

Select lesson text to explore the [Explain with AI](#) Premium Offering



## ... continued

Continuation of the discussion on GIL.

### Removing GIL

One may wonder why the GIL can't be removed from Python because of the limitations it imposes on CPU-bound programs. Attempts at removing GIL resulted in breaking C extensions and degrading the performance of single and multithreaded I/O bound programs. Therefore, so far GIL hasn't been removed from Python.

Python's GIL is intended to serialize access to interpreter internals from different threads. In summary, threads in Python are only good for blocking I/O. While N threads are blocked on network or disk I/O or just waiting to reacquire the GIL, one thread runs in the Python interpreter.

In Python 3.7 the GIL is a boolean variable that is guarded by a mutex. The GIL implementation for Python 3.7 lives in the [ceval\\_gil.h](#) on Github. We reproduce the comments at the start of the file which explain the inner workings of the GIL. Even if you don't completely understand the snippet, it should impart a high-level view of how the GIL works.

```
/*
```

```
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

Select lesson text to explore the [Explain with AI](#) Premium Offering

- 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)

\*/

## Python Implementations without GIL

There are also Python implementations which circumvent the GIL altogether. Examples include [Jython](#), [IronPython](#) and [pypy-stm](#).

[← Back](#)

☒ [Mark As Completed](#)

[Next →](#)

Global Interpreter Lock

Amdahl's Law

