GIL

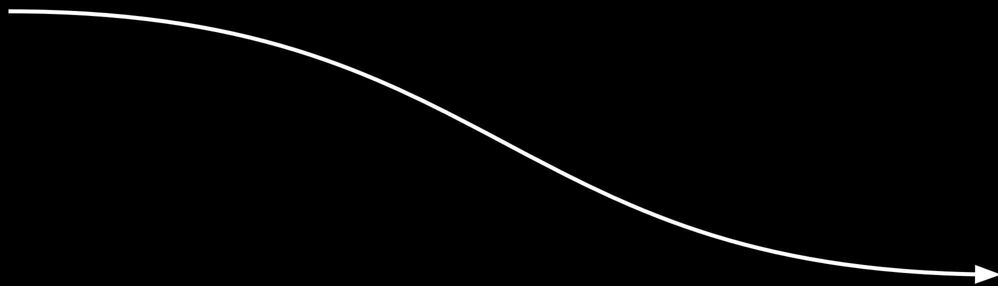
Google: can Python into threads?

„CPython doesn't support multi-threading"

1992

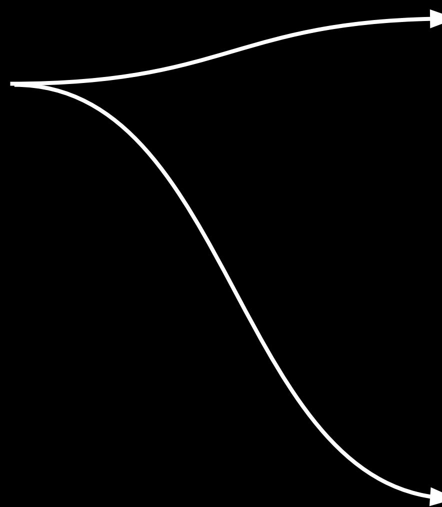
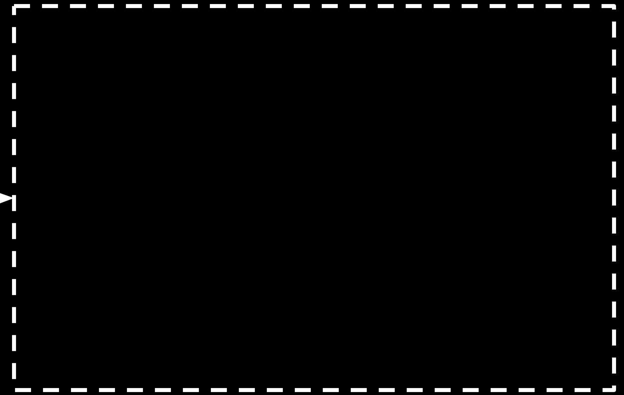
1992

- ja



1992

- ja
- Java & JavaScript

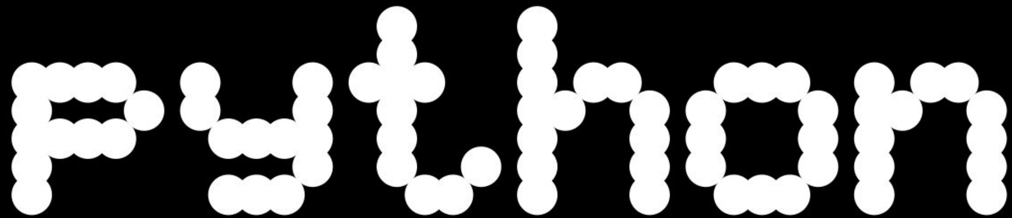



1992

- ja
- Java & JavaScript
- wielordzeniowe CPU dla zwykłych śmiertelników

1992

- ja
- Java & JavaScript
- wielordzeniowe CPU dla zwykłych śmiertelników
- Python (z obsługą wielowątkowości)



× „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: ale najpierw o...

Parallelism

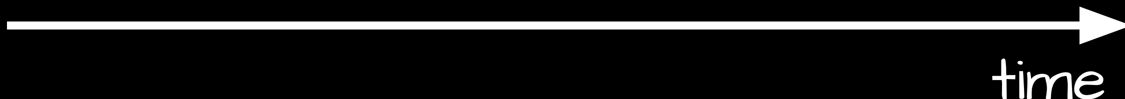
Parallelism

- wymaga wielu rdzeni CPU

Parallelism

- wymaga wielu rdzeni CPU

Wątek 0, rdzeń 0



Parallelism

- wymaga wielu rdzeni CPU

Wątek 0, rdzeń 0

Wątek 1, rdzeń 1

Parallelism

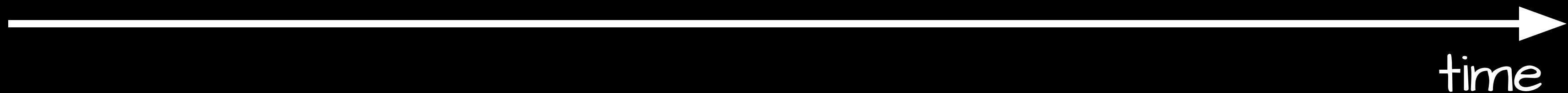
- wymaga wielu rdzeni CPU

Wątek 0, rdzeń 0

Wątek 1, rdzeń 1

- w przeciwieństwie do "concurrency"

Jeden rdzeń



Wątki

Wątki

- proces

Wątki

- proces:
  - instancja aplikacji

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków
  - współdzielona pamięć

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków
  - współdzielona pamięć
- `import threading`

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków
  - współdzielona pamięć
- `import threading`
  - wątki systemowe (kernel/natywne/Posix)

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków
  - współdzielona pamięć
- `import threading`
  - wątki systemowe (kernel/natywne/Posix)
  - w przeciwieństwie do: zielonych wątków, ko-rutyn etc.

Wątki

- proces:
  - instancja aplikacji
  - jeden lub więcej wątków
  - współdzielona pamięć
- `import threading`
  - wątki systemowe (kernel/natywne/Posix)
  - w przeciwieństwie do: zielonych wątków, ko-rutyn etc.
- stan wątku:
  - gotowy
  - pracuje
  - czeka

Blokady i wywołania systemowe

- blokady:

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach

- blokady:
    - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
    - mutex - **mut**ual **ex**clusion

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
  - mutex - **mut**ual **ex**clusion
  - lock (hold) i unlock (release)

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
  - mutex - **mut**ual **ex**clusion
  - lock (hold) i unlock (release)
- wywołania systemowe:

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
  - mutex - **mut**ual **ex**clusion
  - lock (hold) i unlock (release)
- wywołania systemowe:
  - API systemu operacyjnego/kernela

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
  - mutex - **mut**ual **ex**clusion
  - lock (hold) i unlock (release)
- wywołania systemowe:
  - API systemu operacyjnego/kernela
  - funkcje dla aplikacji: dostęp do I/O, zarządzanie procesami...

- blokady:
  - narzędzie do radzenia sobie z race-conditions na współdzielonych zasobach
  - mutex - **mut**ual **ex**clusion
  - lock (hold) i unlock (release)
- wywołania systemowe:
  - API systemu operacyjnego/kernela
  - funkcje dla aplikacji: dostęp do I/O, zarządzanie procesami…
  - może być blokujące (n.p.: dostęp do I/O, sleep…)

Zarządzanie pamięcią

Zarządzanie pamięcią

- CPython:

Zarządzanie pamięcią

- CPython:
    - reference counting

Zarządzanie pamięcią

- CPython:
  - reference counting (współdzielony zasób - wymaga locków!)

Zarządzanie pamięcią

- CPython:
  - reference counting (współdzielony zasób - wymaga locków!)
  - + garbage collector - tylko do odniesień cyklicznych

Zarządzanie pamięcią

- CPython:
  - reference counting (współdzielony zasób - wymaga locków!)
  - + garbage collector - tylko do odniesień cyklicznych
- inna opcja: tracing garbage collector

Zarządzanie pamięcią

- CPython:
  - reference counting (współdzielony zasób - wymaga locków!)
  - + garbage collector - tylko do odniesień cyklicznych
- inna opcja: tracing garbage collector
  - bardziej funkcjonalny ale bardziej złożony

Zarządzanie pamięcią

- CPython:
  - reference counting (współdzielony zasób - wymaga locków!)
  - + garbage collector - tylko do odniesień cyklicznych
- inna opcja: tracing garbage collector
  - bardziej funkcjonalny ale bardziej złożony
  - np. w JVM czy C#

Pythonowy bytecode

Pythonowy bytecode

- kod w Pythonie -> kompilacja -> bytecode

Pythonowy bytecode

- Kod w Pythonie -> kompilacja -> bytecode

```
a = 1

print(a)
```

Pythonowy bytecode

- Kod w Pythonie -> Kompilacja -> 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
```

Pythonowy bytecode

- kod w Pythonie -> kompilacja -> 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 -> wykonanie

GIL

wreszcie

GIL

- Global Interpreter Lock

GIL

- Global Interpreter Lock
- chroni przed race-conditions:

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
    - reference counters

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
  - reference counters
  - Pythonowe obiekty mutowalne z poziomu C API: (słowniki, listy, …)

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
  - reference counters
  - Pythonowe obiekty mutowalne z poziomu C API: (słowniki, listy, stringi, tuple, liczby itd.)

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
  - reference counters
  - Pythonowe obiekty mutowalne z poziomu C API: (słowniki, listy, stringi, tuple, liczby itd.)
  - wewnętrzny stan globalny

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
  - reference counters
  - Pythonowe obiekty mutowalne z poziomu C API: (słowniki, listy, stringi, tuple, liczby itd.)
  - wewnętrzny stan globalny
  - atomiczne API CPythona

GIL

- Global Interpreter Lock
- chroni przed race-conditions:
  - reference counters
  - Pythonowe obiekty mutowalne z poziomu C API: (słowniki, listy, stringi, tuple, liczby itd.)
  - wewnętrzny stan globalny
  - atomiczne API CPythona
- C-extensions/moduły binarne!

GIL

- trzymanie GILa a:

GIL

- trzymanie GILa a:
  - blokujące operacje (np. I/O)

GIL

- trzymanie GILa a:
  - blokujące operacje (np. I/O)
  - obiekty poza-Pythonowe

GIL: scenariusz 1

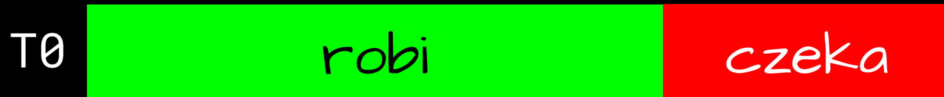
jeden wątek

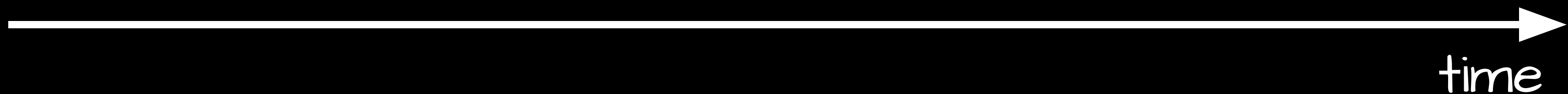
GIL: scenariusz 1

T0 robi

time

GIL: scenariusz 1





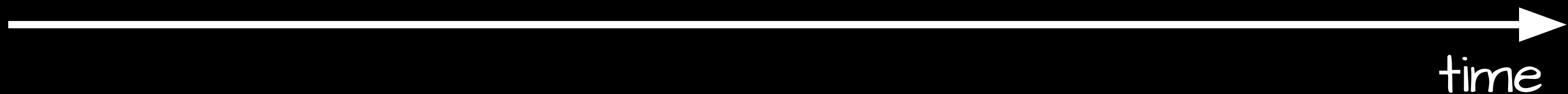
GIL: scenariusz 2

dwa wątki

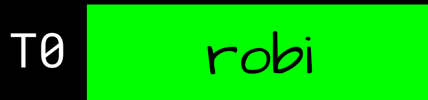
GIL: scenariusz 2

T0 robi

GIL: scenariusz 2



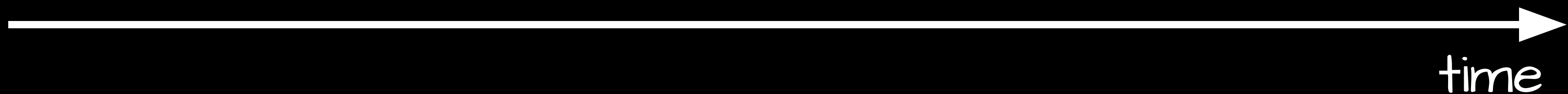
GIL: scenariusz 2



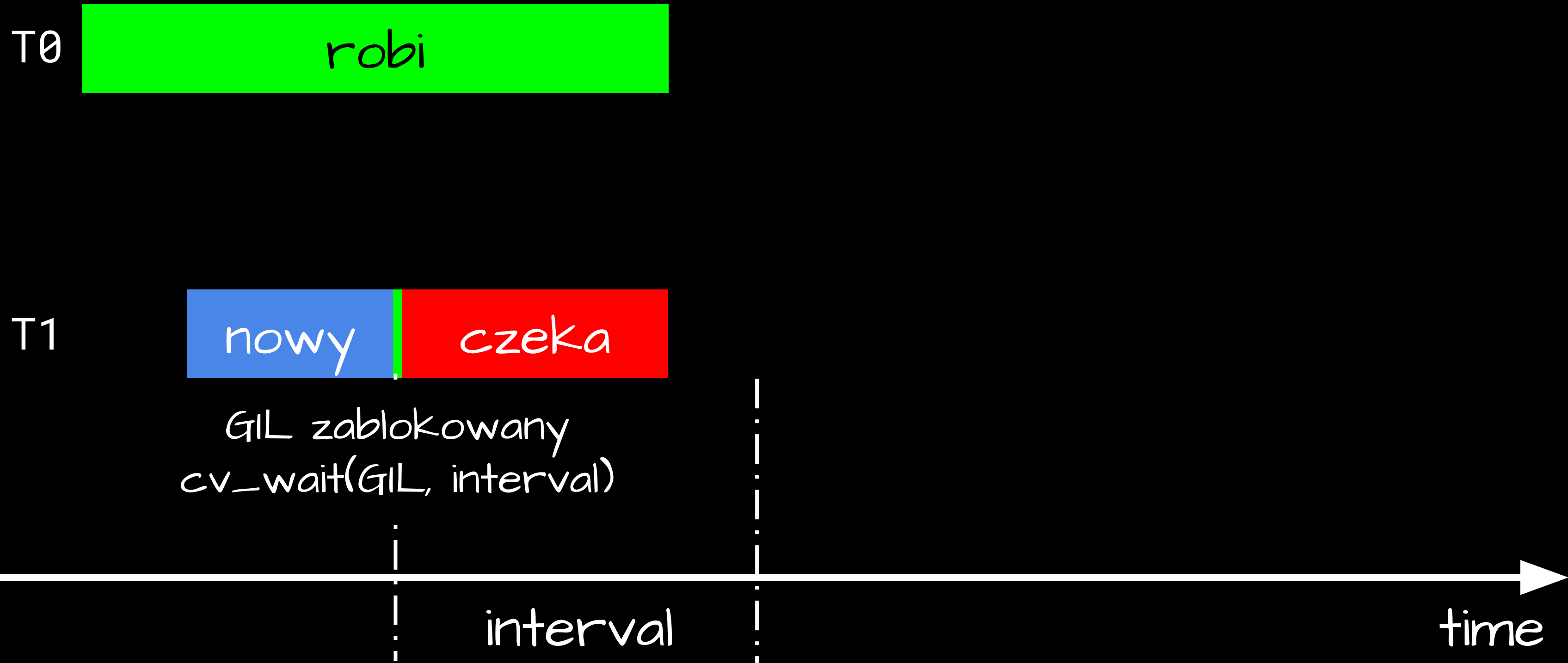
GIL zablokowany

GIL: scenariusz 2

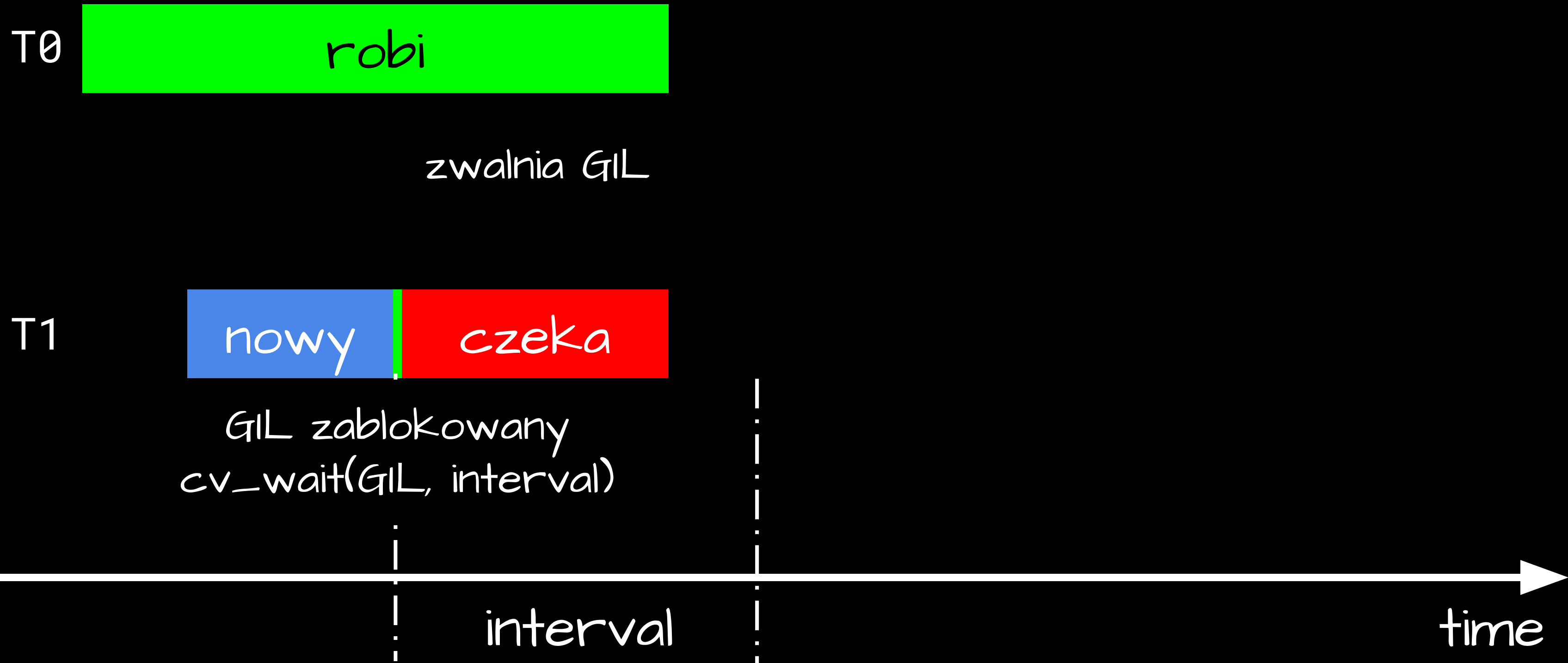
GIL zablokowany
cv_wait(GIL, interval)



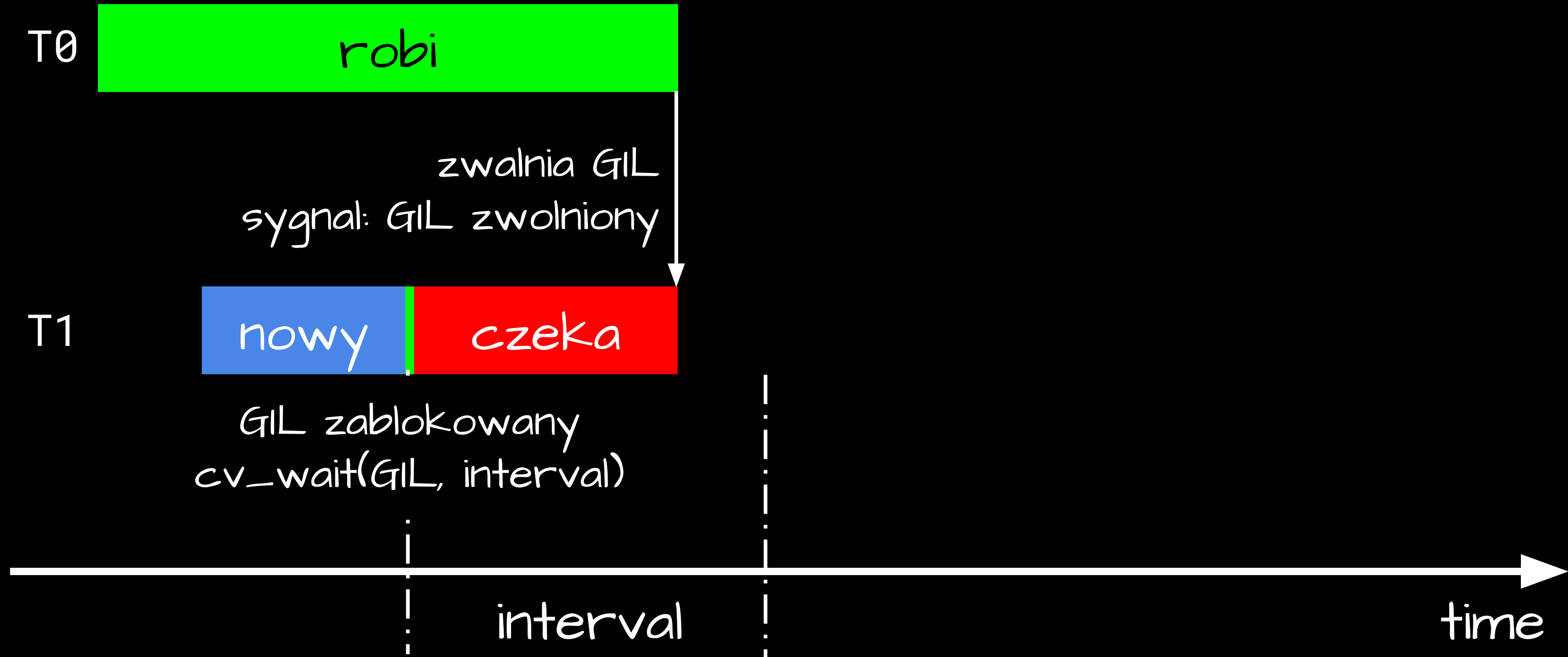
GIL: scenariusz 2



GIL: scenariusz 2



GIL: scenariusz 2



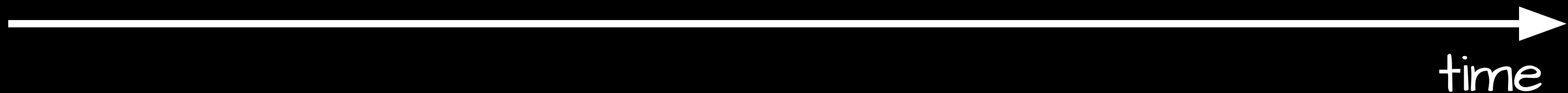
GIL: scenariusz 2

GIL: scenariusz 2

GIL: scenariusz 2

T0

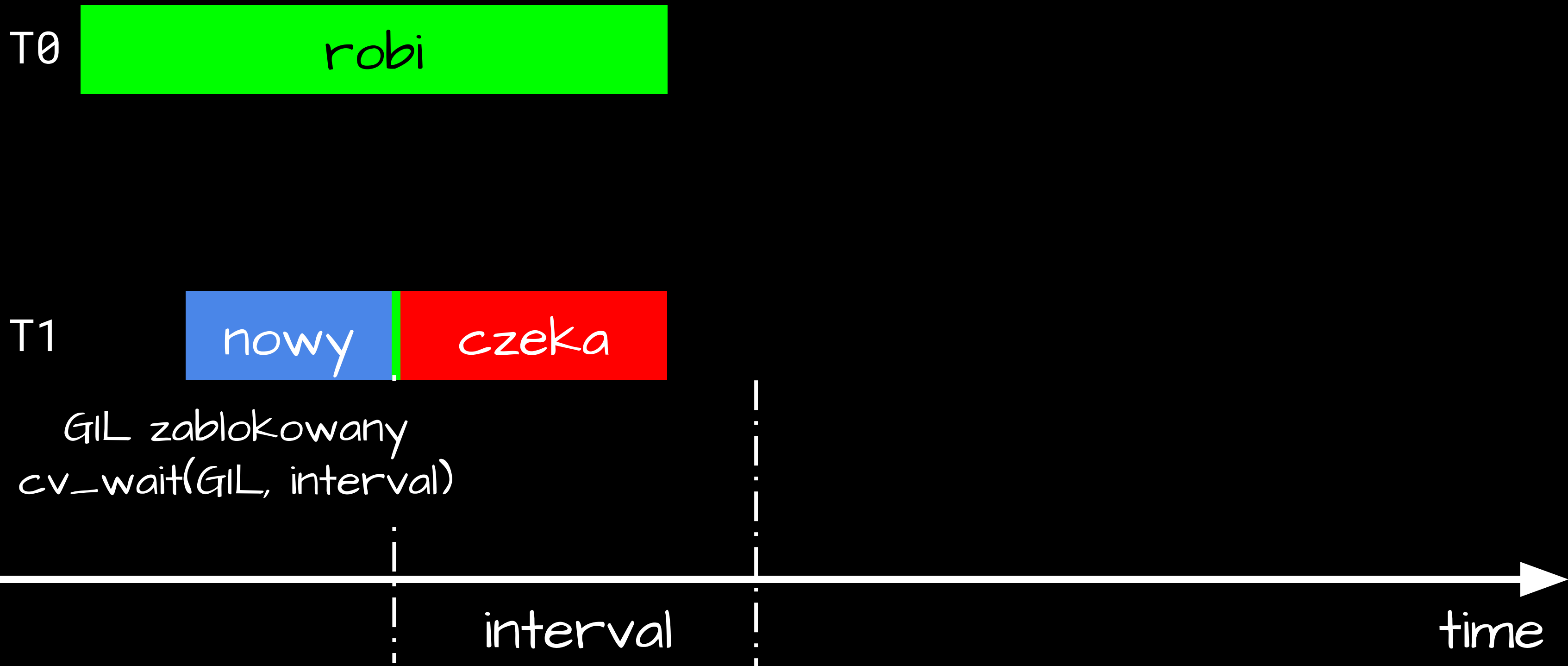
T1



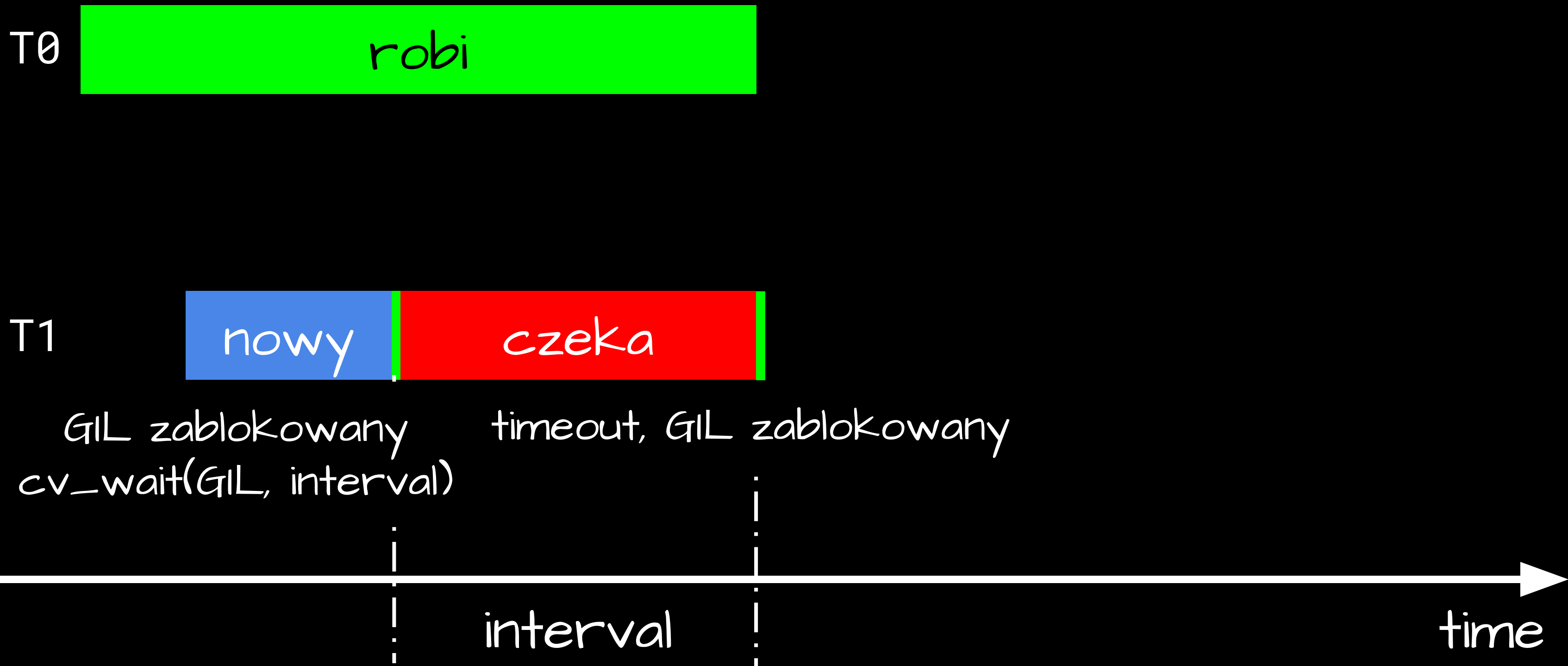
GIL: scenariusz 3

dwa wątki i timeout

GIL: scenariusz 3



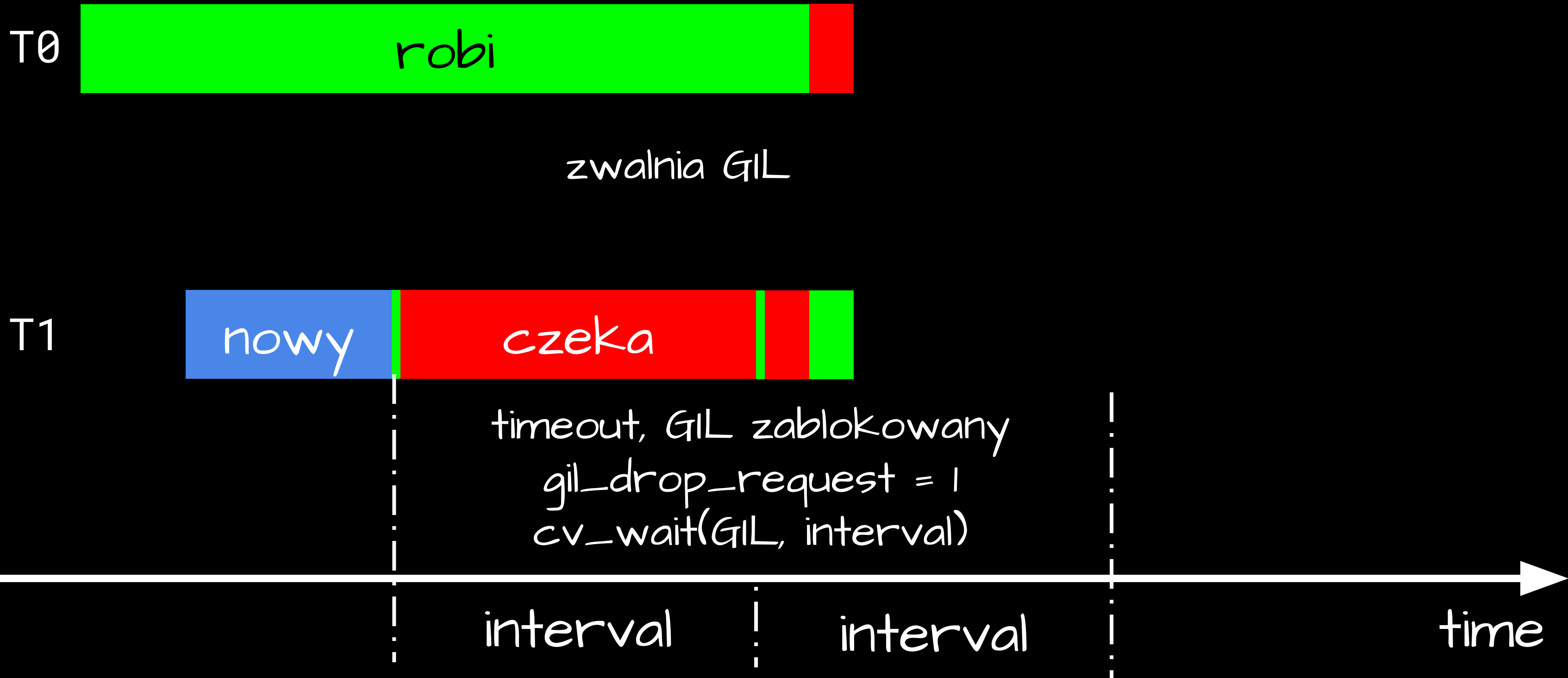
GIL: scenariusz 3



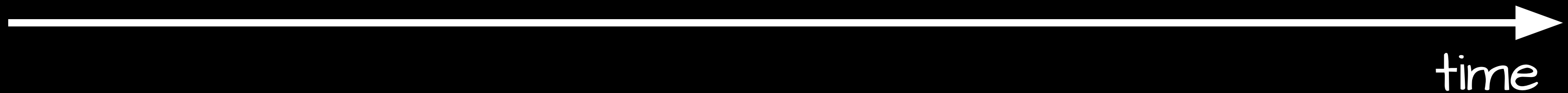
GIL: scenariusz 3

GIL: scenariusz 3

GIL: scenariusz 3



GIL: scenariusz 3



GIL: scenariusz 3

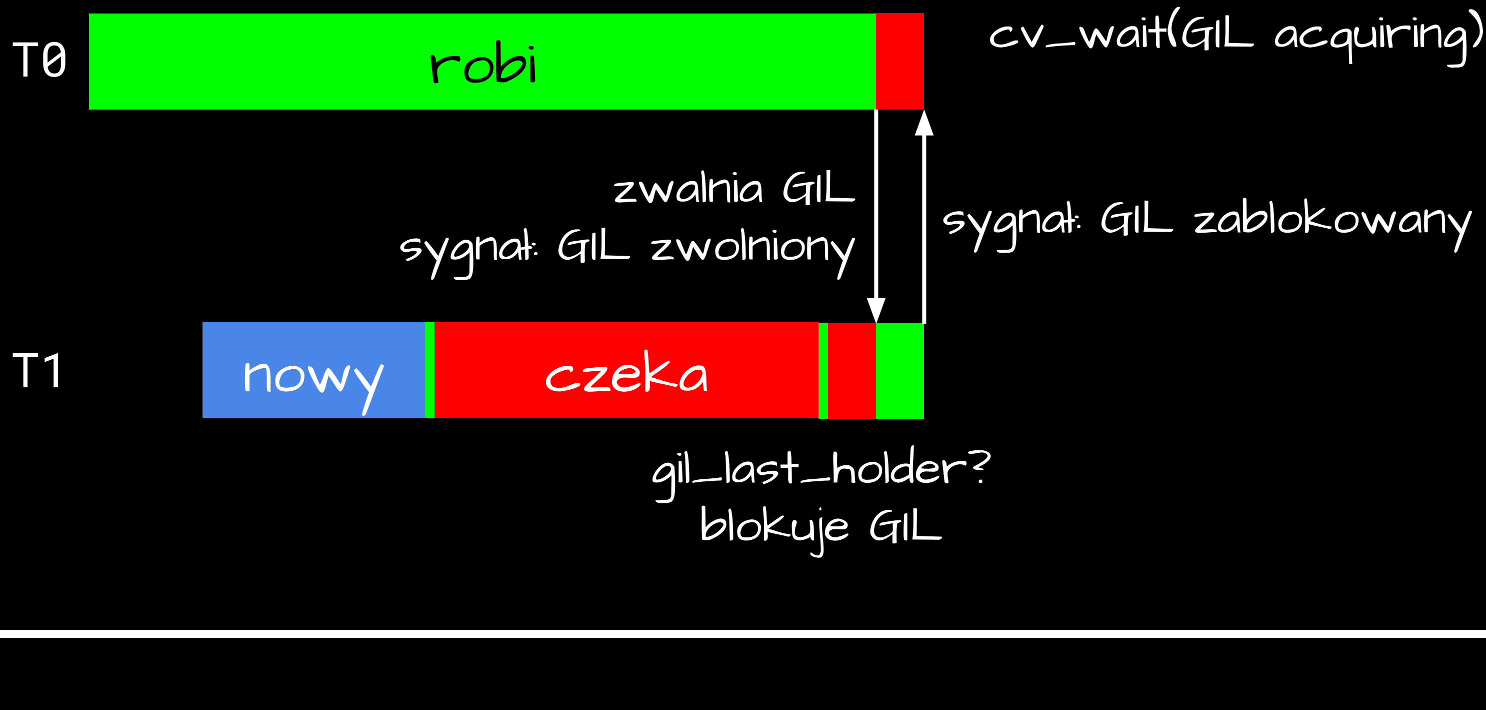
time

GIL: scenariusz 3

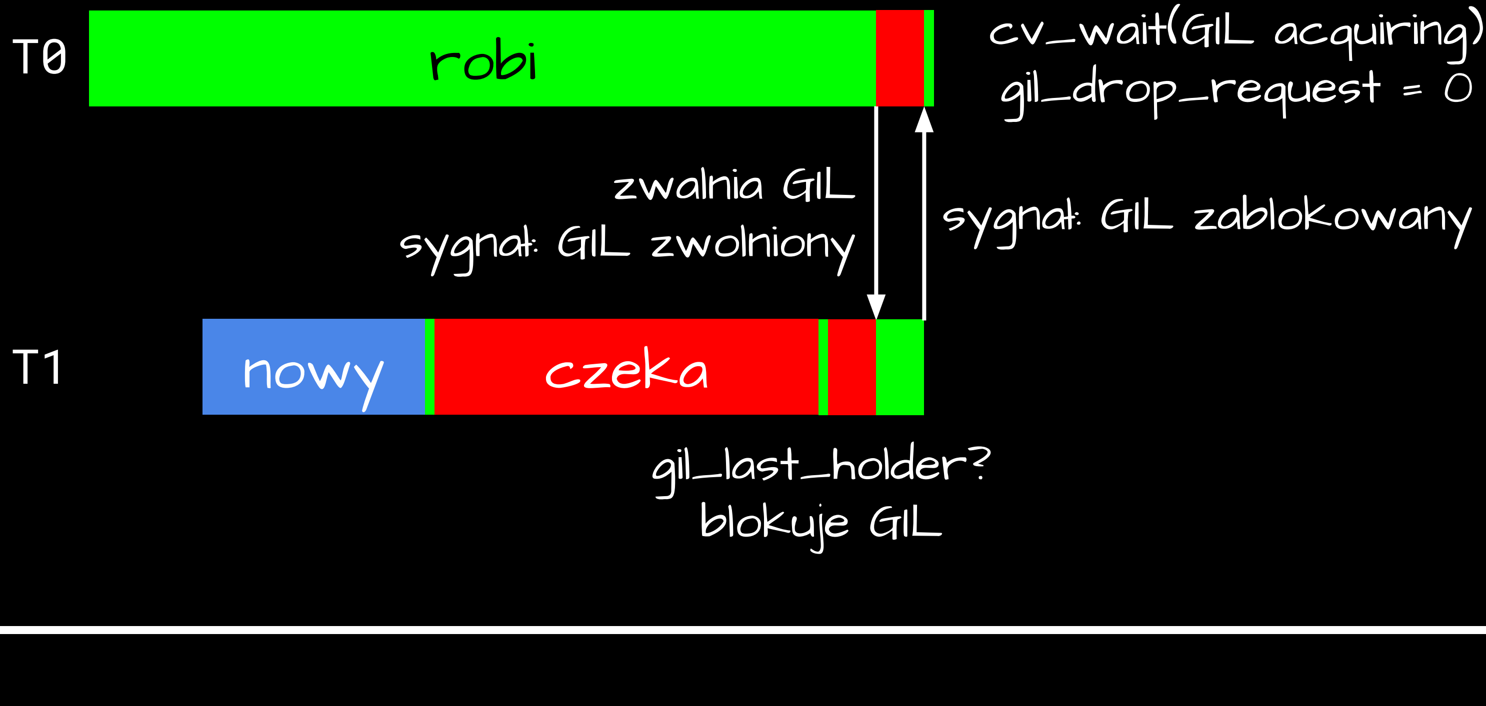
time

GIL: scenariusz 3

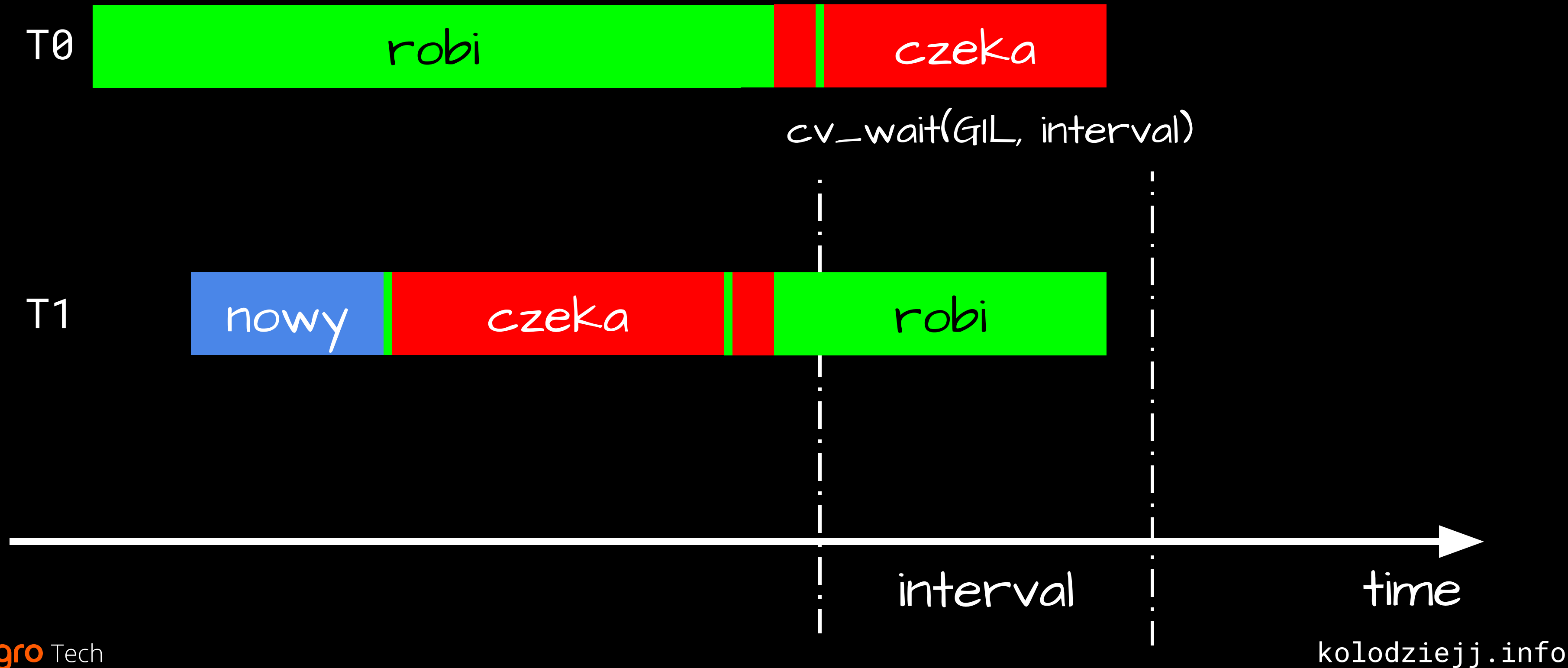
GIL: scenariusz 3



GIL: scenariusz 3



GIL: scenariusz 3



GIL: implementacja

- pętla "ceval": Python/ceval.c
  - sam GIL: Python/ceval_gil.c
- GIL - struktura danych: Include/internal/pycore_gil.h

GIL

De-GIL'ing

De-GIL'ing: oczekiwania

De-GIL'ing: oczekiwania

1. Wydajność:

De-GIL'ing: oczekiwania

1. Wydajność:
    - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni

De-GIL'ing: oczekiwania

1. Wydajność:
   - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni
   - dla kodu jednowątkowego: nie wolniej, niż z GILem

De-GIL'ing: oczekiwania

1. Wydajność:
   - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni
   - dla kodu jednowątkowego: nie wolniej, niż z GILem
2. Kompatybilność wsteczna dla rozszerzeń w C

De-GIL'ing: oczekiwania

1. Wydajność:
   - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni
   - dla kodu jednowątkowego: nie wolniej, niż z GILem
2. Kompatybilność wsteczna dla C-extensions
3. Bez skomplikowanych zmian w kodzie CPython'a

De-GIL'ing: próby

De-GIL'ing: próby

- 1996: free-threading patch

De-GIL'ing: próby

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

De-GIL'ing: próby

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

De-GIL'ing: próby

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

De-GIL'ing: próby

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

De-GIL'ing: próby

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:
  - zmiany w reference counting'u, garbage collection, thread-safety typów "kontenerowych", alokacji pamięci... (warto poczytać PEP)

De-GIL'ing: próby

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:
    - zmiany w reference counting'u, garbage collection, thread-safety typów "kontenerowych", alokacji pamięci... (warto poczytać PEP)
    - `--disable-gil`

De-GIL'ing: próby

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:
  - zmiany w reference counting'u, garbage collection, thread-safety typów "kontenerowych", alokacji pamięci... (warto poczytać PEP)
  - `--disable-gil`
  - long-term: brak GILa jako default

De-GIL'ing: próby

- 2023: PEP 703 - Making the Global Interpreter Lock Optional in CPython:
  - zmiany w reference counting'u, garbage collection, thread-safety typów "kontenerowych", alokacji pamięci... (warto poczytać PEP)
  - `--disable-gil`
  - long-term: brak GILa jako default
  - zaakceptowany do 3.13!

De-GIL'ing: oczekiwania

1. Wydajność:
   - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni
   - dla kodu jednowątkowego: nie wolniej, niż z GILem
2. Kompatybilność wsteczna dla rozszerzeń w C
3. Bez skomplikowanych zmian w kodzie CPython'a

De-GIL'ing: oczekiwania

1. Wydajność:
    - dla kodu wielowątkowego: musi rosnąć z liczbą rdzeni
    - dla kodu jednowątkowego: nie istotnie wolniej, niż z GILem
2. Ścieżka migracji dla rozszerzeń w C
3. Bez skomplikowanych zmian w kodzie CPython'a

Jak żyć w międzyczasie?

Jak żyć?

- jednowątkowy, czysty Python:

Jak żyć?

- jednowątkowy, czysty Python: :)

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:
    - prawdopodobnie już używa wielu rdzeni

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:
    - prawdopodobnie już używa wielu rdzeni
    - profiluj

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:
  - prawdopodobnie już używa wielu rdzeni
  - profiluj
- wielowątkowy, I/O-bound:

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:
  - prawdopodobnie już używa wielu rdzeni
  - profiluj
- wielowątkowy, I/O-bound:
  - profiluj

Jak żyć?

- jednowątkowy, czysty Python: :)
- jednowątkowy Python z wielowątkowymi rozszerzeniami w C:
  - prawdopodobnie już używa wielu rdzeni
  - profiluj
- wielowątkowy, I/O-bound:
  - profiluj
  - dodaj wincyj wątków

Jak żyć?

- kod asynchroniczny, I/O-bound:

Jak żyć?

- kod asynchroniczny, I/O-bound:
  - uruchamiaj proces per rdzeń

Jak żyć?

- kod asynchroniczny, I/O-bound:
  - uruchamiaj proces per rdzeń
  - profiluj

Jak żyć?

- kod asynchroniczny, I/O-bound:
  - uruchamiaj proces per rdzeń
  - profiluj
- kod asynchroniczny z pulami wątków dla wywołań synchronicznych:

Jak żyć?

- kod asynchroniczny, I/O-bound:
  - uruchamiaj proces per rdzeń
  - profiluj
- kod asynchroniczny z pulami wątków dla wywołań synchronicznych:
  - profiluj
  - dodaj wincyj wątków

Jak żyć?

- wielowątkowy, CPU-bound:

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj
  - jeśli to możliwe, użyj multiprocessing

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj
  - jeśli to możliwe, użyj multiprocessing
  - ciężkie obliczenia -> rozszerzenie w C:
    - poszukaj istniejącego rozszerzenia
    - albo napisz własne

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj
  - jeśli to możliwe, użyj multiprocessing
  - ciężkie obliczenia -> rozszerzenie w C:
    - poszukaj istniejącego rozszerzenia
    - albo napisz własne
  - spróbuj Cython'a (`nogil`)

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj
  - jeśli to możliwe, użyj multiprocessing
  - ciężkie obliczenia -> rozszerzenie w C:
    - poszukaj istniejącego rozszerzenia
    - albo napisz własne
  - spróbuj Cython'a (`nogil`)
  - szukaj poza-Pythonowych alternatyw :(, zważ wady i zalety

Jak żyć?

- wielowątkowy, CPU-bound:
  - profiluj
  - jeśli to możliwe, użyj multiprocessing
  - ciężkie obliczenia -> rozszerzenie w C:
    - poszukaj istniejącego rozszerzenia
    - albo napisz własne
  - spróbuj Cython'a (`nogil`)
  - szukaj poza-Pythonowych alternatyw :(, zważ wady i zalety
- profiluj, profiluj, profiluj (timeit, pyistrument, perf…)

Dziękuję! :)

kolodziejj.info/talks/gil