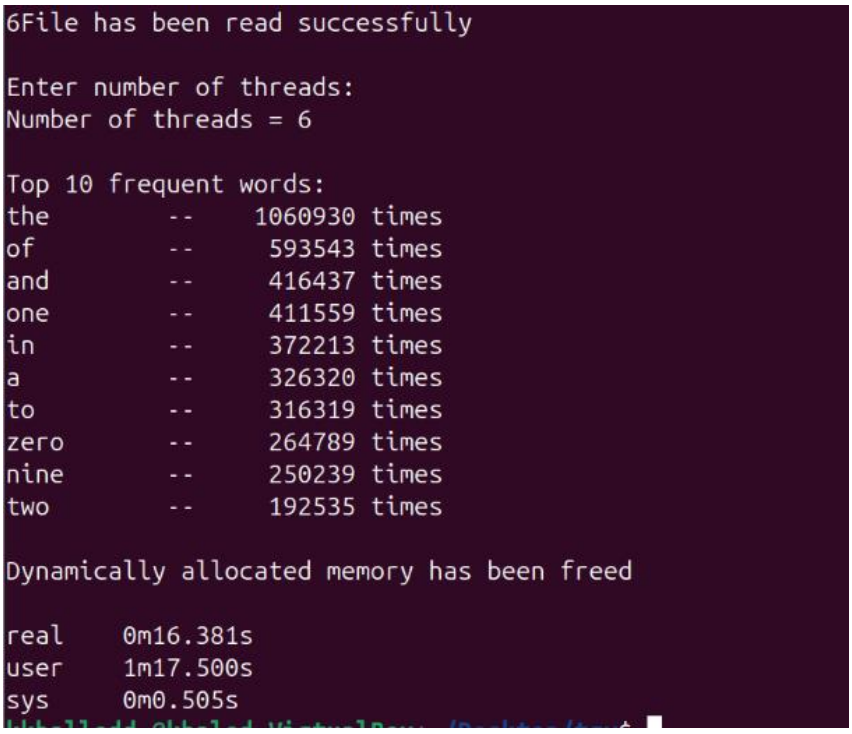


Figure 8 : Multithreading Approach's output (6 threads)

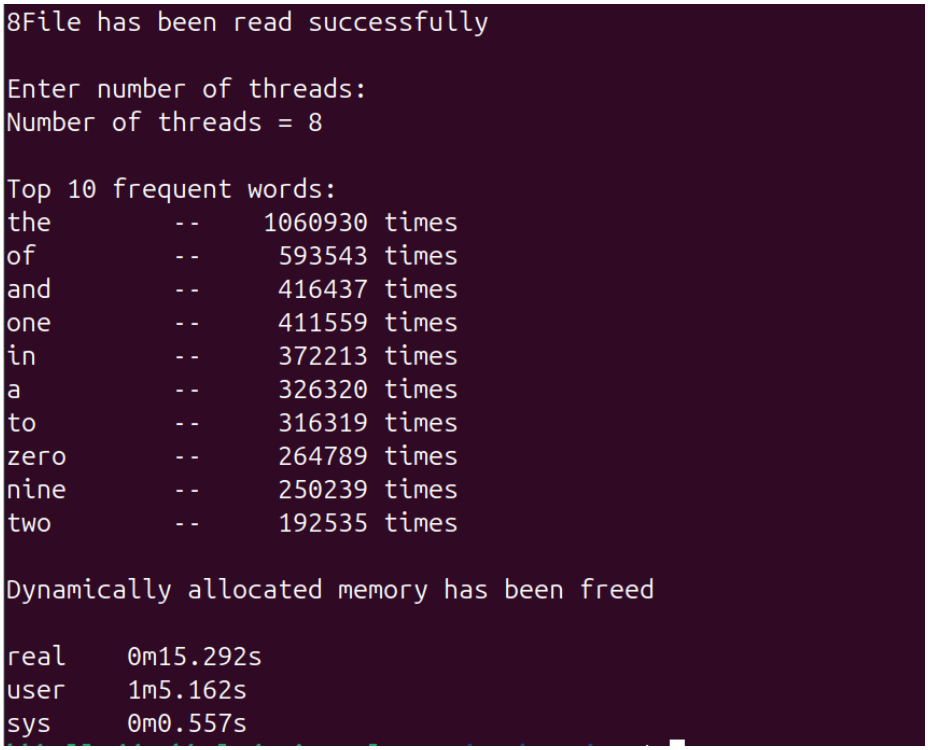


Figure 9 : Multithreading Approach's output (8 threads)

4- Analysis using Amdahl's Law

Amdahl's Law identifies performance gains from adding additional cores to an application that has both serial and parallel components.

$$\text{Speedup} <= \frac{1}{S + \frac{1-S}{N}}$$

S: is serial portion of code.

P: is parallel portion of code.

N: number of cores

P= (1-S).

- To find the series portion of code, we measure time before and after parallel section of code, then divide by the total time to get the series portion.
  We found that **S=12%** approximately. So, **P= 100% - S%= 88%**

The reason for the high portion of parallel code is that complex operations are done in parallel. Only few tasks were implemented in series, like reading the file and printing the top 10 frequent words. The more parallel portion leads to more efficiency on multi-core systems.

- The max speedup according to the available number of cores occurs in ideal scenarios if S=0% and P=100%

  Speedup <= N times, which means **Speedup <= 5 times**

- The optimal number of threads or processes when working on a system with 5 cores is actually about (4 or 5) threads or processes. The reason is that no more than 5 threads or processes can work in parallel, so they allocate resources with no worth improvements.

5- Comments on different approaches

- In Naïve approach, the execution time was relatively high, since there is no more than one thread of execution running at a time.

- In Multiprocessing approach, starting with two parallel processes, the improvement and difference in terms of execution time was very clear. The same scenario occurred with four parallel processes. However, increasing in terms of number of processes does not guarantee much improvement in execution time, since there are only 5 cores allocated to virtual machine. If there are more than five processes, they can't all run in parallel. The system can run five parallel processes at max. That resulted in no much improvement when six or eight processes where used.

- In multithreading approach, the execution time was incredible improved than both Naïve and Multiprocessing approaches. The reason is that Multithreading supports parallel execution with light- weight processes. Thread creation is mush saver than process creation. Moreover, thread switching is lower overhead than process context switching. In addition, threads share code section and global scope section, so no shared memory needed to communicate between threads. However, as same as in multiprocessing, increasing in terms of number of threads does not guarantee much improvement in execution time. It depends on the system used.

  In general, the relation between multiple threads of execution and speedup isn't liner. It is somehow an exponential relation. (assume code with parallel portion).

Conclusion

Today's computers are interactive ones, dealing with user interactions and different I/Os. That led to the motivation of throughput improvement. Many tasks are real- time ones, which must be executed in specific turnaround time.

Completely sequential programs (with 0% parallel portion) probably do not get the advantage of today's multi- core computers. So, it is somehow required to move towards parallel execution.

Parallel execution could be implemented by multiple processes or threads. If the program can be implemented using threads, it is much better than parallel processes. Parallel execution often satisfies users required throughput. However, explicit parallel coding could be difficult for programmers. Programmers should be responsible for dividing activities, load balancing and data dependency. That was a motivation to use implicit threading!