Global Interpreter Lock

This lesson discusses the global interpreter lock also known as GIL and its effects.

Global Interpreter Lock

In one of the previous sections we discussed I/O and CPU bound programs. In this lesson, we'll understand why only I/O bound Python programs can leverage multithreading to speed up execution time. But first, a quick refresher of how Python works is necessary to understand the acronym GIL or global interpreter lock.

How Python works

Python is an interpreted language, that is, there is no static time compiling as that happens in the case of Java, C or C++. The program that interprets user code is called the *Interpreter*. An interpreter is a program that executes other programs. At a higher level when we run a Python program (.py file), the Python interpreter compiles the source code into byte code. The generated byte code is a lower-level platform-independent representation that can be understood by the Python Virtual Machine (PVM). In the next step, the byte code is routed to the PVM for execution. Note that PVM isn't a separate component. Rather, it is just a loop in the Python interpreter that is responsible for executing byte code line by line. The PVM is really a part of the interpreter.

Python Interpreter

The Python interpreter as explained is responsible for executing a program, but it can only execute a single thread at a time. This is the falling of the reference implementation of Python - CPython, called so because it is written in the C language. So if your machine has one, ten, or a hundred processors, the Python interpreter is only able to run a single thread at a time using a single processor. Two threads on a machine with two available processors can't be executed in parallel each running on a single CPU.

This design has direct consequences on the performance of CPU bound programs since they don't experience any speed-up in the presence of additional processors. In fact, they may run slower because of the additional housekeeping overhead required for running multiple threads.

Why is that so?

One may wonder what was the design decision behind restricting the interpreter to run a single thread. The answer lies in how memory management works in Python - reference counter.


```
1  import sys
2
3  # declare a variable
4  some_var = "Educative"
5
6  # check reference count
7  print sys.getrefcount(some_var)
8
9  # create another refrence to someVar
10 another_var = some_var
11
12 # verify the incremented reference count
13 print sys.getrefcount(some_var)
```

Run Save Reset

Output 0.59s

```
4
5
```

If you run the above snippet, you'll see the reference count of the variable `some_var` increase. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero, the object is deallocated. The interpreter executes a single thread in order to ensure that the reference count for objects is safe from race conditions.

A reference count is associated with each object in a program. One possible solution could have been to associate one lock per object so that multiple threads could work on the object in a thread-safe manner. However, this approach would have resulted in too many locks being managed with the possibility of deadlocks. Thus, a compromise was made to have a single lock that provides exclusive access to the Python interpreter. This lock is known as the **Global Interpreter Lock**.

Execution of Python bytecode requires acquiring the GIL. This approach prevents deadlocks as there's a single global lock to manage and introduces little overhead. However, the cost is paid by making CPU-bound tasks essentially single-threaded.