A discussion of Python’s removal of global interpreter locks

Chuanhui Wu
School of Information Engineering, Huzhou University, Huzhou 313000,China

Abstract: Almost all versions of Python use the Global Interpreter lock (GIL) for thread safety. Although GIL removal has been proposed several times, it has never been implemented. However, Python has now adopted PEP 703 for Python projects, which proposes to gradually remove GIL locks over the next few years to improve the multithreading performance of Python code. This is a bold effort that will bring unprecedented changes to Python. Therefore, it is necessary to learn more about GIL, how to remove GIL, analyze the pros and cons of removing GIL, and analyze the various effects of removing GIL on Python.

Key words: Python; GIL; Thread

1. Overview

GIL has been around since Python1 and is a foundation for keeping Python running properly. Recently, Python announced that in the new version of CPython, GIL lock will be "optional" and will be completely removed in the future. The shutdown of the GIL means that multiple threads that previously could only be serial can now run in parallel. For computationally intensive fields such as artificial intelligence and machine learning, Python work efficiency will be greatly improved. At the same time, it will also bring unpredictable changes to the current Python usage environment. This is an event worth exploring.

2. Introduce GIL

Python’s memory management mechanism is to monitor the usage of objects by tracking their reference count, and once an object’s reference count drops to zero, it is cleared. Here is a simple demonstration example:

```
# Create an object and have multiple variables reference it
a = [1, 2, 3]
b = a
c = a
# View the object’s reference count
import sys
print(sys.getrefcount(a)) # Output: 4 (arguments for a, b, c, getrefcount)
# dereference variable
b = None
c = None
# View the object’s reference count
print(sys.getrefcount(a)) # Output: 2
# Object is garbage collected
a = None
```

In the above example, a list object a is created and then assigned to the variables b and c, causing the object’s reference count to increase to 4, which is referenced by the variables a, b, c, and the argument passed to sys.getrefcount(). When the variables b and c are dereferenced, the object’s reference count is reduced to 2. Finally, when the variable a is also dereferenced, the reference count of the object is reduced to zero, indicating that the object can be garbage collected to free memory.

However, when multiple threads are included in a Python process, the program has no control over their relative order of execution, so Race conditions can occur, leading to data inconsistencies, logic errors, and even program crashes. Therefore, this reference counting mechanism is not a complete thread-safe guarantee. To circumvent the competitive risk, Python introduces the Global Interpreter Lock (GIL).

GIL is a mutex lock that ensures that only one thread is executing Python bytecode in the interpreter at any given time. Here are examples of pseudocode that looks like a GIL lock:

```
a = 1
lock.acquire()
if a > 0:
a -= 1
lock.release()
```

In pseudocode, the program locks the code before running the if statement, and no other thread can run the program again until the release. Since all Python related code has the potential for multiple threads trying to read or write at the same time , GIL is designed as a global lock to ensure that only one thread can execute Python bytecode at any given time.

In the early days of Python’s development, GIL was a great convenience. First, GIL simplifies memory management because there is no need to add additional locks to each object to protect their access. Second, the GIL ensures the interpreter’s thread-safety, reducing many potential concurrency issues.

In addition, in order to meet the needs of Python's ease of use and speed of development, developers have created many extensions using existing C libraries. However, in order to avoid introducing inconsistent modifications, these C libraries rely on the thread-safe memory management mechanism provided by the GIL.

3. Reasons for removing the GIL

Although the GIL has played a positive role in ensuring the execution of a single thread, as multi-core processors become more common and computer systems become more complex, the GIL has become a limitation of Python.

(1) GIL is a major obstacle to Python concurrency. On CPU-intensive tasks, the problems caused by a lack of concurrency are often more important than code speed. As an example, consider an image processing application that requires feature extraction and processing of a large number of images. If this application mainly involves IO operations, such as reading pictures from a disk or making network requests, the GIL's impact may not be too obvious because while the thread waits for the IO operation to complete, the GIL releases control and allows other threads to execute. But suppose there is a task queue that contains many picture processing tasks, each of which can be executed in a separate thread. Because of the GIL, although multiple threads can be started at the same time, only one thread can execute Python bytecode at any one time. Even when multiple processor cores are available, threads compete for control of the GIL, preventing true parallel execution.

(2) GIL affects the availability of Python libraries. Although GIL only limits the implementation details of multithreading parallelism, it is difficult to intuitively reveal the usability problems of Python libraries. However, many library authors are very performance conscious and will therefore design apis that work around the GIL. This practice can lead to more complexity in the use of the API.

4. A strategy to ensure that Python runs smoothly without a GIL

(1) Use biased reference counting. Biased reference counting avoids some of the additional synchronization overhead by distinguishing between objects that are only accessed by a single thread and those that may be accessed by multiple threads. Specifically, for objects that are only accessed by a single thread, the reference count operation can be biased so that instead of needing to be synchronized every time the reference count is added or subtracted, the operation is only synchronized under certain circumstances, thereby improving performance.

For this purpose, Python gives the biased reference count structure as follows:

```
struct _object {
_PyObject_HEAD_EXTRA
uintptr_t ob_tid;
uint16_t __padding;
PyMutex ob_mutex;
uint8_t ob_gc_bits;
uint32_t ob_ref_local;
Py_ssize_t ob_ref_shared;
PyTypeObject *ob_type;
};
```

The above code through the ob_tid, ob_ref_local, and ob_ref_share fields to store the thread ID of the thread that owns the object, store the local reference count, store the shared reference count and the status bit, so as to achieve a biased reference count.

Use Immortalization. Immortalization is used to manage specific types of objects that are not considered to need to be released or redistributed during their lifetime. Examples are resident strings, small integers, statically allocated PyTypeObjects, and logical operands, which are permanent for the lifetime of the program. For this reason, they are marked as persistent by the value UINT32_MAX in the ob_ref_local field.

(2) Use deferred reference counting. Deferred reference counting works by delaying the update of the reference count until a certain time. Specifically, the reference count is not updated immediately when the object's reference count changes. Instead, the change in the count is deferred until some appropriate time (such as when the program is idle or during a specific memory management operation). This avoids frequent reference count updates, which reduces overhead.

(3) Improve the garbage collector. In CPython, the garbage collector is an important component responsible for automatically managing memory. In the non-GIL version, the garbage collector makes some improvements to improve efficiency. Examples include eliminating generational garbage collection, adopting "stop-the-world" for thread-safety that used to be guaranteed by GIL, or integrating deferred reference counting.

Delayed reference counting is used in conjunction with the garbage collector. Here is a pseudocode example that demonstrates a similar delayed reference counting:

```
class DelayedReferenceCount:
def __init__(self, value):
self.value = value
self.ref_count = 0
def increment_ref_count(self):
```

```
self.ref_count += 1
def decrement_ref_count(self):
self.ref_count -= 1
if self.ref_count == 0:
# Perform the actual decrement when the reference count reaches 0
self.value = None
# Create deferred reference count instances
obj = DelayedReferenceCount("Hello, world!" )
# Increase the citation count
obj.increment_ref_count()
obj.increment_ref_count()
# Reduce the reference count
obj.decrement_ref_count()
obj.decrement_ref_count()
# This value is set to None only if the reference count goes to 0
print(obj.value) # Output: None
```

In this example, the DelayedReferenceCount class emulates an object that has the property of deferring reference counts. The increment_ref_count method can be called every time the object is referenced to increase the reference count, and the decrement_ref_count method can be called every time the reference is canceled to decrease the reference count. Only when the reference count reaches zero will the value of the object be set to None, completing the release of the object.

5. Undo the GIL effect on Python

(1) Higher parallelism: When the GIL is not present, multiple threads can execute Python code at the same time, which will improve parallelism and allow parallel tasks to be completed faster. This is very beneficial for programs that handle CPU-intensive tasks (computationally intensive tasks), as they enable true parallel computing on multi-core processors.

(2) More complex memory management: Removing the GIL may introduce more complex memory management issues. Since multiple threads can modify and access objects at the same time, more elaborate synchronization and locking mechanisms are needed to avoid race conditions and data inconsistencies. This can add complexity to developers when writing multithreaded code.

(3) thread-safety challenges: Removing the GIL will cause all parts of the CPython standard library that originally relied on the GIL to be modified to ensure that they are thread-safe in a multithreaded environment, which can be a huge challenge.

6. Undo the effect of GIL on Python users

(1) Better multithreading performance: It can be expected that after canceling GIL, each thread will be able to utilize the system's multi-core processor more efficiently, and Python's multithreading performance will be significantly improved.

(2) More concurrency options: Users will be able to choose more freely whether to use multithreading or multiprocess to achieve concurrency. This may require users to have a deeper understanding of multi-threaded and multi-process programming in order to choose the concurrency model that best suits their application needs.

(3) Single-threaded performance damage: Although removing the GIL will increase Python's multi-threaded performance, it will also bring about a performance drop that cannot be ignored when it comes to regular single-threading. In fact, according to the official data provided by Python, the single-threaded performance of Python3.11 will decrease by about 10% after removing the GIL. Users need to weigh the pros and cons and decide whether or not to accept this performance loss.

7. Summarize

Whether for or against, Python's decision to remove GIL has been made. Therefore, all Python developers and users need to realize that the removal of GIL is an inevitable trend in the future development of Python. Behind this decision lies a long-term vision for the language's performance and concurrency capabilities to support more efficient and flexible applications. However, it is important to note that removing the GIL is not an overnight process. It is a complex evolution that may take time to gradually implement and perfect. During the transition, Python users may face some inconveniences and adjustments, especially for projects that rely on multithreaded programming. But these temporary difficulties will all pave the way for a great future for Python.

References:

[1]WANG Y, WU C J, WANG X, et al. Exploiting Parallelism Opportunities with Deep Learning Frameworks[J]. ACM Transactions on Architecture and Code Optimization,ACM Transactions on Architecture and Code Optimization, 2019.

[2]CHOI J, SHULL T, TORRELLAS J. Biased reference counting: minimizing atomic operations in garbage collection[C/OL]//Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. 2018. http://dx.doi.org/10.1145/3243176.3243195.

[3]Sam Gross, PEP 703 – Making the Global Interpreter Lock Optional in CPython[EB/OL]. https://peps.python.org/pep-0703/, 2023

[4]Nogil,Multithreaded Python without the GIL[DB/OL]. https://github.com/colesbury/nogil/tree/f7e45d6bfbbd48c8d5cf851c116b73b85add9fc6, 2022