RESEARCH ARTICLE OPEN ACCESS

Threading and Multiprocessing Module and the Limitations Due to the GIL in Python

Sammed Muttin[1], Deepika Dash[2]

[1]Student, R.V College of Engineering Bangalore,

[2]Assistant Professsor, Computer Science and Engineering, R.V College of Engineering Bangalore

--***************************--

ABSTRACT: With the advent of multiprocessing in Python, almost every application uses the multithreading or the multiprocessing module to parallelize the code as much as possible. This however comes with the intermittent requirement of being able to dynamically terminate a subtask as and when required by the program. Not only should the termination of the subtask is the concern, there is an added requirement that all the resources allocated to it by the CPU like memory and processors needs to be effectively de-allocated failing which, other processes will have less resources to utilize and the resources go wasted. This paper talks about how Multithreading module in python doesn't offer an effective way to terminate threads and what are the alternatives to achieve this.

KEYWORDS: Multithreading, Multiprocessing, Daemons, Zombies, GIL

--***************************--

1. INTRODUCTION

Most applications these days make use of some kind of parallel computing to make effective use of the computer resources. This parallel computing can happen either in the form of multithreading or as multiprocessing. When we take this with regards to Python programming language, the modules used for Multithreading and Multiprocessing are threading and multiprocessing respectively. While using these modules, it becomes important to understand how to terminate a process or thread dynamically. While the multiprocessing module in Python gives a clear method 'terminate' to kill a process, the same is not the case for the threading module. Thus there is a need to look at various alternatives to kill the thread and to understand what are the intricacies that lie when we are trying to achieve the same.

2. Multiprocessing Module in python

Multiprocessing is the package in python used to spawn several sub processes using an API. This API is similar to the threading package. This package offers local and remote concurrency. The important difference here is that this bypasses the Global Interpreter Lock (GIL) since it is using processes instead of threads. This results in making total use of all the cores of the CPU which is not possible in the case of threading. The multiprocessing module additionally introduces APIs which do now no longer have analogs within the threading module. A high instance of that is the Pool item which gives a handy method of parallelizing the execution of a feature throughout more than one input values, dispensing the input data throughout processes (data parallelism).

The process termination is also really simple in the multiprocessing module. There are three methods to achieve this in the multiprocessing package.

TerminateProcess(): This method is used to terminate a process and takes one argument. The process termination doesn't imply that their subprocesses are killed too. They are merely orphaned.

Kill(): This method is the same as terminate and was introduced in the version 3.7.

Close(): This method kills the process and releases all the resources associated with the process. If close returns successfully, any attributes associated with the process returns value error.

If a process has finished but is not joined, then the process becomes a Zombie. The execution of such a process is completed; however the process itself is still at the process table. This can be killed by either killing the parent process or by using the is_alive method to check the status of the process. This also joins the process. However explicitly joining the processes in the beginning is the recommended way to go.

3. Threading Module in Python

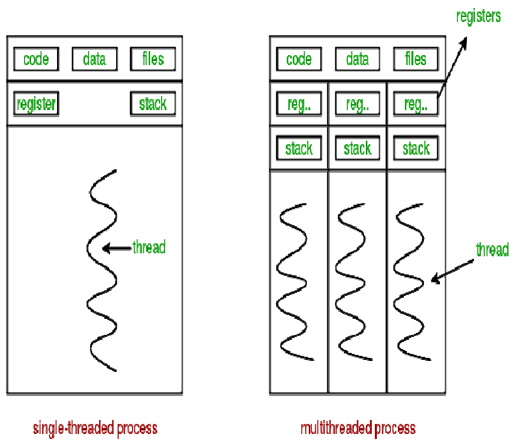

Fig-1:Architecture of a multithreaded process

Multithreading has its own benefits as compared to multiprocessing. The fact that multiple threads within a process share the same data within the main thread which makes inter thread communication much easier as compared to inter process communication. The memory overhead in processes is more as compared to threads. Threads are thus rightly called light-weight processes. Also the user has more control over threads as compared to processes in the fact that threads can be pre-empted or interrupted. A thread can also be put temporarily on hold. This process is called as yielding. However, there is no method in the threading module analogous to the terminate and kill method in the multiprocessing class that can dynamically close the execution of thread. Thus, we need to look at alternatives to kill the thread execution.

4. Alternate Methods to terminate threads in Multithreading

1) Using Exception: Threads can be terminated in python by using exceptions during thread running. The method raise exception is called in Python. This works because, When we raise an exception, we bypass target function run and it jumps out of try block. To clean up the thread, the join method is called to deallocate the resources.

2) Traces: Traces is another popular way to end a thread in python. The way traces work is very similar to that of exceptions. These traces are initially installed in each thread. They wait for their particular trigger to occur before closing the running of that thread. This stimulus can be set by the user to dynamically kill a thread.

3) By setting the threads as Daemons: Daemon threads are the threads which usually are running in the background. These kind of threads are mainly used for less significant tasks that need to take place when a process is going on. For instance, Daemon threads are widely used for clean up tasks and as a helper thread. While the main threads keep running even after the termination of the master script, This is not the case for Daemon threads. Daemon threads end their execution with the main scripts. Thus we can make use of Daemon threads to dynamically stop a thread from execution. This can be done by ending the main script.

4) Using Flags : The other unconventional way to kill a thread is by making use of global variables. We can use these global variables to act as a stimulus to kill a thread. However, This method is very unclean about the code. Also there is a drawback in this method in the fact that the global variables are insecure and can be manipulated by other threads. Python does not offer any concurrency control on these flags and hence the use of this method is rather restricted.

5) _stop() method: This is a method in python to end threads dynamically. However this method is only available in the older versions and was taken down in most new ones.

It must be remembered that all these means of thread termination are just unconventional alternatives and are not robust enough to work in any conditions. This is the drawback of threading module as Python does not provide any direct inbuilt method for thread termination.

5. Global Interpreter Lock in python

The Global Interpreter lock is essentially a mutex lock that enables Python to run one and only one thread at any given instance. Memory Management in Python is done by reference counting. This basically means any object created in python stores a variable with a count of total number of instances it has been referred at. This is used to release the memory as soon as this count reaches the value of 0. It is this variable that needed to make use of the GIL since there is a possibility of the occurrence of a race condition when two different threads are trying to access the values of the same variables. This can cause errors such as leaked memory where a memory is not released when it has to be, or could release a memory while the reference to the object still exists. Thus the GIL adds a lock on interpreter itself that ensures that there is no leak of memory in the multithreading environment.

However, As much as GIL came as a solution to make multithreading possible in python, it came in with more issues and bottlenecks . When we look at the I/O bound operations which are barely intensive on the CPU, the GIL acted as a boon. Since multiple threads running in a time sliced fashion was something that was being looked at in the scenario. Thus multithreading was able to keep the core busy at all times of the program running.

But as soon as we look into the CPU intensive tasks that can be solved using parallel computing, we get to notice the bottlenecks that GIL brings with itself. In a multithreaded environment that is CPU intensive and need to use parallel processing to fasten the process execution, there was no improvement in the time taken for the execution of the program when compared to the time it took to do the serial execution of the program. This is because the GIL restricts the CPU bound threads from running parallel to each other. The same is not true for I/O bound threads since the thread locks are exchanged between the threads when they are waiting for their turns in the much slower I/O process. The execution time actually increases in CPU bound threads scenario. This result is also nothing far from what we would be expecting since we also need to consider the overhead that we are cost during acquisition and release of the locks acquired by the threads. Thus while the GIL stops the problem of race condition for the interpreter in Python, it also limits the use of multiple cores in today's CPUs. This essentially makes threading in python a context switching and a time-slicing affair rather than a parallel computation one as it is usually mistaken to be.

.

6. CONCLUSION

The choice of using Multithreading or Multiprocessing is a rather debatable one. While the communication between the threads is easier in threading module, it is at the cost of slowing down of CPU intensive processes. The multiprocessing module on the other hand makes use of multiple cores and is equally fast in both CPU intensive and I/O intensive processes. Also there is the added ease of termination of sub-process in multiprocessing in module, while there are many intricacies involved for the same in threading module. Thus the choice comes down to the kind of tasks that need to be executed and the amount of communication required between them. This paper tries to clarify the same differences in the Multiprocessing and Threading module.

REFERENCES.

[1] Konrad Hinsen. Parallel Scripting with Python. Computing in Science & Engineering, 9(6):82-89, November 2007.

[2] Glossary. http://docs.python.org/2/glossary.html# term-global-interpreter-lock

[3] Stefano Masini, Paolo Bientinesi. High-Performance Parallel Computations Using Python as High-Level Language. Lecture Notes in Computer Science Volume, 6586:541-548, 2011

[4] Eli Bendersky. Parallelizing CPU-bound tasks with multiprocessing. http://eli.thegreenplace.net/2012/01/16/ python-parallelizing-cpu-bound-tasks-with-multiprocessing/

[5] David Beazley. Understanding the Python GIL. http://www. dabeaz.com/python/UnderstandingGIL.pdf

[6] David Beazley. Inside the Python GIL. http://www.dabeaz. com/python/GIL.pdf

[7] J. Rudd et al. A multi-core parallelization strategy for statistical significance testing in learning classifier systems. Evolutionary Intelligence, October 2013