Python's GIL implemented in pure Python

Last updated on June 14, 2018, in Python

There is an excellent presentation of how the modern GIL performs thread scheduling, but unfortunately, it lacks some interesting details (at least for me). I was trying to understand all the details of the GIL, and it took me some time to fully understand it from the CPython's source code.

So here is a simplified algorithm of the thread scheduling that is taken from CPython 3.7 and rewritten from C to pure Python for those, who are trying to understand all the details.

```python
import threading
from types import SimpleNamespace

DEFAULT_INTERVAL = 0.05

gil_mutex = threading.RLock()
gil_condition = threading.Condition(lock=gil_mutex)
switch_condition = threading.Condition()

# dictionary-like object that supports dot (attribute) syntax
gil = SimpleNamespace(
    drop_request=False,
    locked=True,
    switch_number=0,
    last_holder=None,
    eval_breaker=True
)



def drop_gil(thread_id):
    if not gil.locked:
        raise Exception("GIL is not locked")

    gil_mutex.acquire()

    gil.last_holder = thread_id
    gil.locked = False

    # Signals that the GIL is now available for acquiring to the first awaiting thread
    gil_condition.notify()

    gil_mutex.release()

    # force switching
    # Lock current thread so it will not immediately reacquire the GIL
    # this ensures that another GIL-awaiting thread have a chance to get scheduled

    if gil.drop_request:
        switch_condition.acquire()
        if gil.last_holder == thread_id:
            gil.drop_request = False
            switch_condition.wait()

        switch_condition.release()


def take_gil(thread_id):
    gil_mutex.acquire()

    while gil.locked:
        saved_switchnum = gil.switch_number

        # Release the lock and wait for a signal from a GIL holding thread,
        # set drop_request=True if the wait is timed out

        timed_out = not gil_condition.wait(timeout=DEFAULT_INTERVAL)

        if timed_out and gil.locked and gil.switch_number == saved_switchnum:
            gil.drop_request = True

    # lock for force switching
    switch_condition.acquire()

    # Now we hold the GIL
    gil.locked = True

    if gil.last_holder != thread_id:
        gil.last_holder = thread_id
        gil.switch_number += 1

    # force switching, send signal to drop_gil
    switch_condition.notify()
    switch_condition.release()

    if gil.drop_request:
        gil.drop_request = False

    gil_mutex.release()


def execution_loop(target_function, thread_id):
    # Compile Python function down to bytecode and execute it in the while loop

    bytecode = compile(target_function)

    while True:

        # drop_request indicates that one or more threads are awaiting for the GIL
        if gil.drop_request:
            # release the gil from the current thread
            drop_gil(thread_id)

            # immediately request the GIL for the current thread
            # at this point the thread will be waiting for GIL and suspended until the fu
            take_gil(thread_id)

        # bytecode execution logic, executes one instruction at a time
        instruction = bytecode.next_instruction()
        if instruction is not None:
            execute_opcode(instruction)
        else:
            return
```

Note that this code will not run if you will try to execute it, because it's missing bytecode execution logic.

Some things to note

- Each thread executes its code in the separate `execution_loop` which is run by the real OS threads.
- When Python creates a thread it calls the `take_gil` function before entering the `execution_loop`.
- Basically, the job of the GIL is to pause the while loop for all threads except for a thread that currently owns the GIL. For example, if you have three threads, two of them will be suspended. Typically but not necessarily, only one Python thread can execute Python opcodes at a time, and the rest will be waiting a split second of time until the GIL will be switched to them.
- The C implementation can be found here and here.

A comment from the source code describes the algorithm as follows:

```
/*
   Notes about the implementation:

   - The GIL is just a boolean variable (locked) whose access is protected
     by a mutex (gil_mutex), and whose changes are signalled by a condition
     variable (gil_cond). gil_mutex is taken for short periods of time,
     and therefore mostly uncontended.

   - In the GIL-holding thread, the main loop (PyEval_EvalFrameEx) must be
     able to release the GIL on demand by another thread. A volatile boolean
     variable (gil_drop_request) is used for that purpose, which is checked
     at every turn of the eval loop. That variable is set after a wait of
     `interval` microseconds on `gil_cond` has timed out.

      [Actually, another volatile boolean variable (eval_breaker) is used
       which ORs several conditions into one. Volatile booleans are
       sufficient as inter-thread signalling means since Python is run
       on cache-coherent architectures only.]

   - A thread wanting to take the GIL will first let pass a given amount of
     time (`interval` microseconds) before setting gil_drop_request. This
     encourages a defined switching period, but doesn't enforce it since
     opcodes can take an arbitrary time to execute.

     The `interval` value is available for the user to read and modify
     using the Python API `sys.{get,set}switchinterval()`.

   - When a thread releases the GIL and gil_drop_request is set, that thread
     ensures that another GIL-awaiting thread gets scheduled.
     It does so by waiting on a condition variable (switch_cond) until
     the value of last_holder is changed to something else than its
     own thread state pointer, indicating that another thread was able to
     take the GIL.

     This is meant to prohibit the latency-adverse behaviour on multi-core
     machines where one thread would speculatively release the GIL, but still
     run and end up being the first to re-acquire it, making the "timeslices"
     much longer than expected.
     (Note: this mechanism is enabled with FORCE_SWITCHING above)
*/
```

> Popular posts in Python category

Python, advanced python, cpython internals