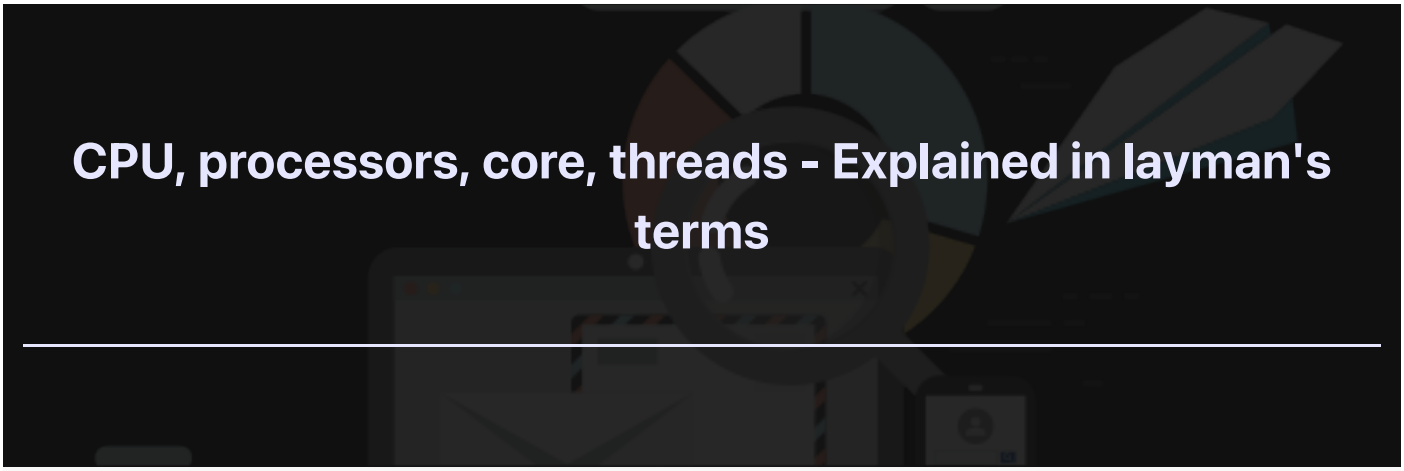


CPU, processors, core, threads - Explained in layman's terms

Table of Contents

If you are an IT Administrator then it is possible that you may have once or more scratched your head wondering what the hell is the difference between CPU, Processors, Core, Threads etc. All of these sound so related and yet they are completely different. When it comes to performance optimization then you should be familiar with these terms. So let me help you understand all these terminologies related to CPU so you can make a wise decision next time you have to work on these topics.

I have already covered the basics of memory management in Linux, so I will only concentrate on the CPU related explanation here.

Understanding CPU Model and Architecture

- CPU is considered as the "heart" of the machine – it reads in, decodes, and executes machine instructions, working on memory and peripherals. It does this by incorporating various stages.
- CPUs and other resources are managed by the kernel, which runs in a special privileged state called system mode.
- In Linux you can use `lscpu` or `cat /proc/cpuinfo` to get the CPU architecture details of your underlying platform/hardware.

Following is a sample output of lscpu:

```
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Stepping:              2
CPU MHz:               2600.000
CPU max MHz:           2600.0000
CPU min MHz:           1200.0000
...
```

This is from RHEL 8 server installed on a physical hardware (HPE ProLiant Blade). Here my CPU Model is "Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz". Let us understand each of these fields:

Core(s) per socket

Core(s) is a hardware term that describes the number of independent central processing units in a single computing component (die or chip).

To get the list of physical cores you can use:

```
 # grep 'core id' /proc/cpuinfo | sort -u
core id         : 0
core id         : 1
core id         : 2
core id         : 3
core id         : 4
core id         : 5
core id         : 6
core id         : 7
```

Now a **socket** is the interface between CPU and the motherboard and since I have two sockets so basically I have 8 cores of CPU in each socket so there are total **16** physical cores available on this server.

```
Core(s) per socket:    8
Socket(s):             2
```

You can also get this information using:

```
# dmidecode -t 4 | grep -E 'Socket Designation|Count'
        Socket Designation: Proc 1
        Core Count: 8
        Thread Count: 16
        Socket Designation: Proc 2
        Core Count: 8
        Thread Count: 16
```

Thread(s) per core

A Thread, or thread of execution, is a software term for the basic ordered sequence of instructions that can be passed through or processed by a single CPU core. A core with two hardware threads can execute instructions on behalf of two different software threads without incurring the overhead of context switches between them.

Here on my server I have 2 threads per core:

```
Thread(s) per core:    2
```

The number of logical cores, which equals "**Thread(s) per core" × "Core(s) per socket" × "Socket(s)**" i.e. **2x8x2=32** so this server has a total of 32 logical cores. You can check this also

using:

```
# nproc --all
32
```

CPU max MHz and CPU min MHz

Based on `CPU max MHz` and `CPU max MHz`, the **clock rate** i.e. the maximum frequency of this CPU is 2.6 GHz while the minimum threshold is 1200 MHz.

Understanding CPU Cache

CPU cache is pretty important for the efficiency of dealing with applications by the CPU. There are different levels of cache memory and they are used to keep data that needs to be close to the CPU. The cache level which is more closer to the CPU is the most expensive.

L1 Cache

- The L1 cache has the smallest amount of memory (often between 1K and 64K) and is directly accessible by the CPU in a single clock cycle, which makes it the fastest as well.
- This is built-in and it store the most frequently used data and remain in L1 until some other thing's usage becomes more frequent than the existing one and there is less space in L1.
- If so, it is moved to a bigger L2.
- Level 1 Cache can also be shared between the hyperthreading cores

L2 Cache

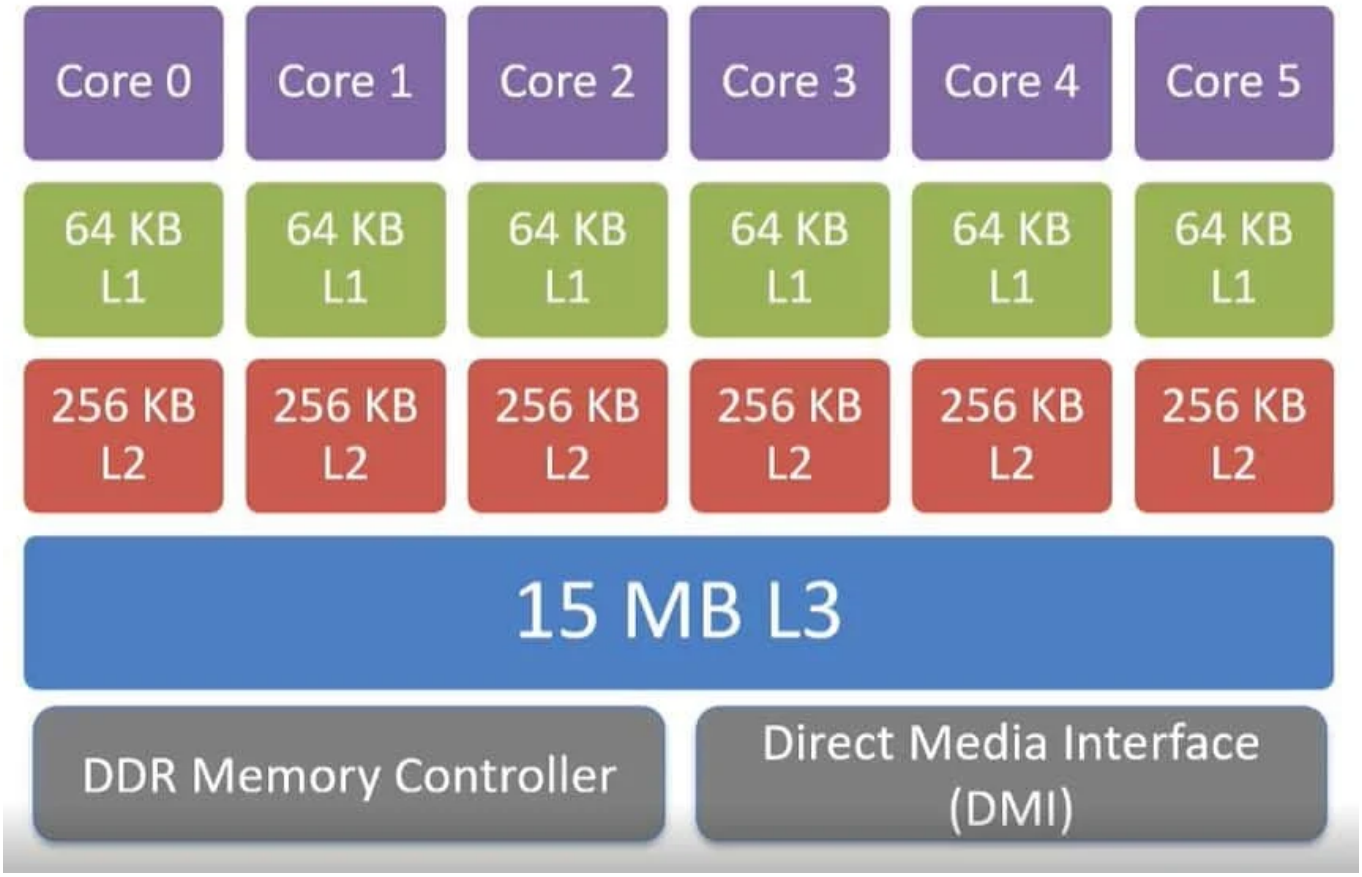
- The L2 cache is the middle level, with a larger amount of memory (up to several megabytes) adjacent to the processor, which can be accessed in a small number of clock cycles.
- This applies when moving things from L2 to L3.

L3 Cache

- The L3 cache, even slower than L1 and L2, may be twice as fast as the main memory (RAM).
- Each core may have its own L1 and L2 cache; therefore, they all share the L3 cache.
- Size and speed are the main criteria that change between each cache level: L1 < L2 < L3.
- Whereas original memory access may be 100 ns, for example, the L1 cache access can be 0.5 ns.

Understanding Cache Architecture

Let's try to understand the cache architecture with the following image of 6 core processor:



- In this image, each core here gets 64KB of L1 cache, 256KB of L2 Cache and 15MB of L3 Cache which is shared across all the cores. After that head out to either DDR Memory Controller or Direct Media Interface (DMI) depending upon where the information has to go.
- Cache is organized in lines, and each line can be used to cache a specific location in memory.
- Each CPU has its separate cache and its own cache controller.
- And if a processor references main memory, it first checks cache for the data. If it's there, then it's referred to as a cache hit. If it's not there, then you've got a cache miss.
- A cache line fill occurs after a cache miss, and it means that data is loaded from main memory.
- So, a cache miss involves extra activity and that means from a performance perspective, it's not dead good.

Write-through and Write-back cache

- Now there is also something called write-through and write-back cache.

- If **write-through** cache is enabled, when a line of cache memory is updated, the line is updated in memory as well, and that is efficient.
- If **write-back** is enabled, the write to cache is only written to main memory at the moment that the cache line is deallocated.
- So, write-back is more efficient and write-through ensures a higher state of stability.
- So, as often in performance optimization, it's a choice again between stability and efficiency.
- If on multi-CPU systems changes are not committed to memory immediately, the other CPUs need to be updated that something is changed if they are changing it also.
- So, on multi-CPU systems, write-through versus write-back poses another problem. This problem is referred to as **cache snooping**. Cache snooping is a hardware feature that ensures that all CPUs have access to the most up to date information in cache.
- There are other cache features to consider as well, like **direct mapped cache**. This means that each line of cache can only cache a specific location in main memory and that is a cheaper solution.
- **Fully associated cache** means that a cache line can cache any location in main RAM. That's more flexible, and for that reason, it's more expensive.
- And **set associative cache** offers a compromise between direct mapped and fully associated cache, and allows memory location to be cached into any n lines of cache, where n can be a number like two, for instance.

And these are features that you might want to consider if you are purchasing new hardware.

Analyse CPU cache misses using Cachegrind

You can use valgrind tool with **Cachegrind** to check the cache misses with your software or system applications.

You can execute valgrind as shown in the following snippet with your application:

```
# valgrind --tool=cachegrind ls
==28266== Cachegrind, a cache and branch-prediction profiler
==28266== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==28266== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==28266== Command: ls
==28266==
--28266-- warning: L3 cache found, using its data for the LL simulation.
```

```
some_rpm-17.0.1-115.03.noarch.rpm        failed_service.py
cust_collection_valid.conf               node_list
cust_collect_valid1.conf                 testlc.conf
==28266==
==28266== I   refs:      612,284
==28266== I1  misses:      1,635
==28266== LLi misses:      1,529
==28266== I1  miss rate:    0.27%
==28266== LLi miss rate:    0.25%
==28266==
==28266== D   refs:      240,476  (162,433 rd   + 78,043 wr)
==28266== D1  misses:      5,889  (  4,707 rd   +  1,182 wr)
==28266== LLd misses:      4,008  (  2,950 rd   +  1,058 wr)
==28266== D1  miss rate:     2.4% (    2.9%     +    1.5%  )
==28266== LLd miss rate:     1.7% (    1.8%     +    1.4%  )
==28266==
==28266== LL refs:         7,524  (  6,342 rd   +  1,182 wr)
==28266== LL misses:       5,537  (  4,479 rd   +  1,058 wr)
==28266== LL miss rate:      0.6% (    0.6%     +    1.4%  )
```

I have just execute 'ls' command which shows that it was missed in L1 cache 2.4% of the time and 1.7% in L2 Cache. Currently also this program was found in L3 cache. Instruction data was fetched properly at the level 1 cache almost all the time, except for **0.27%** of the time and with a cache miss rate of **0.25%** at L2 cache.

Sometimes, your software will do something the processor wasn't built to predict, and the data will not be in the L1 cache. This means that the processor will ask the L1 cache for the data, and this L1 cache will see that it doesn't have it, losing time. It will then ask the L2, the L2 will ask the L3, and so on. If you are lucky, it will be on the second level of cache, but it might not even be on the L3 and thus your program will need to wait for the RAM, after waiting for the three caches.

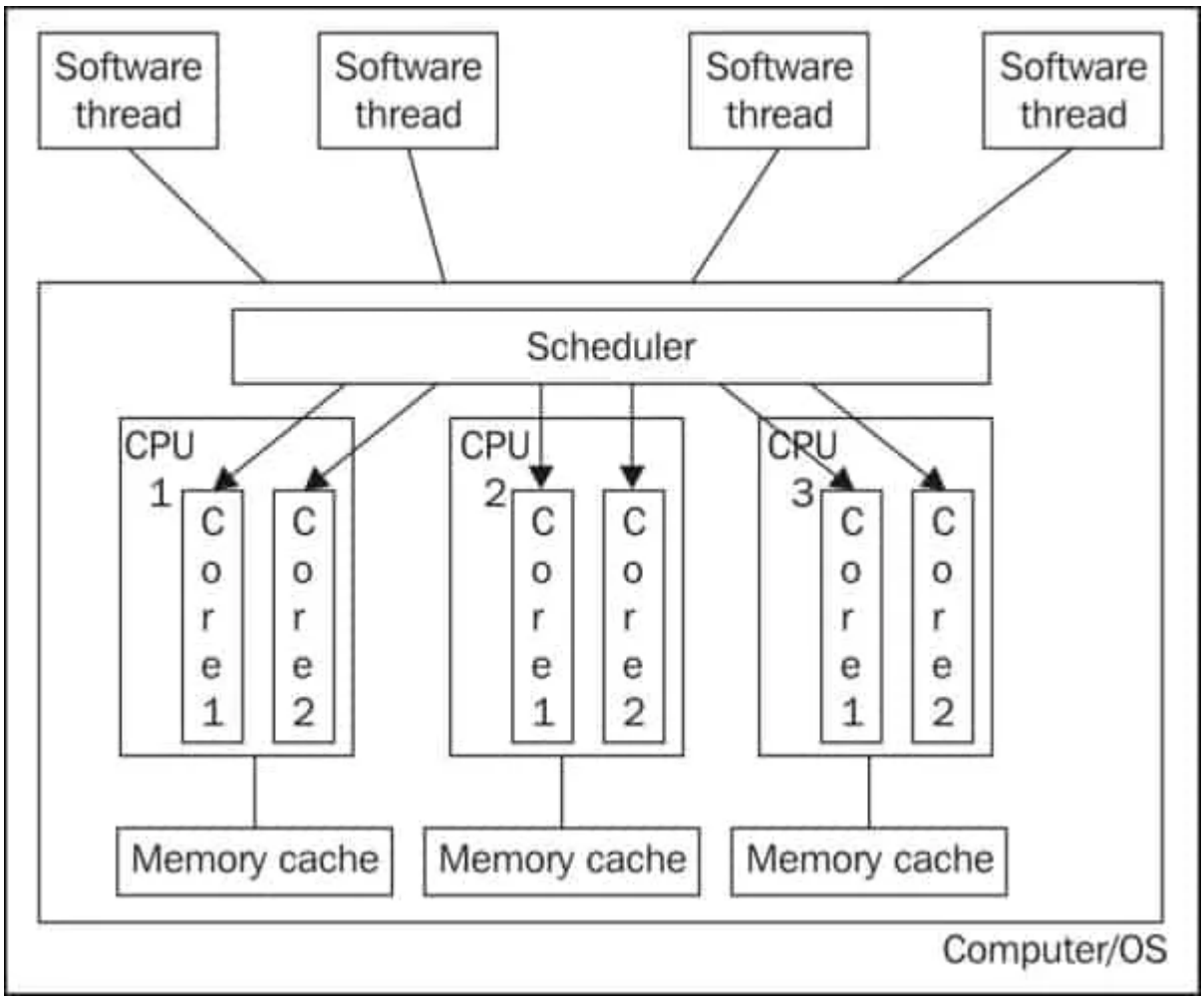
This is what is called a **cache miss**. How often does this happen? Depending on how optimized your code and the CPU are, it might be between 2% and 5% of the time. It seems low, but when it happens, the whole processor stops to wait, which means that even if it doesn't happen many times, it has a huge performance impact, making things much slower

Multi-core processor

- A multiple core CPU has more than one physical processing unit. In essence, it acts like more than one CPU.
- The only difference is that all cores of a single CPU share the same memory cache instead of having their own memory cache.

- From the multithreaded parallel developer standpoint, there is very little difference between multiple CPUs and multiple cores in a CPU.
- The total number of cores across all of the CPUs of a system is the number of physical processing units that can be scheduled and run in parallel, that is, the number of different software threads that can truly execute in parallel.

This diagram shows three physical CPUs each having two logical cores:



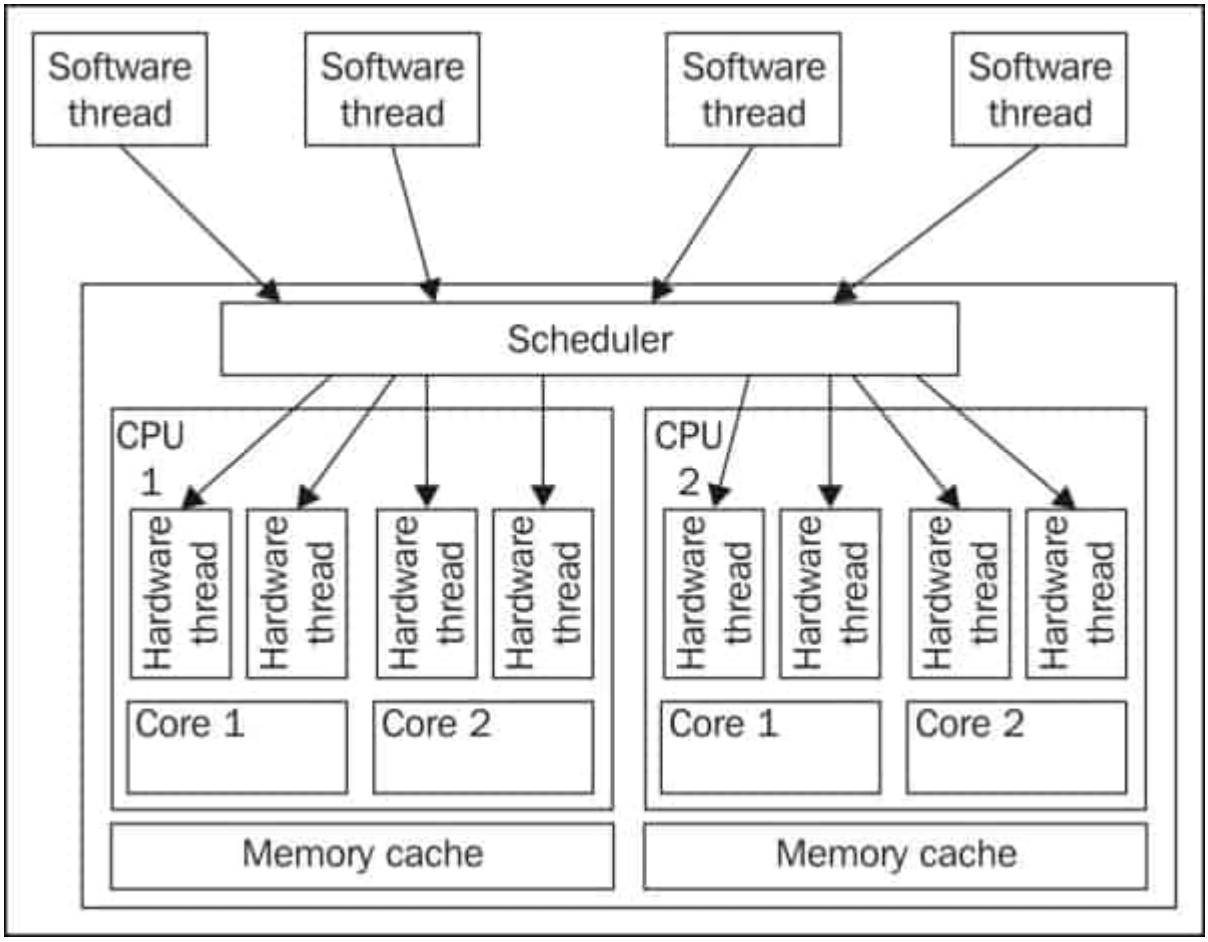
Hyper-Threading

- Processors with Hyper-Threading Technology (HT Technology) are seen by the operating system as two logical processors.
- These processors are different from multi-core processors because processors with HT Technology do not contain all the components of two separate processors.
- Only specific parts of a second processor are included so that two process threads can be executed at the same time.

You can learn about How to properly check if Hyper Threading (HT) is enabled or disabled on my

Linux server

Hyperthreading is explained in the following diagram:



Performance comparison: Hyper-threading vs physical CPU cores

Here we have a sample Python script:

```
#!/usr/bin/env python3

import sys
import datetime
import multiprocessing


def busy_wait(n):
    while n > 0:
        n -= 1


if __name__ == '__main__':
    n = 10000000
    start = datetime.datetime.now()
    if sys.argv[-1].isdigit():
        processes = int(sys.argv[-1])
```

```
    else:
        print('Please specify the number of processes')
        print('Example: %s 4' % ' '.join(sys.argv))
        sys.exit(1)

    with multiprocessing.Pool(processes=processes) as pool:
        # Execute the busy_wait function 8 times with parameter n
        pool.map(busy_wait, [n for _ in range(8)])

    end = datetime.datetime.now()
    print('The multithreaded loops took: %s' % (end - start))
```

The pool code makes starting a pool of workers and processing a queue a bit simpler as well. In this case we used map but there are several other options such as `imap`, `map_async`, `imap_unordered`, `apply`, `apply_async`, `starmap`, and `starmap_async`. Since these are very similar to how the similarly named itertools methods work, there won't be specific examples for all of them.

We will execute this script with different amount of processors:

```
server:/tmp # python3 check_ht_performance.py 1
The multithreaded loops took: 0:00:03.519231

server:/tmp # python3 check_ht_performance.py 4
The multithreaded loops took: 0:00:00.917242

server:/tmp # python3 check_ht_performance.py 16
The multithreaded loops took: 0:00:00.528790
```

As soon as the single processes actually use 100 percent of a CPU core, the task switching between the processes actually reduces performance. With single core the script took around **3.5** seconds which with 16 physical cores it took **0.5** seconds. So obviously we see better results with multi-core processes.

But we have 32 logical processors using Hyper-Threading so let us try to utilize them as well and observe the results:

```
server:/tmp # python3 check_ht_performance.py 28
The multithreaded loops took: 0:00:00.548657

server:/tmp # python3 check_ht_performance.py 32
The multithreaded loops took: 0:00:00.646604
```

Obviously you weren't expecting these results. With logical CPU processor, **the run time of the task is much higher compared to physical cores**. Since there are only 16 physical cores, the other 16 have to fight to get something done on the processor cores. This fight takes time which is why the 16 process version is slightly faster than the 32 process version.

Summary

In this article I have tried to cover some of the most asked question related to CPU, CPU Cache, multi-processing and hyper-threading. Another question that is asked far too often is whether to enable or disable hyper-threading. From hundreds of reports and testing, it appears that it is better to leave it on for newer Intel servers. The fear of performance degradation due to hyper-threading is no longer valid, as it has gone through decades of development and all the initial issues have been resolved. It is also best to not count all hyper-threading cores as real cores, since they are still virtual. When counting the number of total cores available in a node, take a conservative approach and count slightly fewer than the total cores.