[544] Locks

Tyler Caraza-Harter

Learning Objectives

- identify critical sections in code
- protect critical sections with locks
- write code that avoids concurrency bugs, such as race conditions and deadlocks
- use Python packages written in non-Python languages to get around the GIL (global interpreter lock)

Outline

Critical Sections and Locks

Worksheet and Demos

Advanced Topics

- Global Interpreter Lock
- Instruction Reordering and Caching

Critical Sections

```
 1  # in dollars
 2  bank_accounts = {"x": 25, "y": 100, "z": 200}
 3
 4  def transfer_euros(src, dst, euros):
 5      dollars = euros_to_dollars(euros)
 6      success = False
 7
 8      if bank_accounts[src] >= dollars:
 9          bank_accounts[src] -= dollars
10          bank_accounts[dst] += dollars
11          success = True
12
13      print("transferred" if success else "denied")
```

If two threads are calling transfer_euros concurrently, *during which lines would a context switch between those two be problematic?*

A section of code we don't want interrupted by certain other code is a "critical section"

Critical Sections

```
# in dollars
bank_accounts = {"x": 25, "y": 100, "z": 200}

def transfer_euros(src, dst, euros):
    dollars = euros_to_dollars(euros)
    success = False

    if bank_accounts[src] >= dollars:
        bank_accounts[src] -= dollars
        bank_accounts[dst] += dollars
        success = True

    print("transferred" if success else "denied")
```

critical section

Goals:
Atomiticy: want withdrawal+deposit seen together (never seen half done).
Consistency: rules (called "invarants") like "no account goes negative" must be enforced

Locks

```
# in dollars
bank_accounts = {"x": 25, "y": 100, "z": 200}
lock = threading.Lock() # protects bank_accounts

def transfer_euros(src, dst, euros):
    lock.acquire()
    dollars = euros_to_dollars(euros)
    success = False
    if bank_accounts[src] >= dollars:
        bank_accounts[src] -= dollars
        bank_accounts[dst] += dollars
        success = True
    print("transferred" if success else "denied")
    lock.release()
```

Lock Rules

- between acquire and release, a lock is held by the thread that acquired it
- **a lock may only be held by one thread at a time**
- if T2 wants to acquire a lock held by T1, T2 blocks until T1 releases it

Locks

```
# in dollars
bank_accounts = {"x": 25, "y": 100, "z": 200}
lock = threading.Lock() # protects bank_accounts

def transfer_euros(src, dst, euros):
    dollars = euros_to_dollars(euros)
    success = False
    lock.acquire()
    if bank_accounts[src] >= dollars:
        bank_accounts[src] -= dollars
        bank_accounts[dst] += dollars
        success = True
    lock.release()
    print("transferred" if success else "denied")
```

Tradeoffs

- different patterns may accomplish the same goal
- some are more efficient; some are simpler

Locks

```python
# in dollars
bank_accounts = {"x": 25, "y": 100, "z": 200}
lock = threading.Lock() # protects bank_accounts

def transfer_euros(src, dst, euros):
    dollars = euros_to_dollars(euros)
    success = False
    if bank_accounts[src] >= dollars:
        lock.acquire()
        bank_accounts[src] -= dollars
        bank_accounts[dst] += dollars
        lock.release()
        success = True
    print("transferred" if success else "denied")
```

Tradeoffs

- different patterns may accomplish the same goal
- some are more efficient; some are simpler
- be careful! (this incorrect version provides atomicity but not consistency)

Worksheet and Demos...

Outline

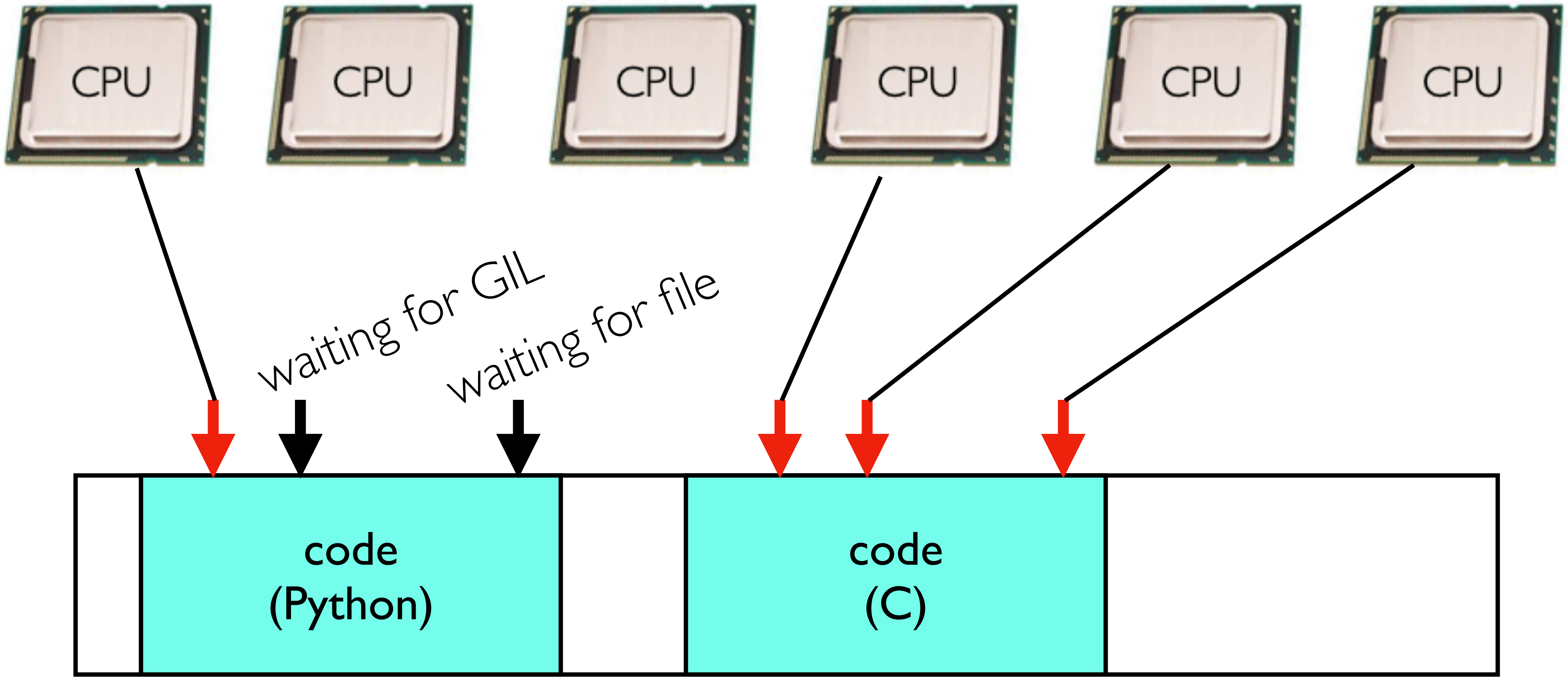
Critical Sections and Locks

Worksheet and Demos

Advanced Topics

- Global Interpreter Lock
- Instruction Reordering and Caching

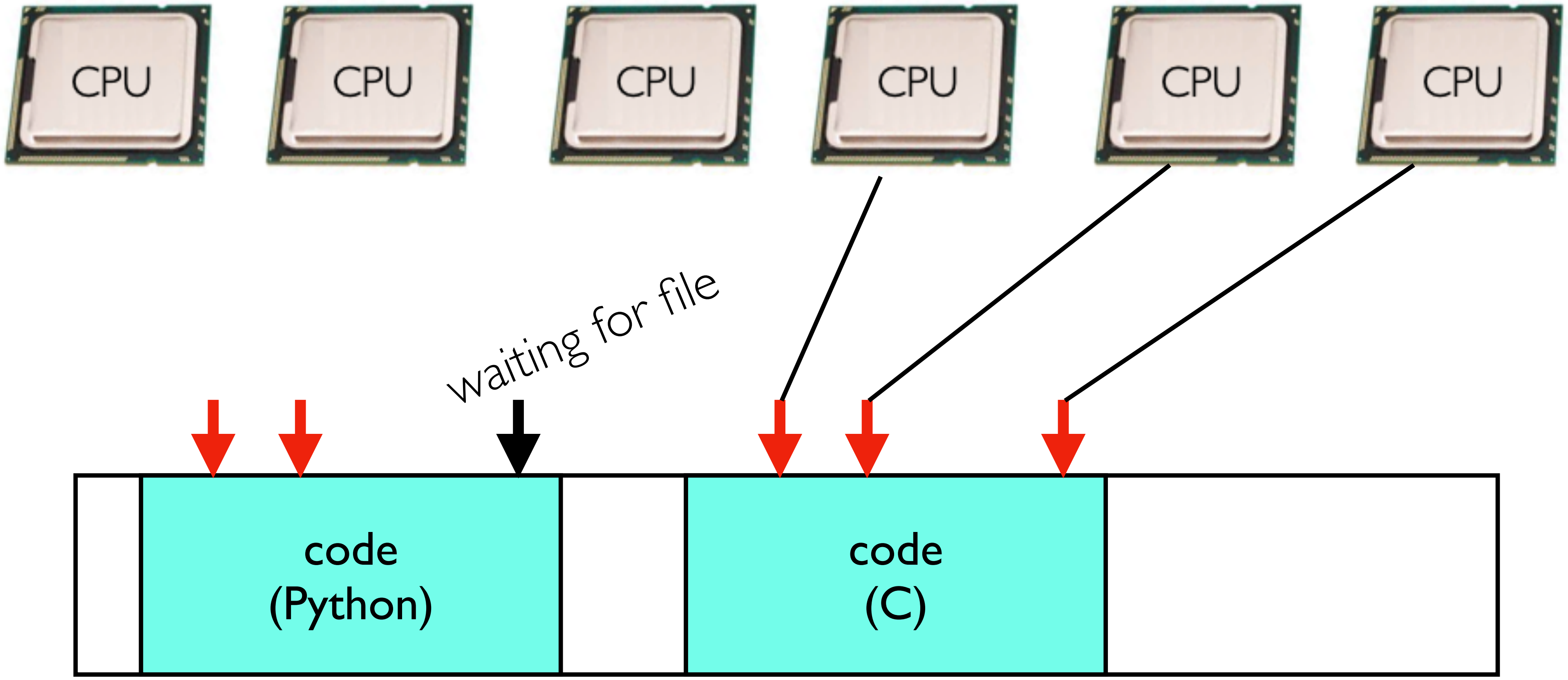
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

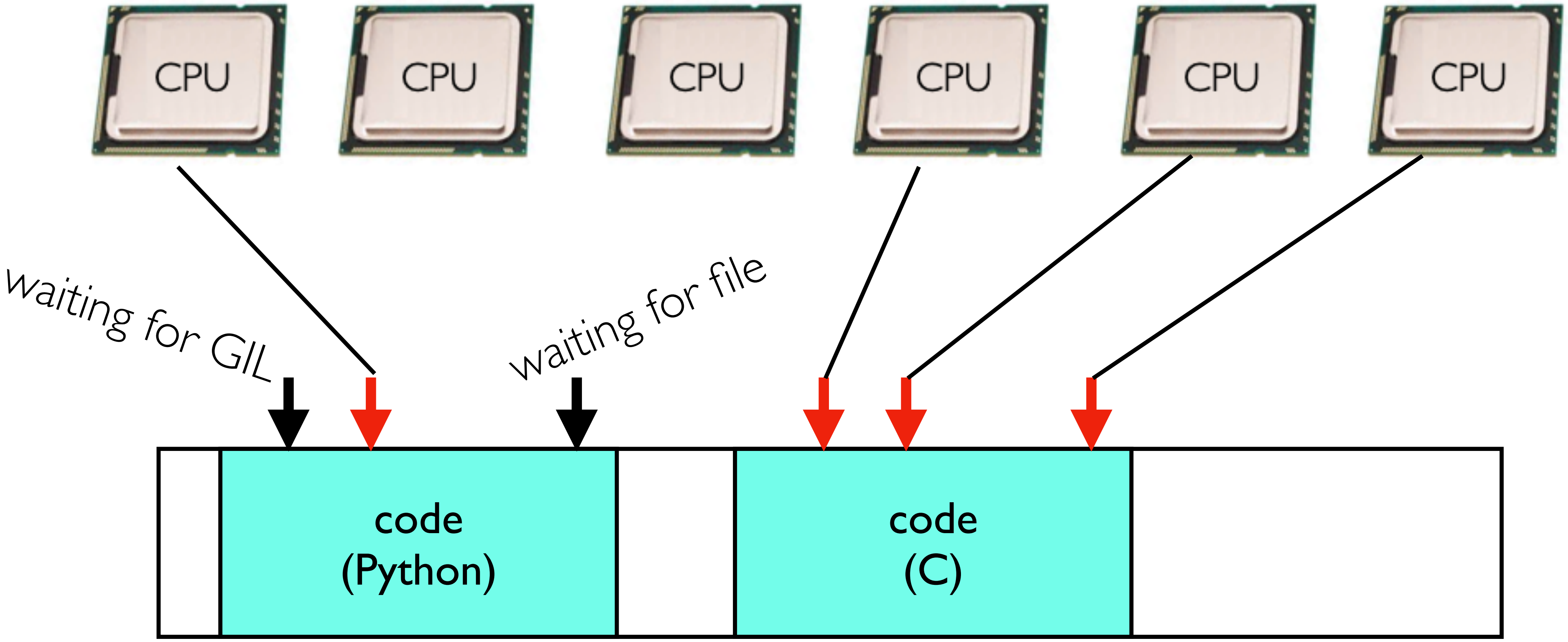
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

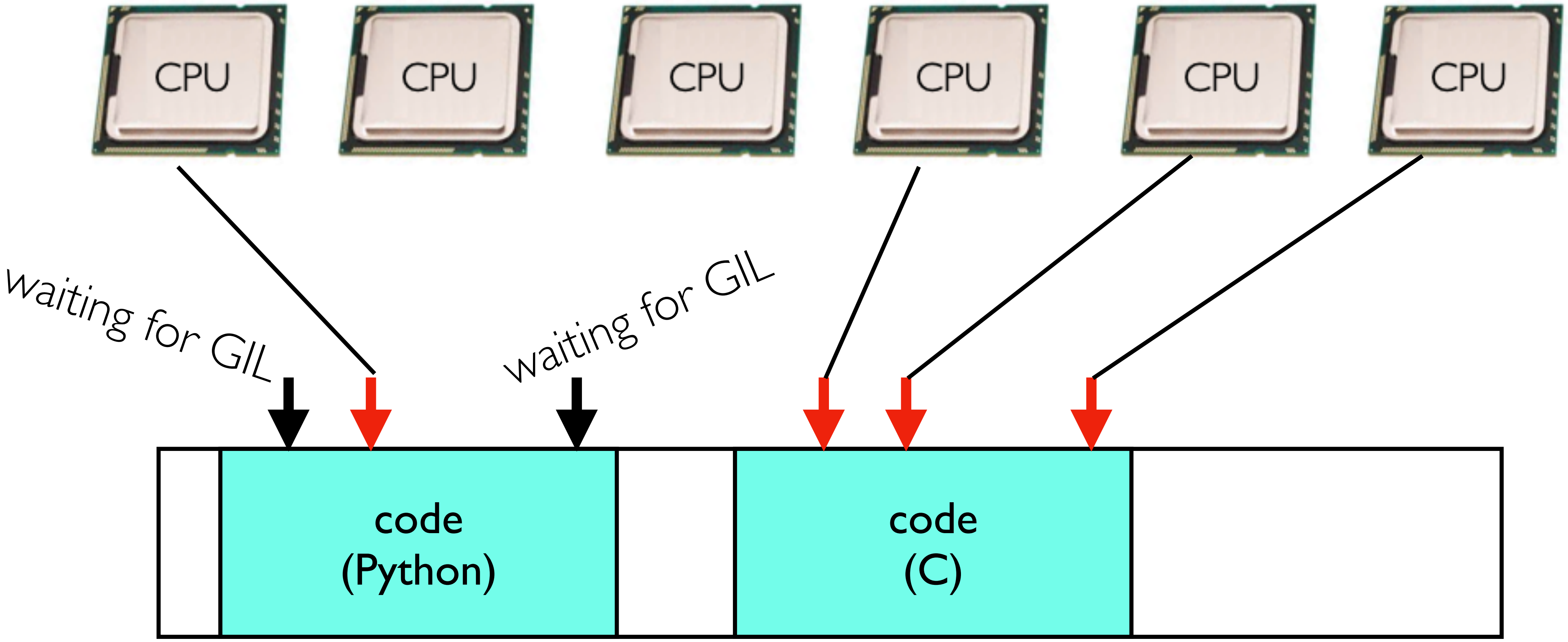
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

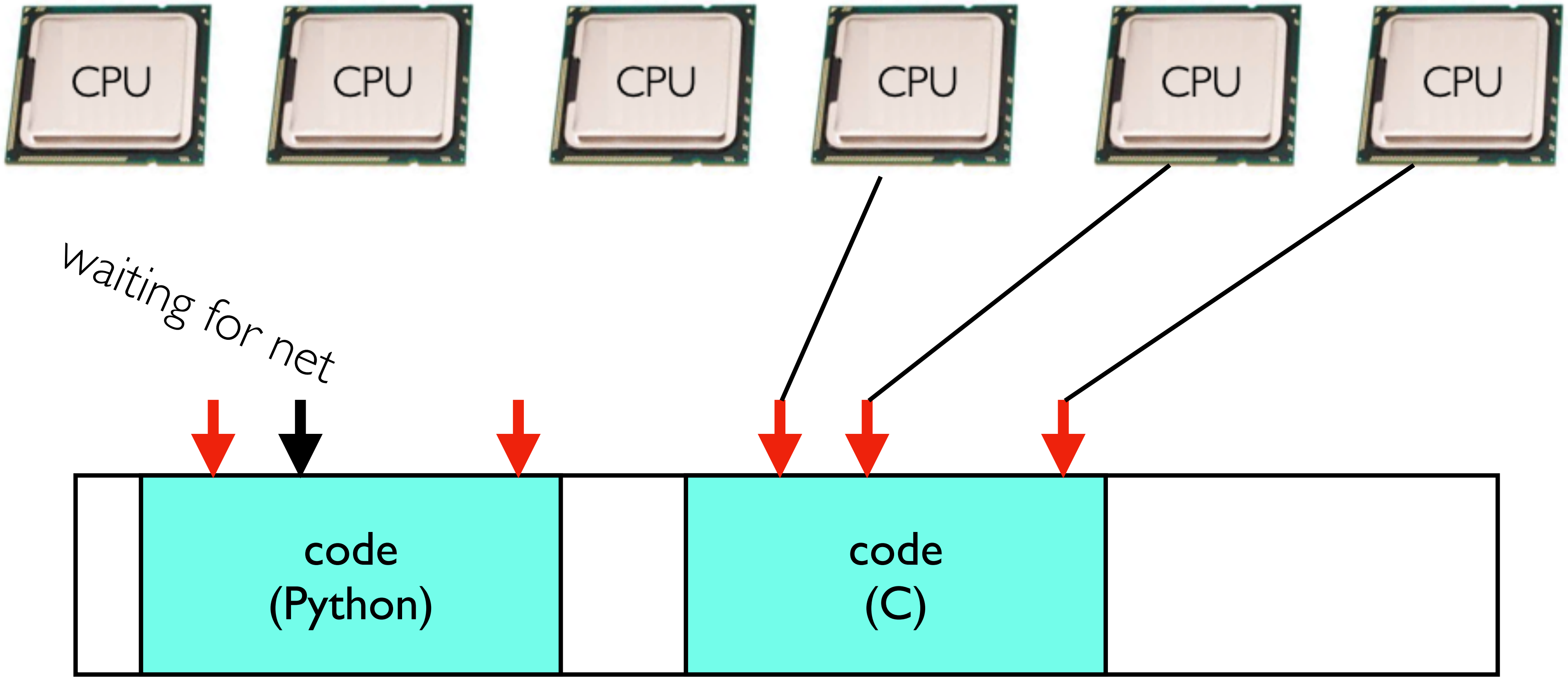
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

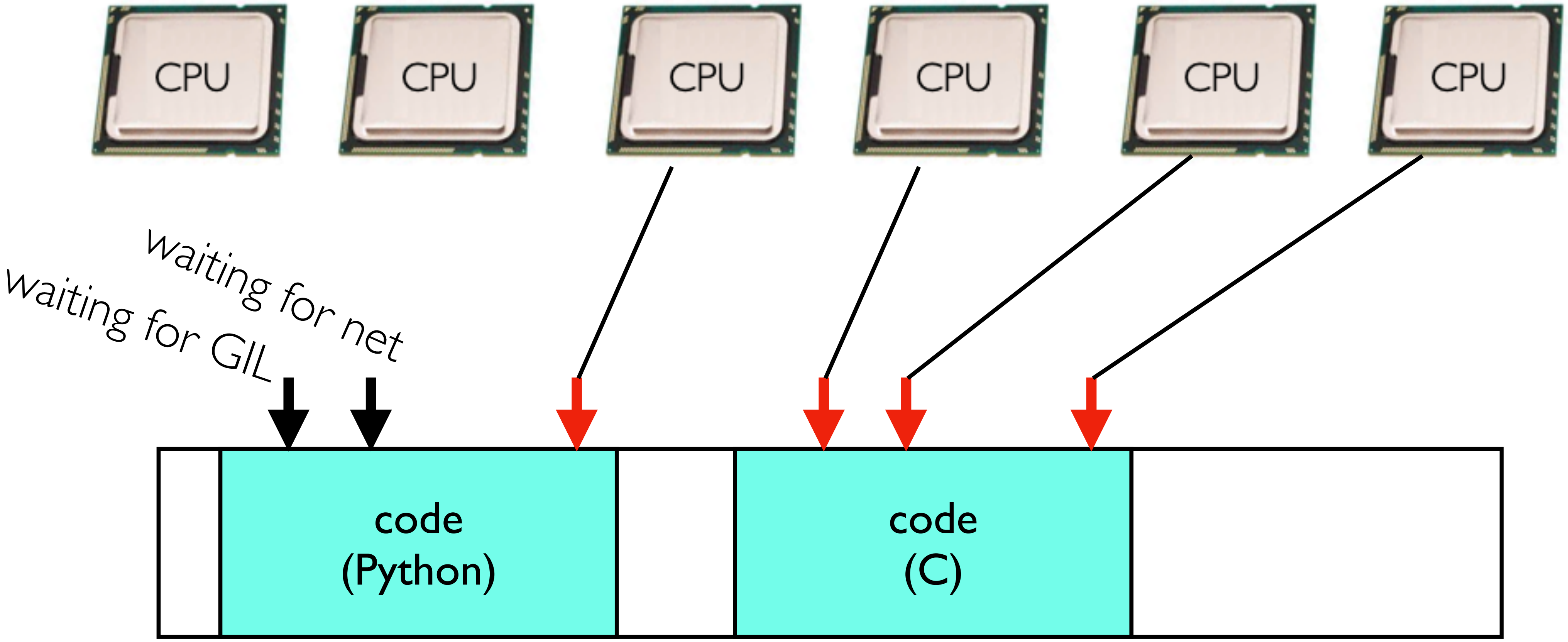
Python's GIL (Global Interpreter Lock)



Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

Python's GIL (Global Interpreter Lock)



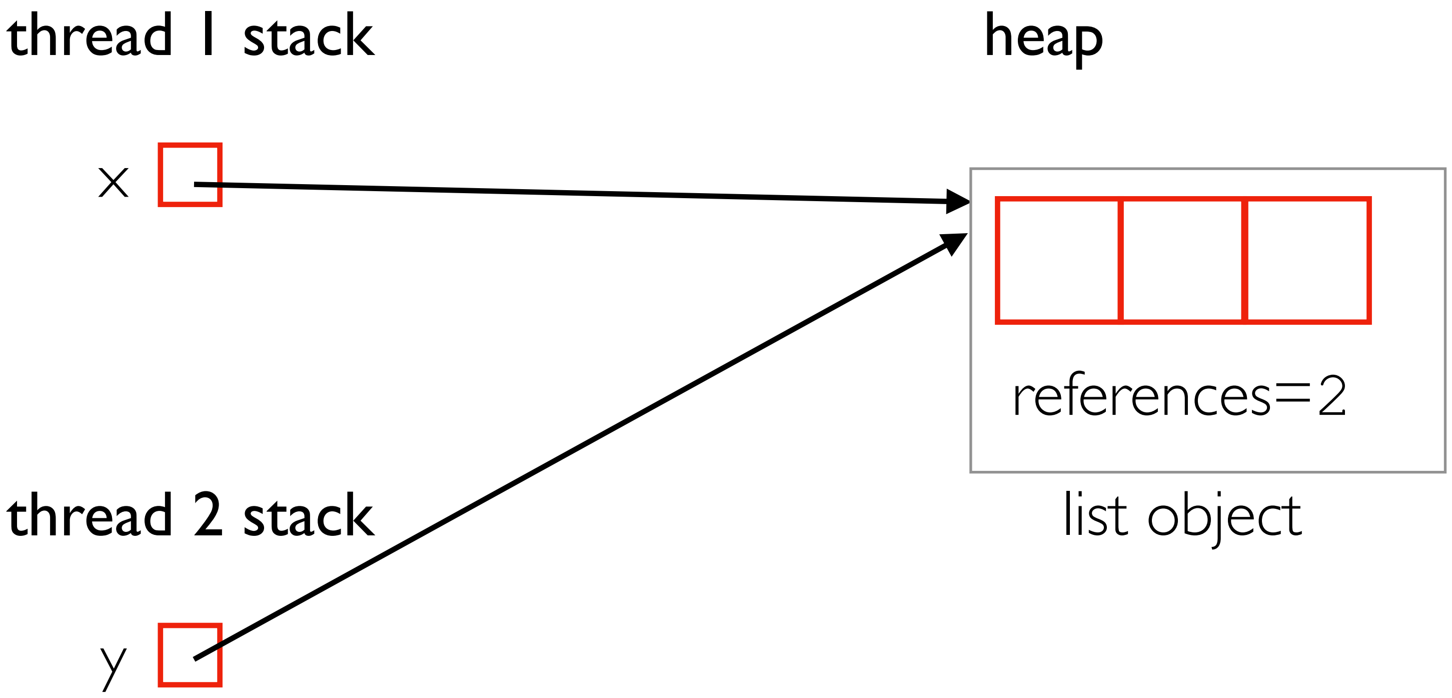
Global Interpreter Lock

- Only one thread can be running Python code in a process at once
- Python threads are bad for using multiple cores
- They're still useful for threads blocked on I/O
- Some Python libraries using other languages allow parallelism

Why the GIL?

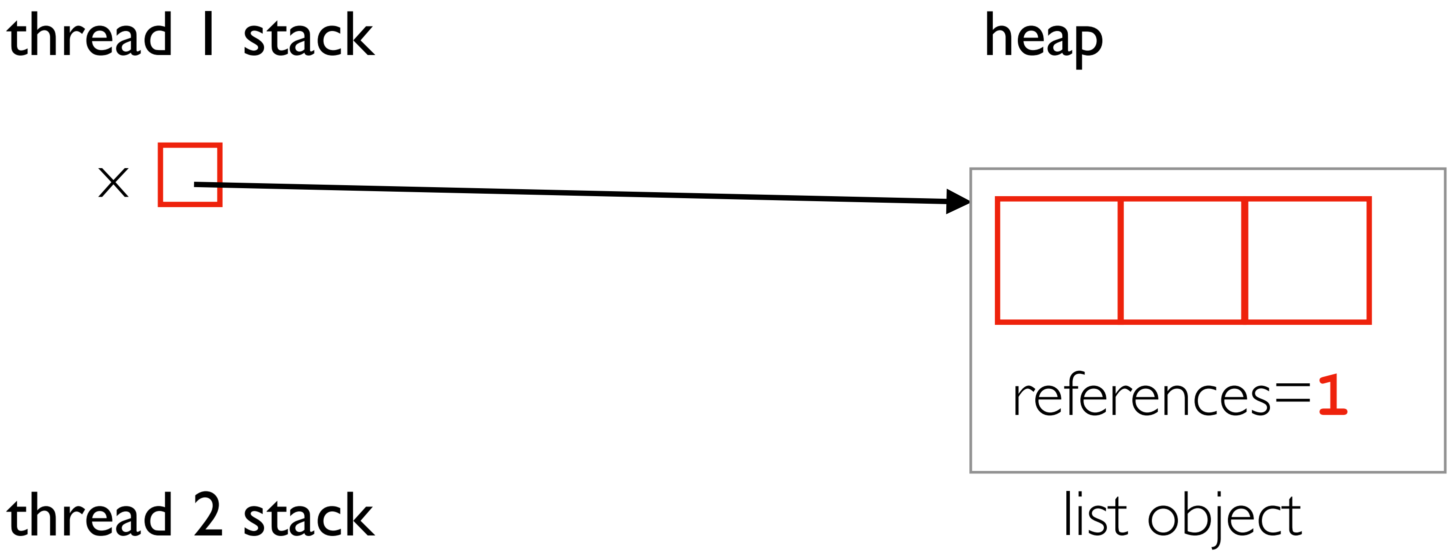
# thread 1
x = *some list*
x = *None*

# thread 2
y = *that same list*
y = None



Why the GIL?

```
# thread 1
x = some list
x = None

# thread 2
y = that same list
y = None
```



object will be freed when references is 0

Why the GIL?

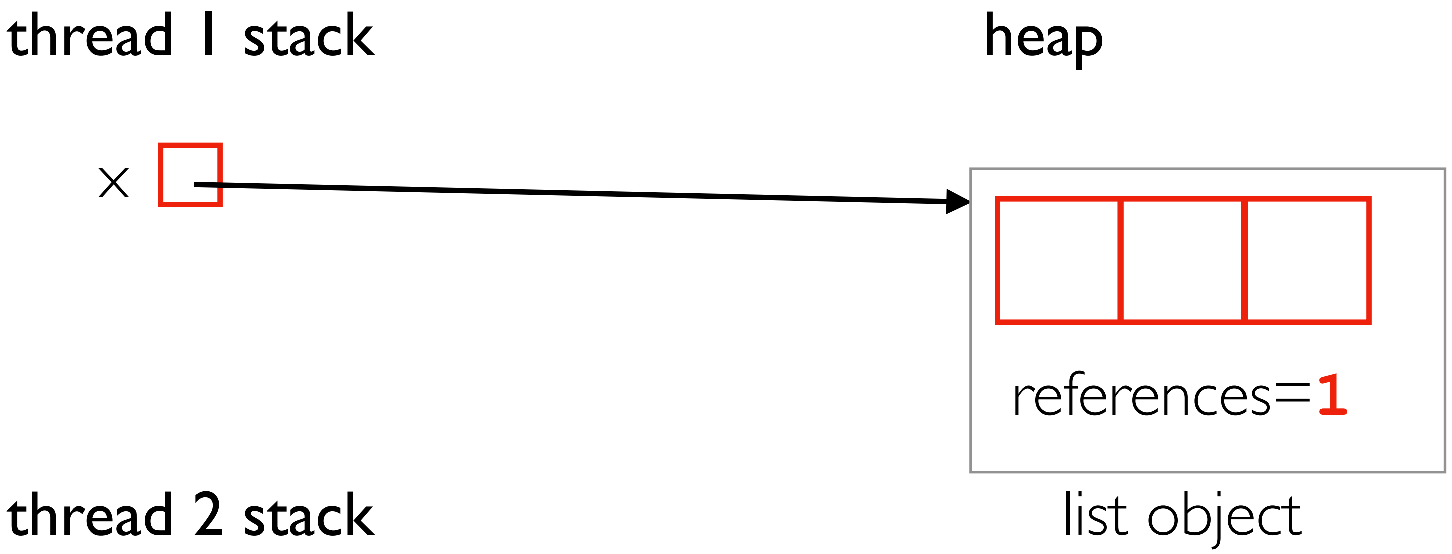
```
# thread 1
x = some list
x = None

# thread 2
y = that same list
y = None
```

situation

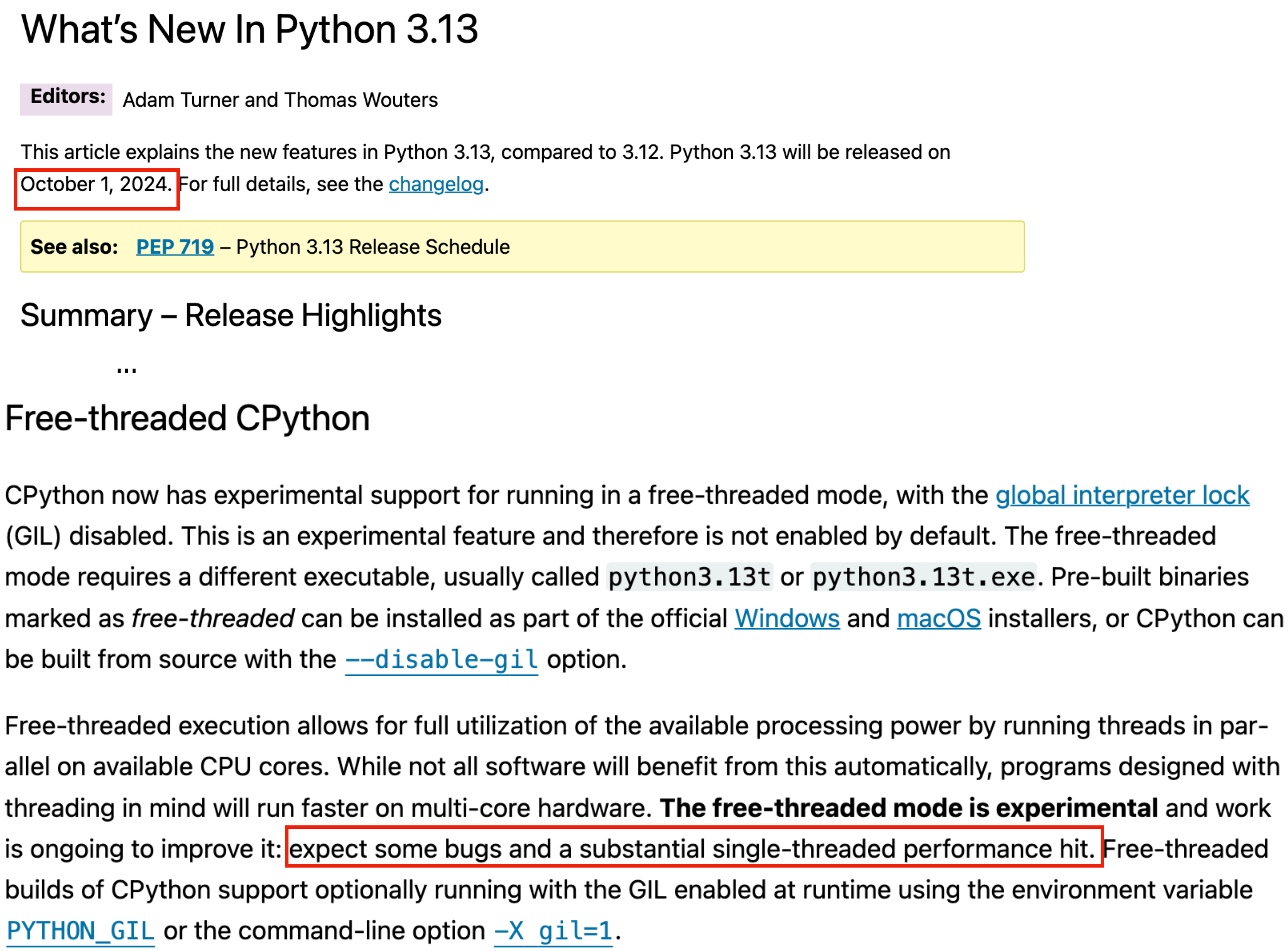
- cpython (main Python interpreter) uses reference counting internally to know when it can free objects
- implication: multiple threads modifying same integer

solutions

- run one thread at a time (Python's approach)
- lots of locking (slower for single-threaded code)
- *other?*



Future of GIL



What’s New In Python 3.13

Editors: Adam Turner and Thomas Wouters

This article explains the new features in Python 3.13, compared to 3.12. Python 3.13 will be released on October 1, 2024. For full details, see the changelog.

See also: PEP 719 – Python 3.13 Release Schedule

Summary – Release Highlights

...

Free-threaded CPython

CPython now has experimental support for running in a free-threaded mode, with the global interpreter lock (GIL) disabled. This is an experimental feature and therefore is not enabled by default. The free-threaded mode requires a different executable, usually called `python3.13t` or `python3.13t.exe`. Pre-built binaries marked as *free-threaded* can be installed as part of the official Windows and macOS installers, or CPython can be built from source with the `--disable-gil` option.

Free-threaded execution allows for full utilization of the available processing power by running threads in parallel on available CPU cores. While not all software will benefit from this automatically, programs designed with threading in mind will run faster on multi-core hardware. **The free-threaded mode is experimental** and work is ongoing to improve it: expect some bugs and a substantial single-threaded performance hit. Free-threaded builds of CPython support optionally running with the GIL enabled at runtime using the environment variable `PYTHON_GIL` or the command-line option `-X gil=1`.

https://docs.python.org/3.13/whatsnew/3.13.html

Outline

Critical Sections and Locks

Worksheet and Demos

Advanced Topics

- Global Interpreter Lock
- Instruction Reordering and Caching

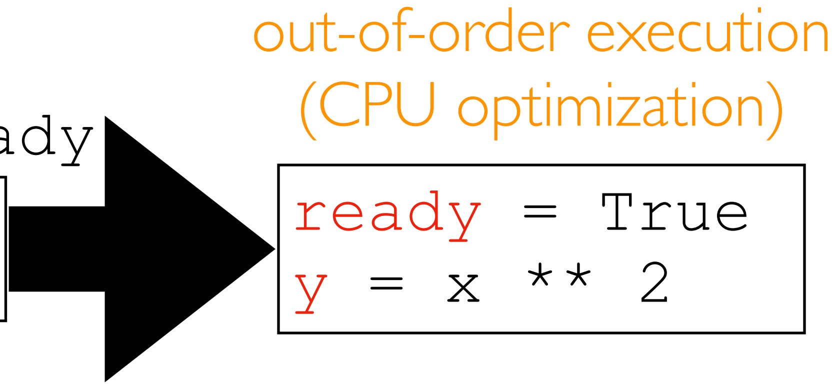
Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```



out-of-order execution
(CPU optimization)

```
ready = True
y = x ** 2
```

no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

main

core 1 (running task)

L1 cache:
y = 25
ready = True

core 2 (running main)

L1 cache:
y = 0 (stale)
ready = False (stale)

no interleaving is problematic, but it's still not correct on a modern CPU!

Challenges Beyond Interleaving

```
import threading

y = 0
ready = False

def task(x):
    global y, ready
    y = x ** 2
    ready = True

t = threading.Thread(target=task, args=[5])
t.start()
while not ready:
    pass
print(y) # want 25 (not 0)
```

main

core 1 (running task)

L1 cache:
y = 25
ready = True

core 2 (running main)

L1 cache:
y = 0 (stale)
ready = True

no interleaving is problematic, but it's still not correct on a modern CPU!

Concluding Advice

Use provided primitives (like locks+joins) to control isolation+ordering

- these calls control interleavings AND memory barriers (topic beyond 544)
- it's easy to get lockless approaches wrong

Correctness tips (keep it simple to avoid bugs!):

- can you use multiple processes instead of threads?
- is one big lock good enough for protecting all your data?
- is it OK to hold the lock through a whole function call?

Performance tips:

- avoid holding a lock while blocking on I/O (network, disk, user input, etc)
- if you have multiple updates, can you hold the lock for more than one of them?
- use performant packages like numpy
  - ➡ the code in C/C++/Fortran/Rust can often run without the GIL
  - ➡ these will often create threads for you