

gill

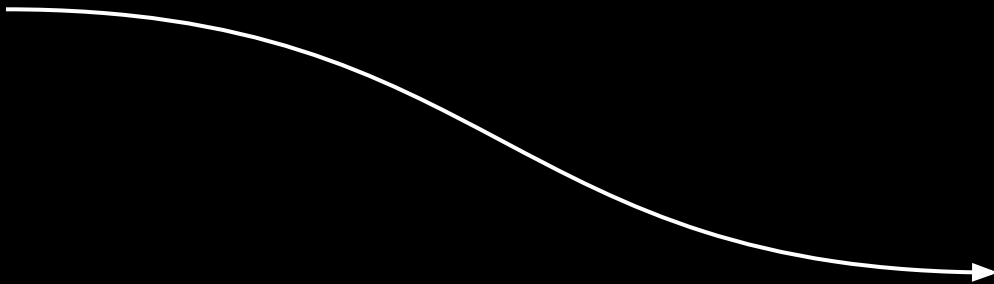
Google: can Python into threads?

„CPython doesn't support multi-threading”

1992

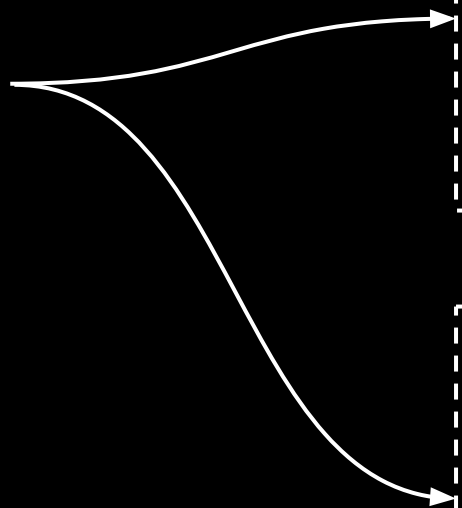
1992

- me



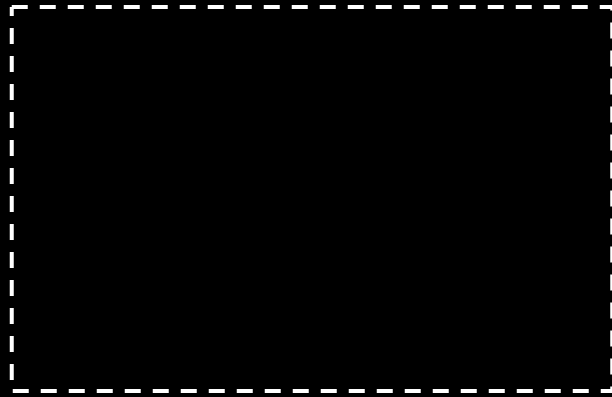
# 1992

- me
- Java & JavaScript




# 1992

- me
- Java & JavaScript
- consumer-grade multi-core CPUs



# 1992

- me
- Java & JavaScript
- consumer-grade multi-core CPUs
- Python (with threading support)



python



× "CPython doesn't support multi-threading"

× "CPython doesn't support multi-threading"  
"CPython can run only on single core"

- × "CPython doesn't support multi-threading"
- × "CPython can run only on single core"

- × "CPython doesn't support multi-threading"
  - × "CPython can run only on single core"
- "CPython process can execute Python bytecode in one thread at the time"

- ✗ "CPython doesn't support multi-threading"
- ✗ "CPython can run only on single core"
- ✓ "CPython process can execute Python bytecode in one thread at the time"



kolodziejj.info  
@unit03



GIL

# GIL: prerequisites



# Parallelism

# Parallelism

- requires multiple cores

# Parallelism

- requires multiple cores

Thread 0 on core 0



# Parallelism

- requires multiple cores

Thread 0 on core 0

Thread 1 on core 1



# Parallelism

- requires multiple cores

Thread 0 on core 0

Thread 1 on core 1



- as opposed to concurrency

Single core



# Threads

# Threads

- process

# Threads

- process:
  - instance of an application



# Threads

- process:
  - instance of an application
  - one or more threads

# Threads

- process:
  - instance of an application
  - one or more threads
  - shared memory

# Threads

- process:
  - instance of an application
  - one or more threads
  - shared memory
- import threading

# Threads

- `process`:
  - instance of an application
  - one or more threads
  - shared memory
- `import threading`
  - system (kernel/native/Posix) threads

# Threads

- `process`:
  - instance of an application
  - one or more threads
  - shared memory
- `import threading`
  - system (kernel/native/Posix) threads
  - as opposed to: green threads, greenlets, coroutines etc.

# Threads

- process:
  - instance of an application
  - one or more threads
  - shared memory
- import threading
  - system (kernel/native/Posix) threads
  - as opposed to: green threads, greenlets, coroutines etc.
- thread state:
  - ready/runnable
  - running
  - waiting/blocked

# Locks and system calls

- locks:



- locks:
  - tool for preventing race-conditions in shared resources

- locks:
  - tool for preventing race-conditions in shared resources
  - mutex - **mutual exclusion**

- locks:

- tool for preventing race-conditions in shared resources
- mutex - **mutual exclusion**
- lock (hold) and unlock (release)

- locks:
  - tool for preventing race-conditions in shared resources
  - mutex - **mutual exclusion**
  - lock (hold) and unlock (release)
- system calls:

- locks:
  - tool for preventing race-conditions in shared resources
  - mutex - **mutual exclusion**
  - lock (hold) and unlock (release)
- system calls:
  - operating system/kernel API

- locks:
  - tool for preventing race-conditions in shared resources
  - mutex - **mutual exclusion**
  - lock (hold) and unlock (release)
- system calls:
  - operating system/kernel API
  - services for applications: I/O access, process control...

- locks:
  - tool for preventing race-conditions in shared resources
  - mutex - **mutual exclusion**
  - lock (hold) and unlock (release)
- system calls:
  - operating system/kernel API
  - services for applications: I/O access, process control...
  - can be blocking (e.g.: I/O access, sleeping...)

# Memory management



# Memory management

- CPython:
  - reference counters

# Memory management

- CPython:
  - reference counters (shared resources - needs locking!)

# Memory management

- CPython:
  - reference counters (shared resources - needs locking!)
  - + garbage collector for circular references only

# Memory management

- CPython:
  - reference counters (shared resources - needs locking!)
  - + garbage collector for circular references only
- other option: tracing garbage collector

# Memory management

- CPython:
  - reference counters (shared resources - needs locking!)
  - + garbage collector for circular references only
- other option: tracing garbage collector
  - more robust but more complex

# Memory management

- CPython:
  - reference counters (shared resources - needs locking!)
  - + garbage collector for circular references only
- other option: tracing garbage collector
  - more robust but more complex
  - JVM or C#

# Python bytecode

# Python bytecode

- Python code -> compilation -> bytecode



# Python bytecode

- Python code -> compilation -> bytecode
- 

```
a = 1
```

```
print(a)
```

# Python bytecode

- Python code -> compilation -> bytecode

---

a = 1	2	LOAD_CONST	1	(1)
	4	STORE_FAST	0	(a)
print(a)	6	LOAD_GLOBAL	1	(NULL + print)
	18	LOAD_FAST	0	(a)
	20	PRECALL	1	
	24	CALL	1	

# Python bytecode

- Python code -> compilation -> bytecode

a = 1	2	LOAD_CONST	1	(1)
	4	STORE_FAST	0	(a)
print(a)	6	LOAD_GLOBAL	1	(NULL + print)
	18	LOAD_FAST	0	(a)
	20	PRECALL	1	
	24	CALL	1	

- bytecode -> interpreter -> execution

finally  
GILL

# GIL

- Global Interpreter Lock

# GIL

- Global Interpreter Lock
- protects against race conditions

# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters

# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters
  - "C-mutable" Python data structures (dicts, lists, ...)



# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters
  - "C-mutable" Python data structures (dicts, lists, strings, tuples, integers etc.)

# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters
  - "C-mutable" Python data structures (dicts, lists, strings, tuples, integers etc.)
  - internal global state

# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters
  - "C-mutable" Python data structures (dicts, lists, strings, tuples, integers etc.)
  - internal global state
  - atomic APIs

# GIL

- Global Interpreter Lock
- protects against race conditions:
  - reference counters
  - "C-mutable" Python data structures (dicts, lists, strings, tuples, integers etc.)
  - internal global state
  - atomic APIs
- C extensions/binary modules are written assuming GIL exists

# GIL

- holding the GIL not needed when:

# GIL

- holding the GIL not needed when:
  - waiting for I/O

# GIL

- holding the GIL not needed when:
  - waiting for I/O
  - executing code that doesn't access Python data structures

# GIL: scenario I

## single thread



# GIL: scenario 1

T0  running



# GIL: scenario 1



# GIL: scenario 2

## two threads

# GIL: scenario 2

T0 running



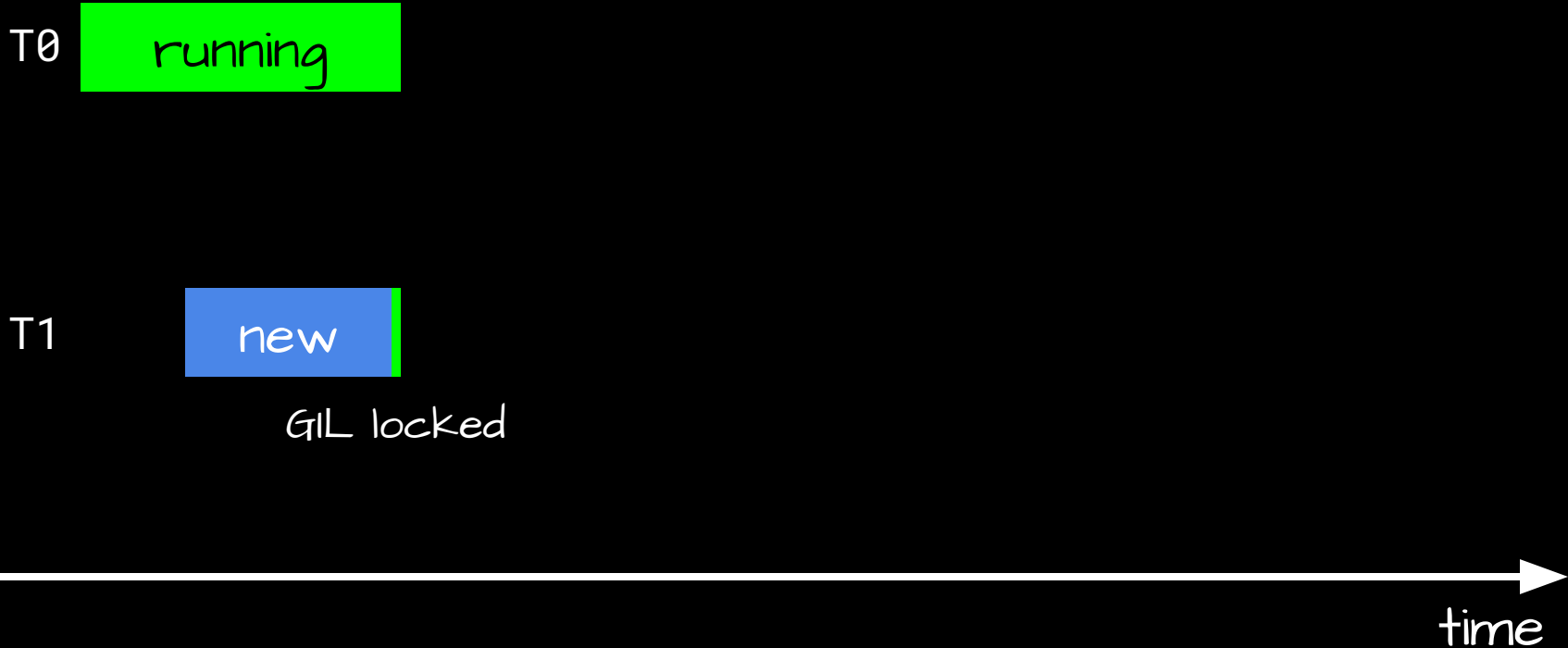
# GIL: scenario 2

T0  running

T1  new



# GIL: scenario 2



# GIL: scenario 2

T0  running

T1  new

GIL locked  
`cv_wait(GIL, interval)`



# GIL: scenario 2

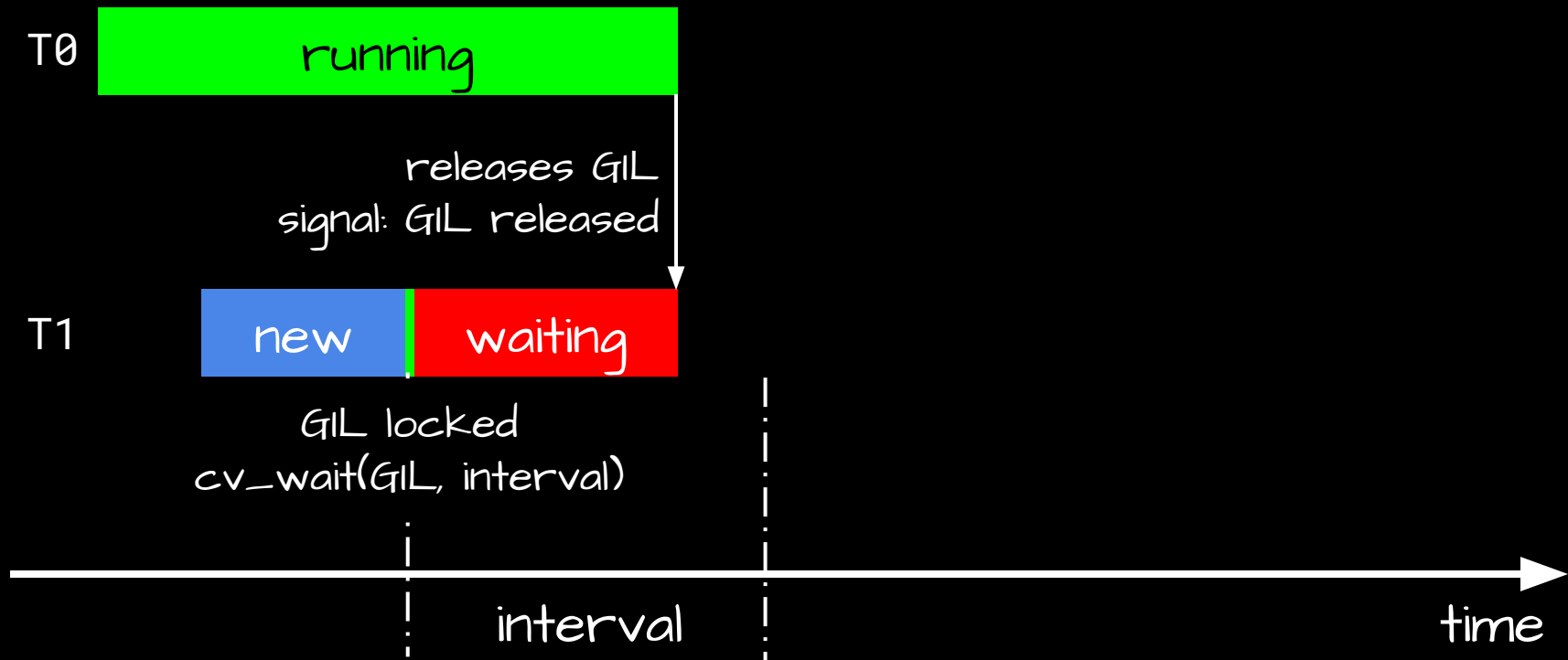




# GIL: scenario 2



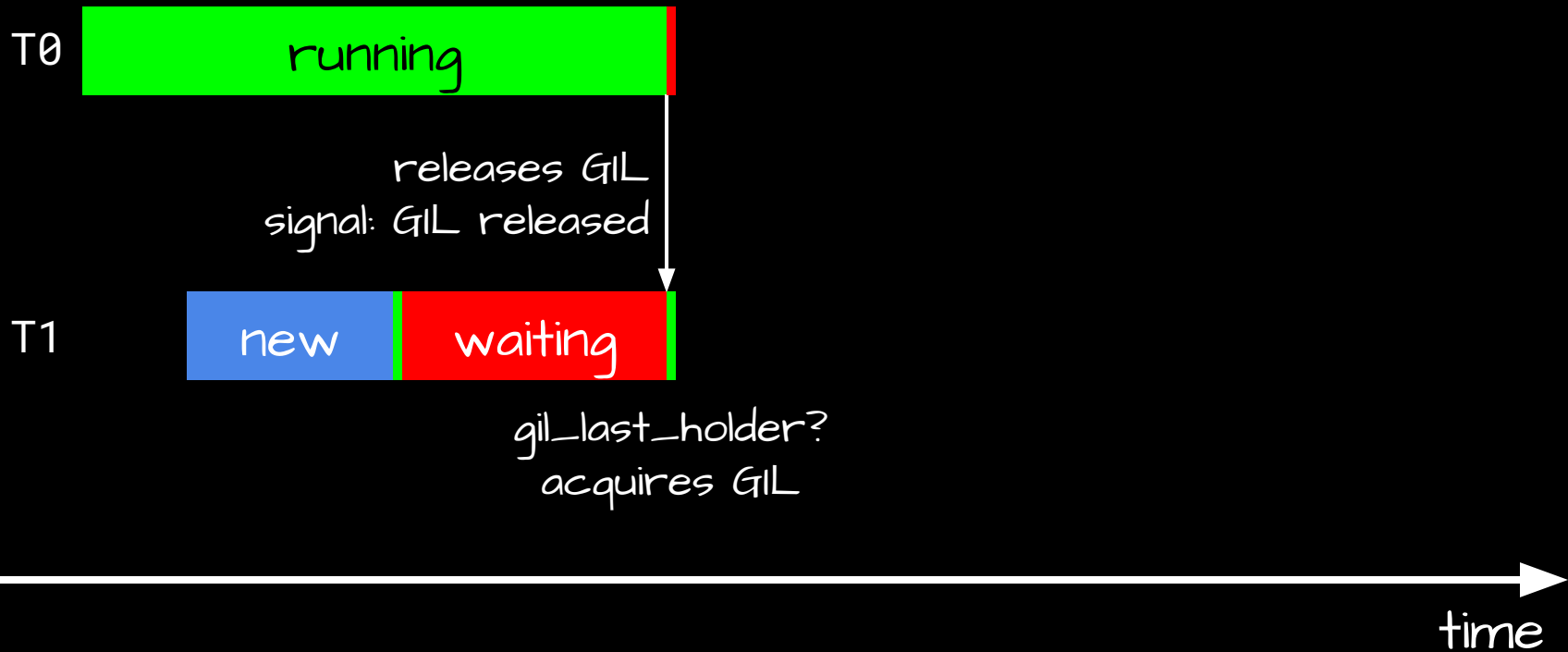
# GIL: scenario 2



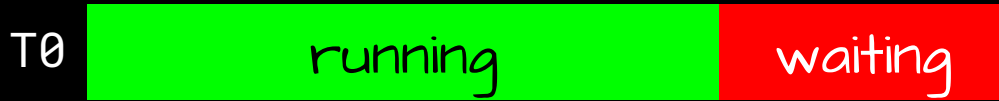
# GIL: scenario 2



# GIL: scenario 2



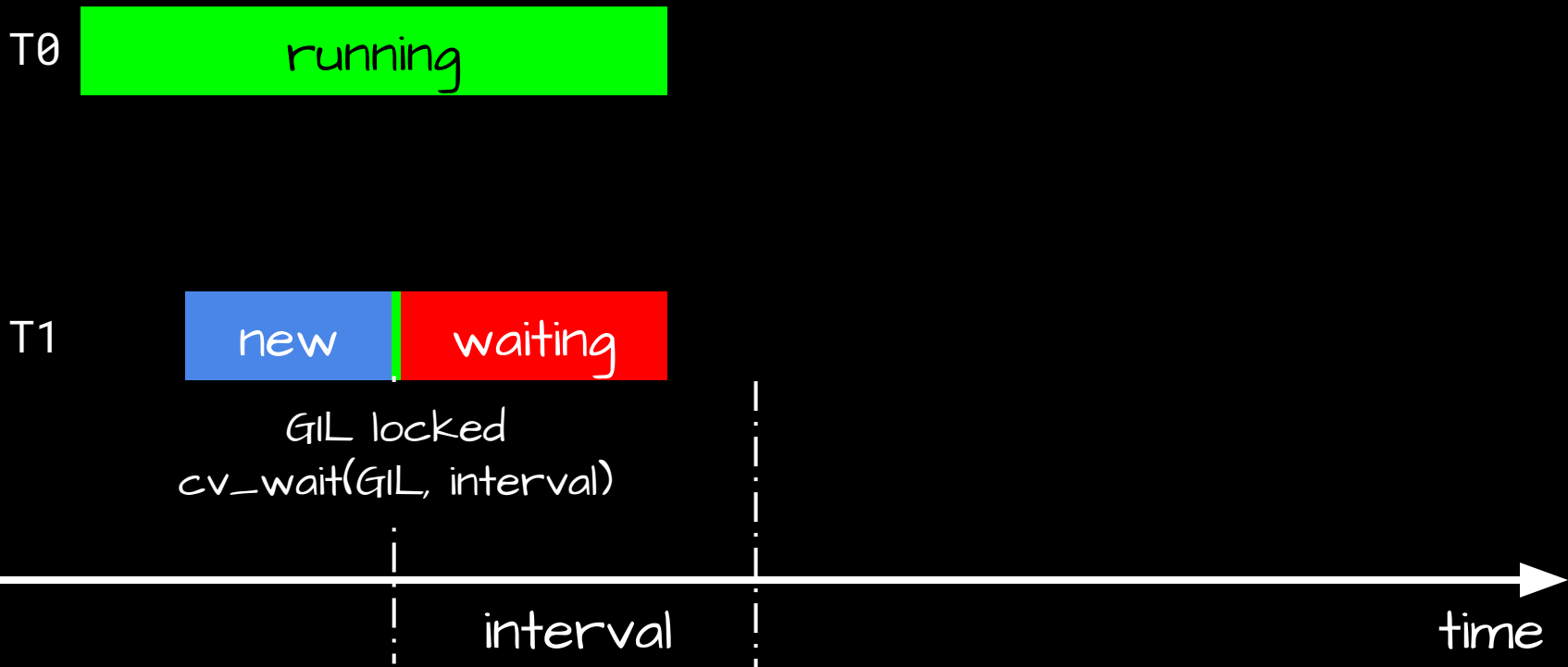
# GIL: scenario 2



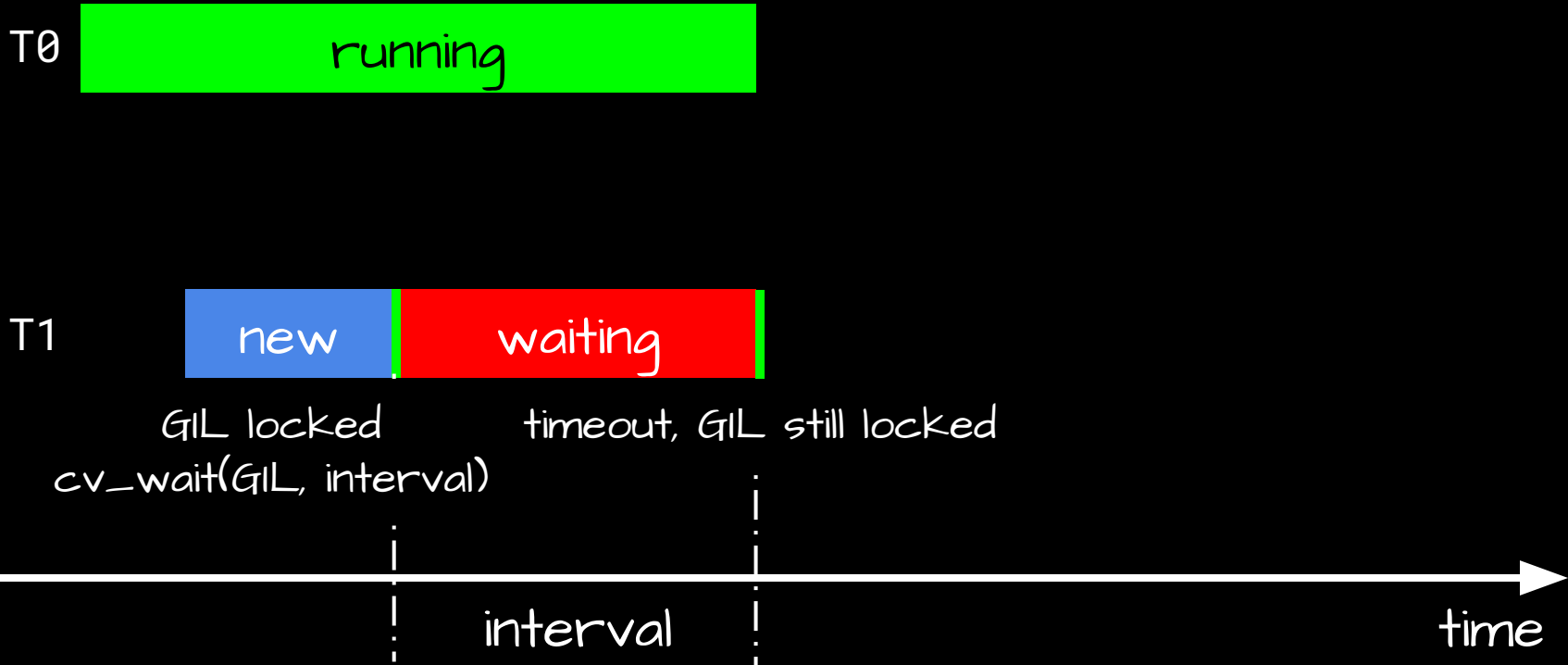
# GIL: scenario 3

## two threads and timeout

# GIL: scenario 3

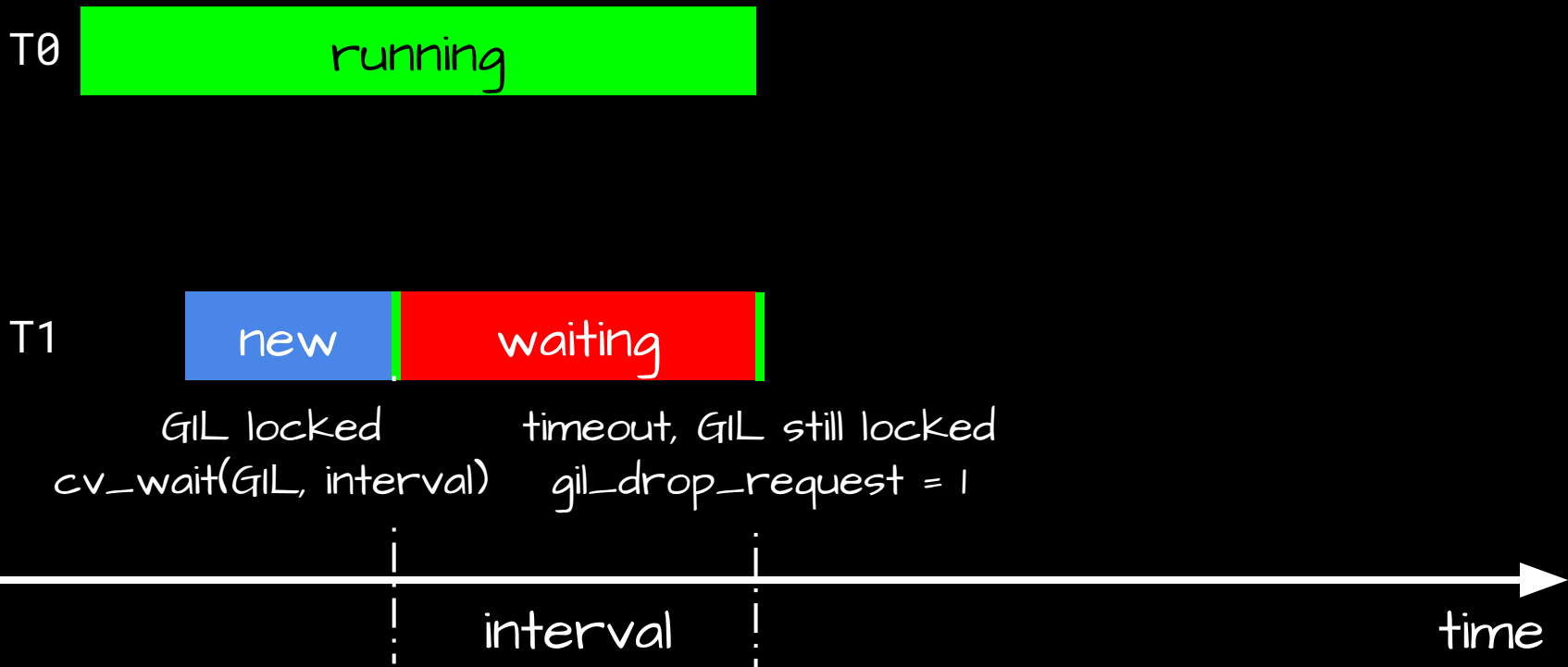


# GIL: scenario 3

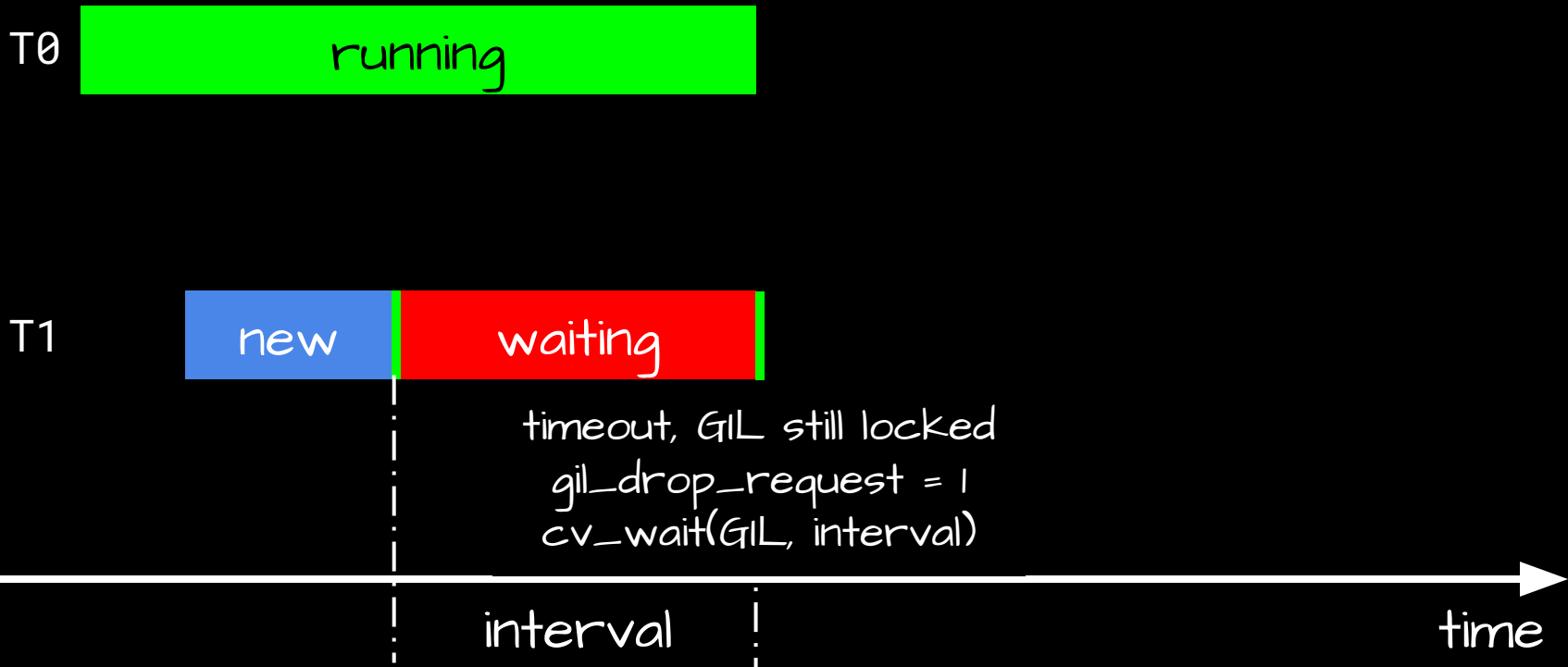




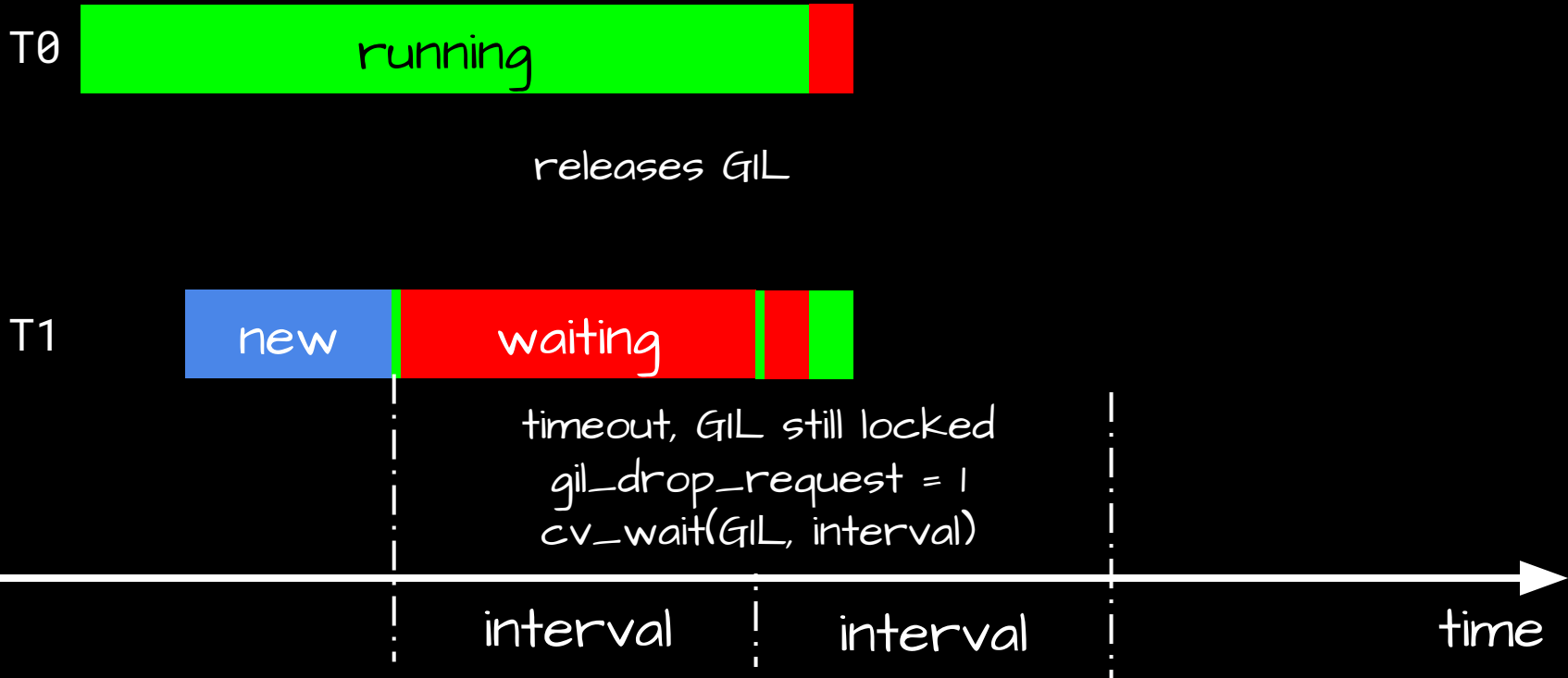
# GIL: scenario 3



# GIL: scenario 3



# GIL: scenario 3

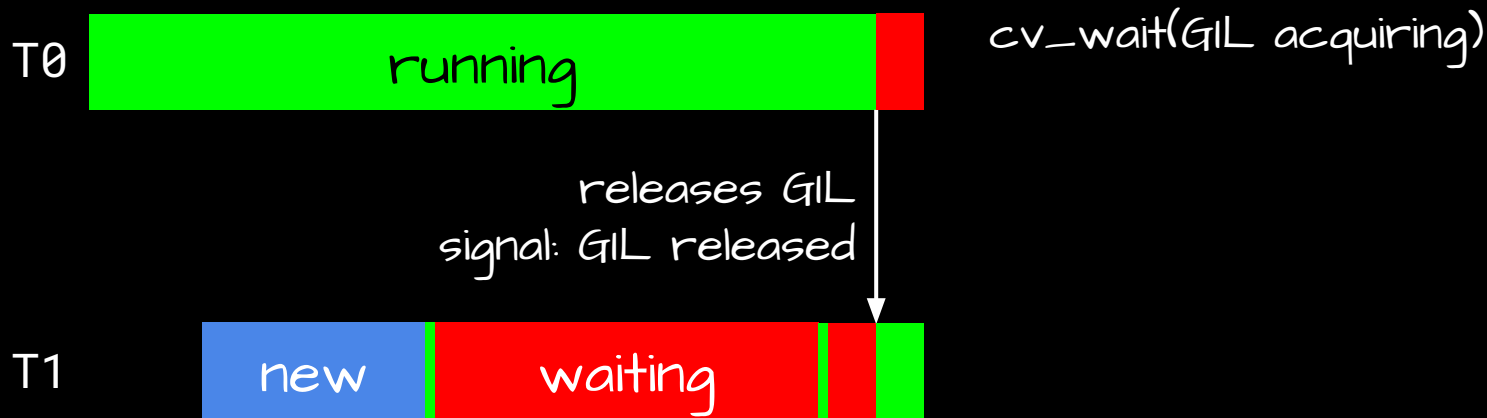


# GIL: scenario 3

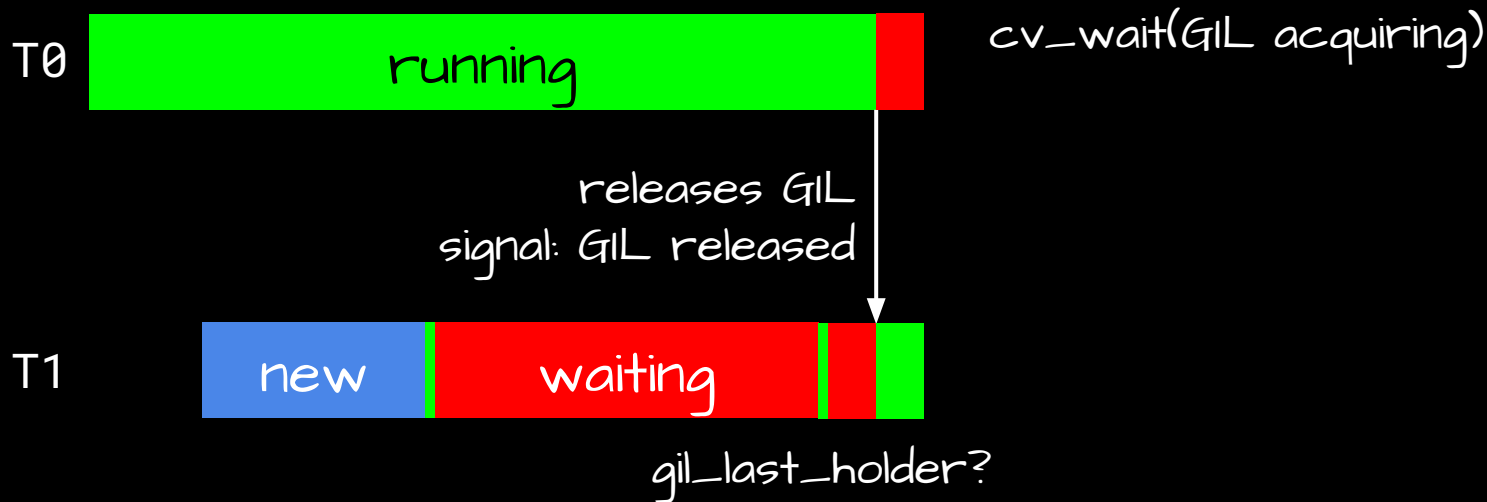


time

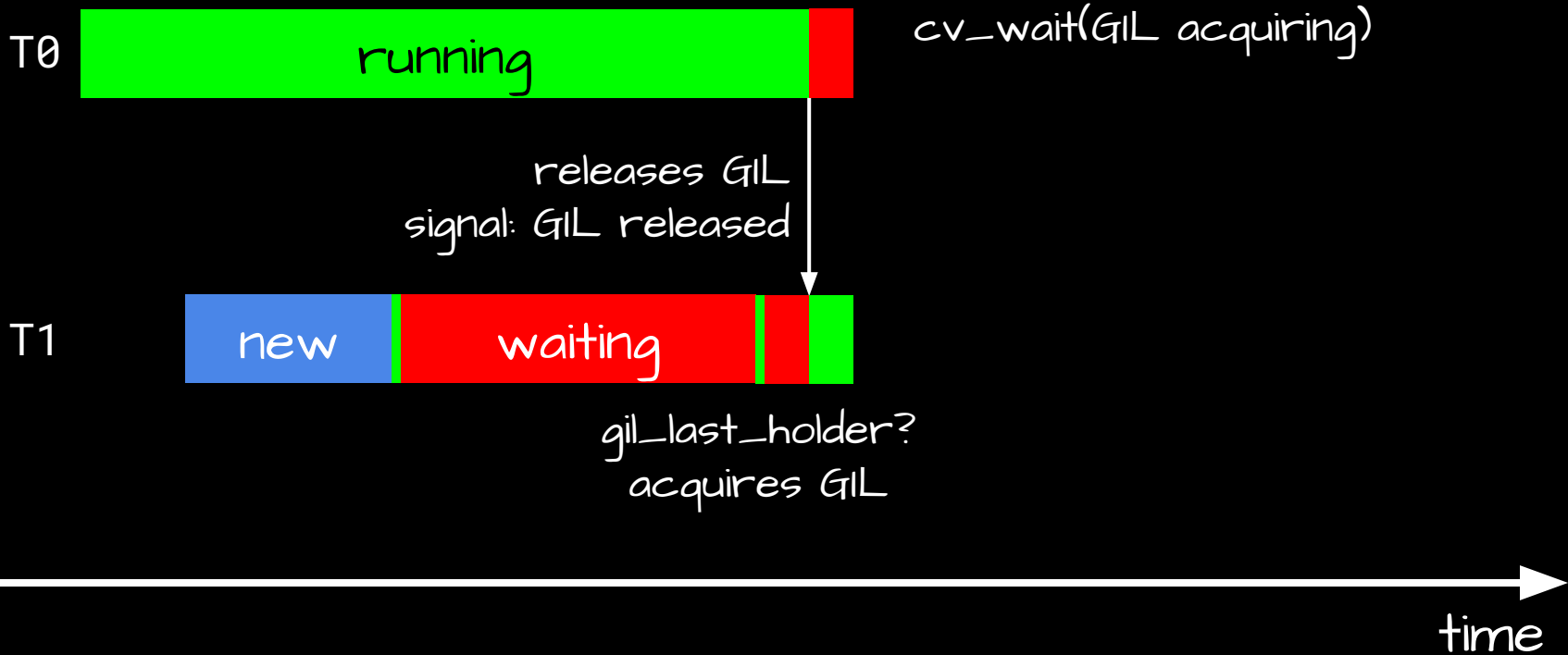
# GIL: scenario 3



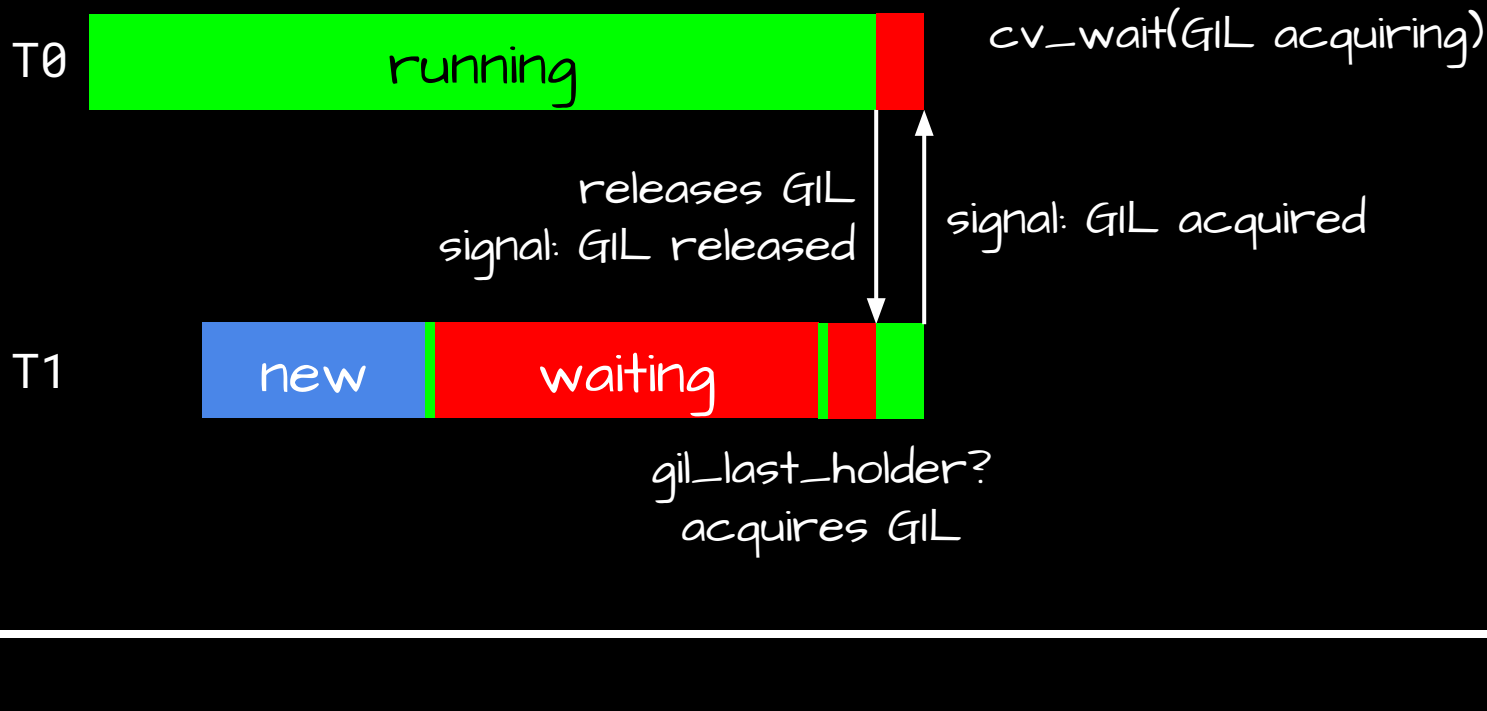
# GIL: scenario 3



# GIL: scenario 3

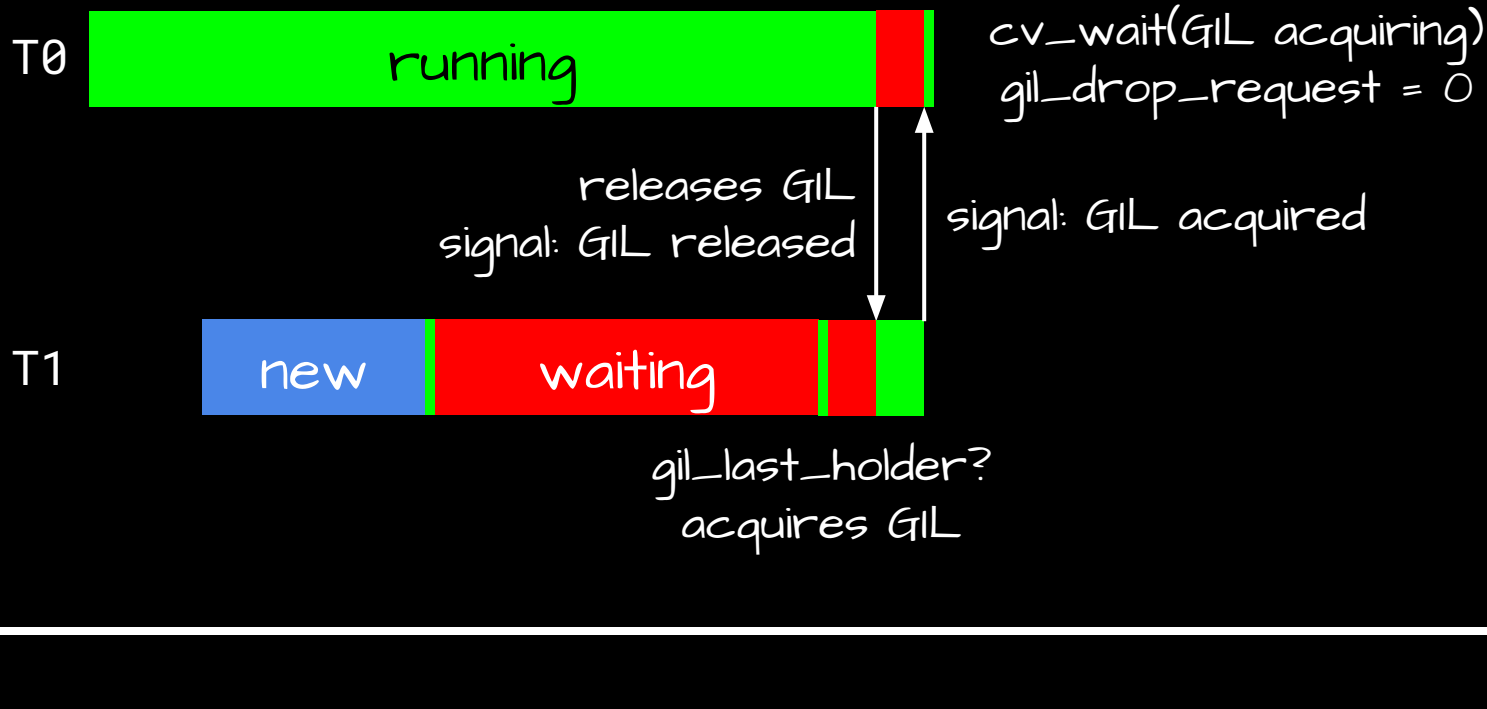


# GIL: scenario 3

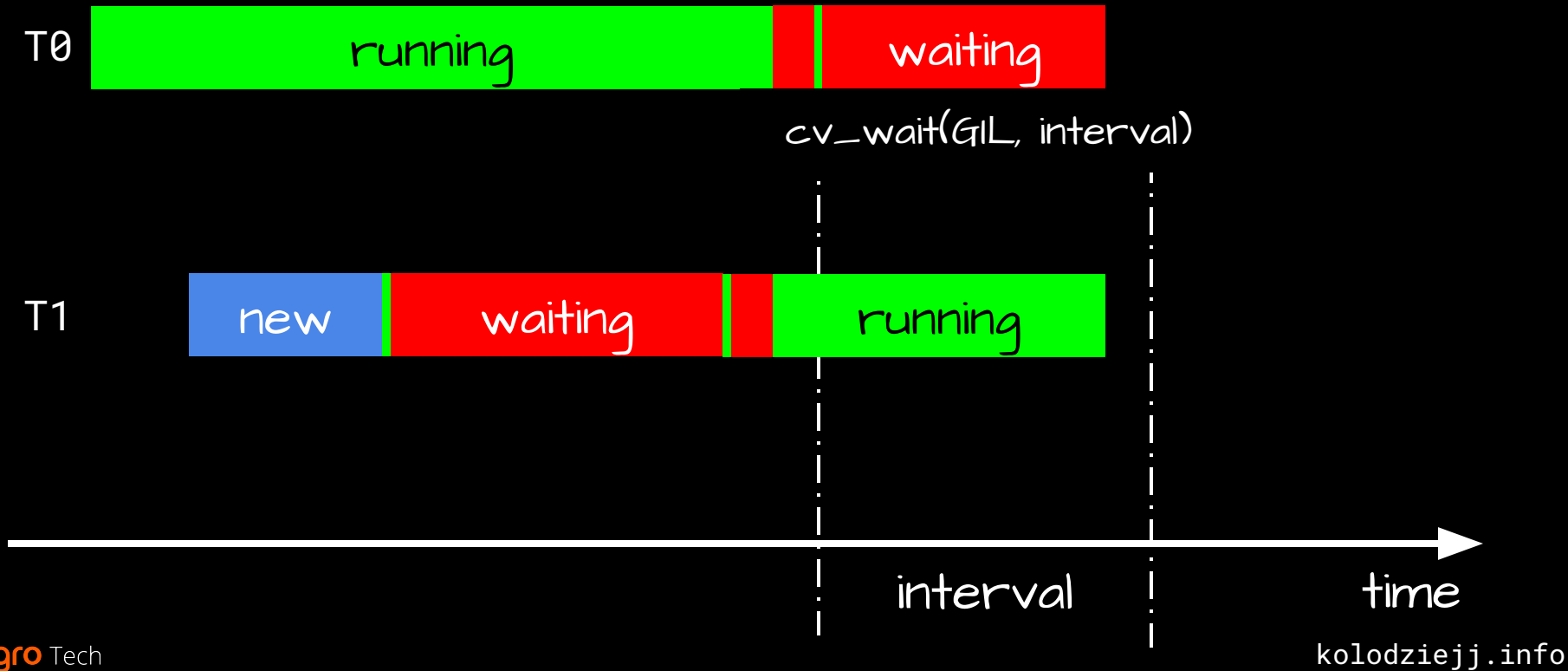




# GIL: scenario 3



# GIL: scenario 3



# GIL: implementation

- "ceval" loop: [Python/ceval.c](#)
  - the GIL part: [Python/ceval\\_gil.c](#)
- GIL data structure: [Include/internal/pycore\\_gil.h](#)

GILL

De-GIL'ing

# De-GIL'ing expectations

# De-GIL'ing expectations

I. Performance:

# De-GIL'ing expectations

## I. Performance:

- for multi-threaded code: must scale with number of cores



# De-GIL'ing expectations

## I. Performance:

- for multi-threaded code: must scale with number of cores
- single-threaded code: not slower

# De-GIL'ing expectations

## 1. Performance:

- for multi-threaded code: must scale with number of cores
- single-threaded code: not slower

## 2. Compatible with C-extensions

# De-GIL'ing expectations

1. Performance:
  - for multi-threaded code: must scale with number of cores
  - single-threaded code: not slower
2. Compatible with C-extensions
3. CPython codebase: not significantly more complex than with the GIL

# De-GIL'ing attempts

# De-GIL'ing attempts

- 1996: free-threading patch

# De-GIL'ing attempts

- 1996: free-threading patch
- 2008: python-safethread

# De-GIL'ing attempts

- 1996: free-threading patch
- 2008: python-safethread
- 2016: GILectomy

# De-GIL'ing attempts

- 1996: free-threading patch
- 2008: python-safethread
- 2016: GILectomy
- 2022: PEP 684 - A Per-Interpreter GIL (accepted)



# De-GIL'ing attempts

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython

# De-GIL'ing attempts

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:

Lock Optional in CPython:

- changes in reference counting, garbage collection, thread-safety of container types, memory allocation... (go read the PEP)

# De-GIL'ing attempts

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:

Lock Optional in CPython:

- changes in reference counting, garbage collection, thread-safety of container types, memory allocation... (go read the PEP)
- `--disable-gil`

# De-GIL'ing attempts

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:

Lock Optional in CPython:

- changes in reference counting, garbage collection, thread-safety of container types, memory allocation... (go read the PEP)
- `--disable-gil`
- long-term: making it the default

# De-GIL'ing attempts

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:
  - changes in reference counting, garbage collection, thread-safety of container types, memory allocation... (go read the PEP)
  - `--disable-gil`
  - long-term: making it the default
  - accepted for 3.13!

# De-GIL'ing expectations

1. Performance:
  - for multi-threaded code: must scale with number of cores
  - single-threaded code: not slower
2. Compatible with C-extensions
3. CPython codebase: not significantly more complex than with the GIL

# De-GIL'ing expectations

1. Performance:
  - for multi-threaded code: must scale with number of cores
  - single-threaded code: not significantly slower
2. Gradual migration path for C-extensions
3. CPython codebase: not significantly more complex than with the GIL

In the meantime:  
what to do?



# What to do?

- single-threaded, pure Python: nothing

# What to do?

- single-threaded, pure Python: nothing
- single-threaded Python with multi-threaded C-extensions:
  - may use all the cores already
  - profile

# What to do?

- single-threaded, pure Python: nothing
- single-threaded Python with multi-threaded C-extensions:
  - may use all the cores already
  - profile
- multi-threaded, I/O-bound:
  - profile
  - add more threads

# What to do?

- asynchronous, I/O-bound:

# What to do?

- asynchronous, I/O-bound:
  - run multiple processes

# What to do?

- asynchronous, I/O-bound:
  - run multiple processes
  - profile

# What to do?

- asynchronous, I/O-bound:
  - run multiple processes
  - profile
- asynchronous with thread pools for synchronous code:

# What to do?

- asynchronous, I/O-bound:
  - run multiple processes
  - profile
- asynchronous with thread pools for synchronous code:
  - profile
  - add more threads



# What to do?

- multi-threaded, CPU-bound:
  - profile
  - use multiprocessing if possible

# What to do?

- multi-threaded, CPU-bound:
  - profile
  - use multiprocessing if possible
  - offload the CPU-intensive work to a C-extension:
    - look for existing one
    - or write your own

# What to do?

- multi-threaded, CPU-bound:
  - profile
  - use multiprocessing if possible
  - offload the CPU-intensive work to a C-extension:
    - look for existing one
    - or write your own
  - try Cython with `nogil`

# What to do?

- multi-threaded, CPU-bound:
  - profile
  - use multiprocessing if possible
  - offload the CPU-intensive work to a C-extension:
    - look for existing one
    - or write your own
  - try Cython with `nogil`
  - research non-Python alternatives, compare pros and cons

# What to do?

- multi-threaded, CPU-bound:
  - profile
  - use multiprocessing if possible
  - offload the CPU-intensive work to a C-extension:
    - look for existing one
    - or write your own
  - try Cython with `nogil`
  - research non-Python alternatives, compare pros and cons
- profile, profile, profile (cProfile, timeit, perf...)

Thank you! :)

[kolodziejj.info/talks/gil](http://kolodziejj.info/talks/gil)