2019 HAWAII UNIVERSITY INTERNATIONAL CONFERENCES
SCIENCE, TECHNOLOGY & ENGINEERING, ARTS, MATHEMATICS & EDUCATION JUNE 5 - 7, 2019
HAWAII PRINCE HOTEL WAIKIKI, HONOLULU, HAWAII

PYTHON: THREADS OR PROCESSES

EGGEN, ROGER
SCHOOL OF COMPUTING – INFORMATION SCIENCE
UNIVERSITY OF NORTH FLORIDA
JACKSONVILLE, FLORIDA

EGGEN, MAURICE
COMPUTER SCIENCE DEPARTMENT
TRINITY UNIVERSITY
SAN ANTONIO, TEXAS

Dr. Roger Eggen
School of Computing - Information Science
University of North Florida
Jacksonville, Florida

Dr. Maurice Eggen
Computer Science Department
Trinity University
San Antonio, Texas

Python: Threads or Processes

Synopsis:

With the advent of multi-core processors, parallel processing has become common; even economy lap top computers and cell phones are quad core. Professional programmers need to be aware of the underlying functionality of the operating system and language support in order to maximize program execution efficiency. This paper examines the impact of the global lock (GIL) on the interpreter for python and threads compared to processes.

Python: Threads or Processes

Abstract - *Operating system (OS) concepts are often perceived by students as being theoretical or obscure. While the Python programming language is not typically used for operating system implementation, Python's syntax and ease of programming can be used to demonstrate often confusing OS concepts. This paper uses Python to demonstrate concepts associated with threads and processes allowing students to see obvious differences in their functionality.*

Keywords: Python, Threads, Processes, Language Support, Operating Systems

1 Introduction

Python is an interpreted scripting language used for a wide variety of web and computer applications. Python is used for general purpose programming to develop both desktop and web applications. Python is also used for developing complex scientific and numeric applications. Python is rich with features allowing access to operating system resources such as threads, process creation, and process shared memory access through global queues. Threads and Processes are discussed in a traditional OS class and students learn that threads are light weight, meaning that the computer can create and destroy threads faster than processes. This paper shows students just how much faster threads can be created than processes. However, in many scripting languages such as Python, Ruby, Javascript, and others, the interpreter executing the source code is not thread safe. These languages employ a global interpreter lock (GIL) which allows only one thread to execute at any given moment, even on a multicore machine. This paper shows that threads can be created and destroyed significantly faster than processes, but given sufficient data, a program utilizing processes can execute significantly faster than one using threads alone as a result of the GIL.

In these examples, ubiquitous programs were created using both threads and processes. Quicksort and Matrix Multiply representing O(nlgn), and O(n^3) algorithms were executed and timed to determine relative efficiency using both threads and processes allowing insight to the impact of the GIL. Also, a program was created that times the spawning of many threads and, similarly, the forking of as many processes. The resulting timings clearly show the relative efficiency of creating numerous threads as well as numerous processes. Students will see, first hand, the efficiency comparison. Sample Python code is included for demonstration purposes.

2 Hardware

The execution environment consisted of genuine Intel Xeon v4 2.10 GHz with 16 cores that are hyperthreaded allowing the operating system to see 32 cores. The system is configured with 32 gigabytes of RAM. The machine is a symmetric multiprocessor allowing each core access to main memory. The machine has 32K L1d and 32K L1i cache on each core with additional 256K L2 and 2,0480K L3 cache.

3 Software

The operating system is CENTOS version 7 (Core) which has Redhat Linux support. The research was done using Python version 2.7.5. While this isn't the most recent version, many applications are being developed using Python version 2. The research will be duplicated using version 3 to see if significant performance changes occur. Python 3 also uses the GIL so major changes in execution timings are not expected. There are varying techniques and optimizations that can be employed to speed up python. These algorithms were carefully implemented in a consistent manner for both threads and processes.

4 The Research

This research tests two major concepts: 1) how quickly threads and processes can be created and destroyed and 2) the efficiency of Python threads compared to processes. Threads are considered light weight; they do not demand many resources to start, maintain, or remove. Processes are more resource intensive requiring more time to create, support, and destroy. With minimal data and small computation times, threads should outperform processes, but with sufficient data and computation demands, processes (which execute in parallel) should require less total computing time. To evaluate the hypothesis, the relative performance of threads versus processes tests of 100, 1000, 10000, 100000, and 1000000 integers were randomly generated. The integers were sorted using quick sort which executes in O(nlgn) time. Each of these data sizes were run with 1, 2, 4, 8, 16, and 32 processes and then again with 1, 2, 4, 8, 16, and 32 threads. Data were portioned relatively equally for each thread or process so that maximum efficiency was achieved. Square matrices were multiplied using an $O(n^3)$ algorithm. The sizes chosen were 16x16 (somewhat more than 100 total numbers), 32x32 (nearly 1,000 total numbers), 96x96 (nearly total 10,000 numbers) 320x320 (100,000 total numbers) and 1024x1024 numbers (nearly 1,000,000 total numbers). The number of numbers was relatively close to the number of numbers used in the sorting routine. Theses sizes were chosen so that each thread or process had exactly the same amount of work to do in producing the product matrix. Each thread or process sorted n numbers divided by the number of threads or processes which insured equal amounts of work accomplished by each thread or process. Similarly, each thread or process computed exactly n number of rows in the resulting matrix. For example, if the matrix was 96 rows by 96 columns and 8 threads were used, each thread would multiply 96/8 = 12 rows of the product matrix. Similarly, a 96x96 matrix computed with 8 processes would cause each process to compute 12 rows of the resulting matrix. All matrix sizes were chosen so that each thread or process had exactly the same amount of work to do.

5 Thread or Process Creation

Table 1 provides expected results. Processes take longer to create than light weight threads. The time difference is about 40 times. The primary code for spawning threads is:

```
0 # create threadnum of threads
1 starttime = time.time()
2 for tName in range(threadnum):
3   th=Thread(target=quickSort,args=(num,))
4   th.start()
5   th.join()
6 tottime = time.time() - starttime
```

Line 1 gets the initial time, line 2 iterates for the desired number of threads. Line 3 actually creates the threads and line 4 starts the thread execution. Line 5 waits for the thread to finish and line 6 records the elapsed time. The primary code for forking processes is:

```
1 starttime = time.time()
2 for pName in range(procnum):
3   pro = Process(target=meh, args=(nums,))
4   pro.start()
5   pro.join()
6 tottime = time.time() - starttime
```

The logic of the two code segments is essentially identical. Students will have no trouble understanding the code and realizing the significant difference in the requirements of resources to create a significant number of threads and processes. Complete programs can be found at www.unf.edu/~ree.

Clearly, process creation and destruction is a more expensive task than thread creation and destruction. However, this is not the end of the issue. As stated above, the interpreter for many languages is not thread safe which requires a global interpreter lock (GIL). The GIL requires all threads to run sequentially in a time sliced fashion, even on multicore / multiprocessor machines. This is all reinforced by considering the times necessary to execute well known quicksort and matrix multiplication algorithms.

Since no parallelism is possible with 1 process or 1 thread, threads execute faster than processes. For O(nlgn) algorithms where processing time is very small, the time to create a process versus a thread is shown, particularly when small data sizes are used (100 numbers). A process takes nearly 4 times longer to execute as shown by 0.0031 for processes and 0.0008 for threads with 100 numbers and an O(nlgn) algorithm. The 40 times longer from table 1 is mitigated by the processing time to sort the numbers. This reflects time to create a process versus time to create a thread and then process the data. However, the times are more nearly equal with threads being approximately 10% faster when larger data sizes are used with more computation (1,000,000 number and O(n^3) algorithm).

Table 1 Times to Create and Destroy threads and processes (times in seconds)

Data Size		1,000		10,000
Threads		0.1288		0.9339
Processes		4.2549		41.1683

Table 2 shows the results for two processes and two threads. The table shows processes time is longer than the time to create two threads for 100 numbers (nearly 3 times as long). When processing time becomes an issue, even with 100 numbers, considering the O(n^3) algorithm, the parallelism provides a benefit allowing the python processes to complete the work in half the time compared to concurrent threads since more computation is required in O(n^3) algorithm. The parallelism for processes versus concurrency for threads allows the process version to execute faster. When studying column 1 in Table 2, the time to create two O(n^3) processes with larger data sets of 100,000 and 1,000,000 integers, the gains realized by parallelism via processes are much more pronounced.

Table 2 Times for 1 Process 1 Thread (times in seconds)

Data Size		100		1,000		10,000		100,000		1,000,000
Process O(nlgn)		0.0031		0.0045		0.0243		0.2396		2.144215
Thread O(nlgn)		0.0010		0.0053		0.0564		0.4886		20.9265
Process O(n^3)		0.0219		0.1074		1.8356		76.3424		4344.6862
Thread O(n^3)		0.0441		0.2564		4.9144		213.5061		8818.2759

Table 3 Times for 2 Processes 2 Threads (times in seconds)

Data Size		100		1,000		10,000		100,000		1,000,000
Process O(nlgn)		0.0031		0.0063		0.0445		0.6122		25.2113
Thread O(nlgn)		0.0008		0.0043		0.0451		0.5929		26.0556
Process O(n^3)		0.0348		0.1831		3.3402		150.7323		6947.8297
Thread O(n^3)		0.0317		0.1747		3.3341		152.7535		6541.0419

Tables 3, 4, 5, and 6 provide similar but even more striking differences between the execution times of processes and threads. Again, the time to create processes with very small data overcomes the realized parallel speed up of processes. However, when larger data sizes are used, the benefit of parallelism over concurrent threads shows a significant benefit. For example, with 1,000,000 integers using an O(n^3) algorithm, an 18 times speed up is realized.

Table 4 Times for 4 Processes 4 Threads (times in seconds)

Data Size		100		1,000		10,000		100,000		1,000,000
Process O(nlgn)		0.0044		0.0049		0.0142		0.1129		0.8261
Thread O(nlgn)		0.0014		0.0076		0.0660		0.5846		17.0908
Process O(n^3)		0.0148		0.0659		1.0154		41.4070		2356.5256
Thread O(n^3)		0.0537		0.3202		6.7891		264.6105		9823.1040

Table 5 Times for 8 Processes 8 Threads (times in seconds)

Data Size		100		1,000		10,000		100,000		1,000,000
Process O(nlgn)		0.0074		0.0070		0.0113		0.0628		0.3454
Thread O(nlgn)		0.0022		0.0085		0.0690		0.5995-		11.6326
Process O(n^3)		0.0132		0.0410		0.5959		21.4535		1279.1322
Thread O(n^3)		0.0591		0.3290		7.5634		294.8082		10098.1179

Table 6 Times for 16 Processes 16 Threads (times in seconds)

Data Size		100		1,000		10,000		100,000		1,000,000
Process O(nlgn)		0.0122		0.0113		0.0134		0.0402		0.2766
Thread O(nlgn)		0.0032		0.0093		0.0714		0.6133		8.9336
Process O(n^3)		0.0178		0.0382		0.4019		13.8212		776.3787
Thread O(n^3)		0.0661		0.3250		7.4593		304.7265		10706.2346

Table 7 shows no testing for 100 numbers with 32 processes or threads since half of the processes/threads would have no data.

.

Table 7 Times for 32 Processes 32 Threads (times in seconds)

Data Size	100	1,000	10,000	100,000	1,000,000
Process O(nlgn)	0.0200	0.0208	0.0230	0.0403	0.2765
Thread O(nlgn)	0.0048	0.0107	0.0750	0.5387	7.4602
Process O(n^3)	no data	0.0383	0.3536	13.0681	605.1103
Thread O(n^3)	no data	0.3339	7.6025	311.0278	11023.3230

6 Conclusion

It is clear from Table 1 that processes require more resources to create and destroy than threads. The programs clearly demonstrate the results. Operating System students will be able to see firsthand the differences in time required. Parallelism versus timed sliced concurrency is demonstrated by the remaining tables which show that given sufficient processing requirements, parallelism by using processes will yield better computation times.

Generally speaking, threads are used for small tasks and processes are used for more heavyweight tasks. The advantage of threads is that they are generally easier to program, whereas processes provide slightly more of a challenge in that data is not as easily shared among processes. As stated earlier, good quality software that takes advantage of modern multi-core hardware is required for many applications.

This research shows that programmers must consider processes when using languages such as Python and Ruby and others that utilize the GIL. Significant performance increases are realized as the result of using processes, each of which has their own GIL. Parallelism is achieved via processes as opposed to concurrency from using threads. Considering the fact that Python is used for general purpose computing [7], efficiency is always a consideration. The data clearly show that processes execute faster with significant data than threads. However, this does not mean threads should never be used. When an application does considerable I/O, threads being time sliced can provide significant speed up since one thread can execute while another thread waits on the I/O. The programmer should consider the nature of the application, and if there is much processing to be done, processes should be used.

7 Future Work

Languages that use a GIL should yield similar times; however, each language executes at differing rates. Consideration should be given to Python 3, Cython, Jython, and Ruby. These languages have similar characteristics, but likely will yield differing run times. Jython generates Java byte code so it is expected thread execution would be faster than processes as a result of the GIL not being involved.

8 References

[1]https://www.codementor.io/satwikkansal/python-practices-for-efficient-code-performance-memory-and-usability-aze6oiq65

[2]https://www.codementor.io/satwikkansal/python-practices-for-efficient-code-performance-memory-and-usability-aze6oiq65

[3]http://www.monitis.com/blog/python-performance-tips-part-1/

[4]https://stackoverflow.com/questions/20639180/explanation-of-how-nested-list-comprehension-works

[5]https://medium.com/@mindfiresolutions.usa/python-7-important-reasons-why-you-should-use-python-5801a98a0d0b

[6]https://www.google.com/search?client=firefox-b-1&q=Uses+of+python

[7]https://docs.python.org/2/library/multiprocessing.html

[8]Beazley, David, *Python Cookbook*, Third Edition O'Reilly, 2013

[9]Bader, Dan, *Python Tricks: A Buffet of Awesome Python Features*, Dan Bader Publisher, 2017

[10] https://www.slideshare.net/dabeaz/an-introduction-to-python-concurrency