to GIL or not to GIL: the Future of Multi-Core (C)Python

Eric Snow
PyCon 2019

https://bit.ly/2UMMJey

@ericsnowcrntly

Who Am I?

- software engineer at Microsoft (Python extension for VS Code)
- CPython core developer (since 2012)
    - 8 PEPs (5 accepted, 3 open)
    - sys.implementation
    - module.__spec__
    - C OrderedDict

https://bit.ly/2UMMJey

Who Am I?

- software engineer at Microsoft (Python extension for VS Code)
- CPython core developer (since 2012)
    - 8 PEPs (5 accepted, 3 open)
    - sys.implementation
    - module.__spec__
    - C OrderedDict
- tired of hearing about how the GIL makes Python awful
- in late 2014 decided to do something about it

https://bit.ly/2UMMJey

Overview

1. Context
    - CPython’s Architecture
    - What happens when Python Runs?
    - Threads and Locks
2. The GIL
3. The Future
    - The C-API
    - Subinterpreters!
4. Q&A

https://bit.ly/2UMMJey

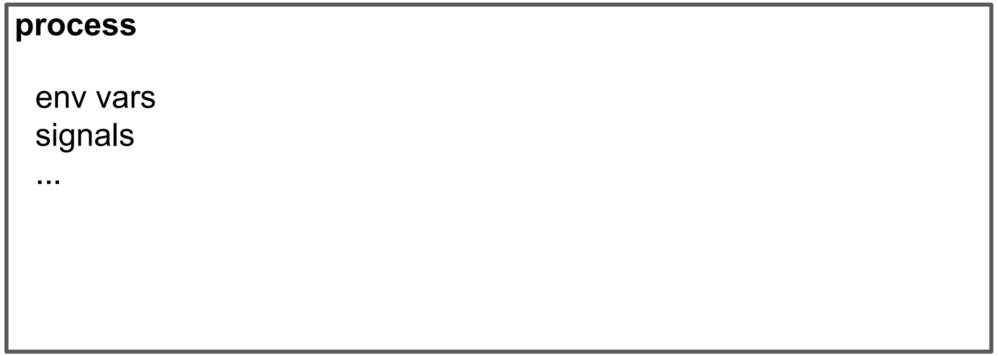
Context

An Overview of CPython’s Architecture

● process	- the OS process
● runtime	- everything Python-related in a process
● interpreter	- all Python threads and everything they share
● Python thread	- wrapper around OS thread with eval loop inside
● call stack	- stack of eval loop instances (i.e. Python function calls)
● eval loop	- executes the sequence of instructions in a code obj

https://bit.ly/2UMMJey

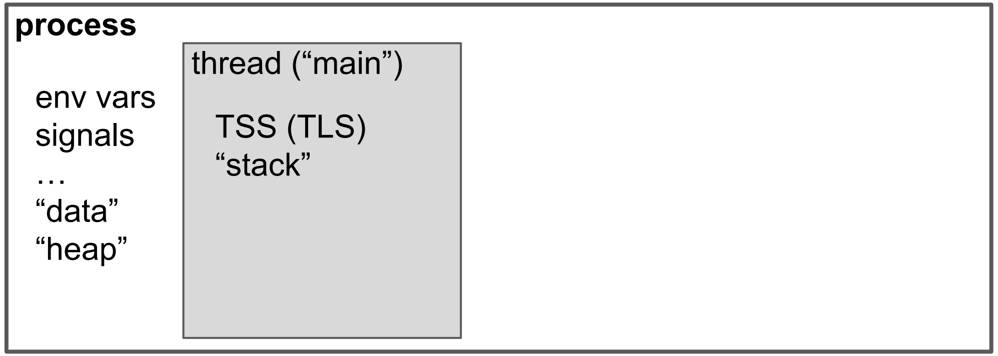
What Happens When Python Runs?



1. process initializes

https://bit.ly/2UMMJey

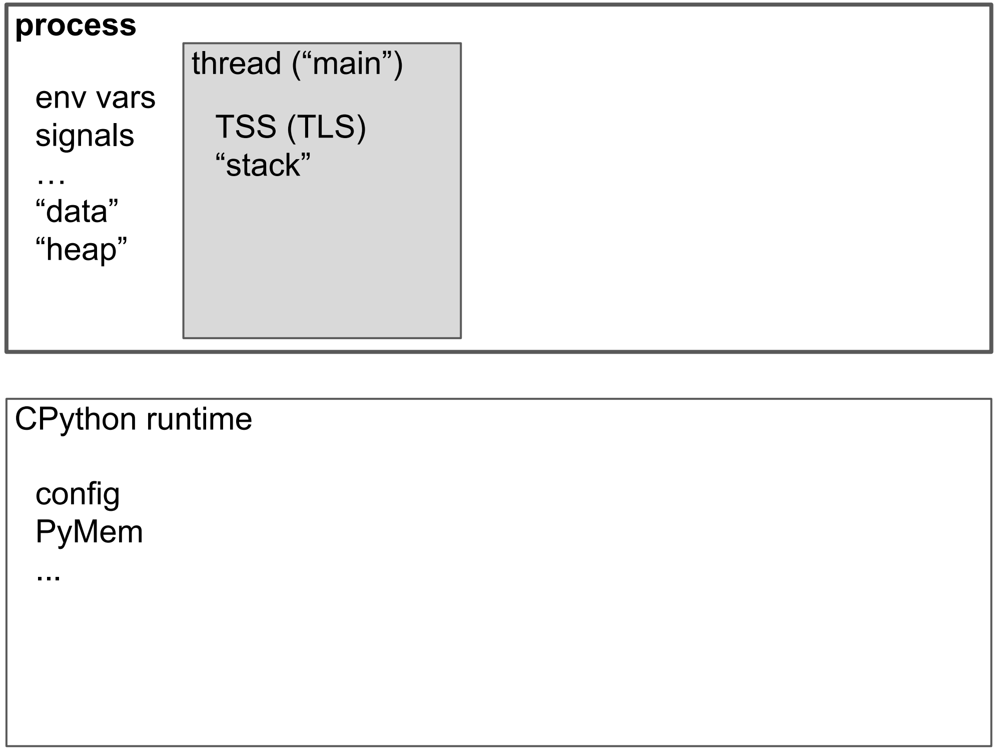
What Happens When Python Runs?



1. process initializes
2. main thread starts

https://bit.ly/2UMMJey

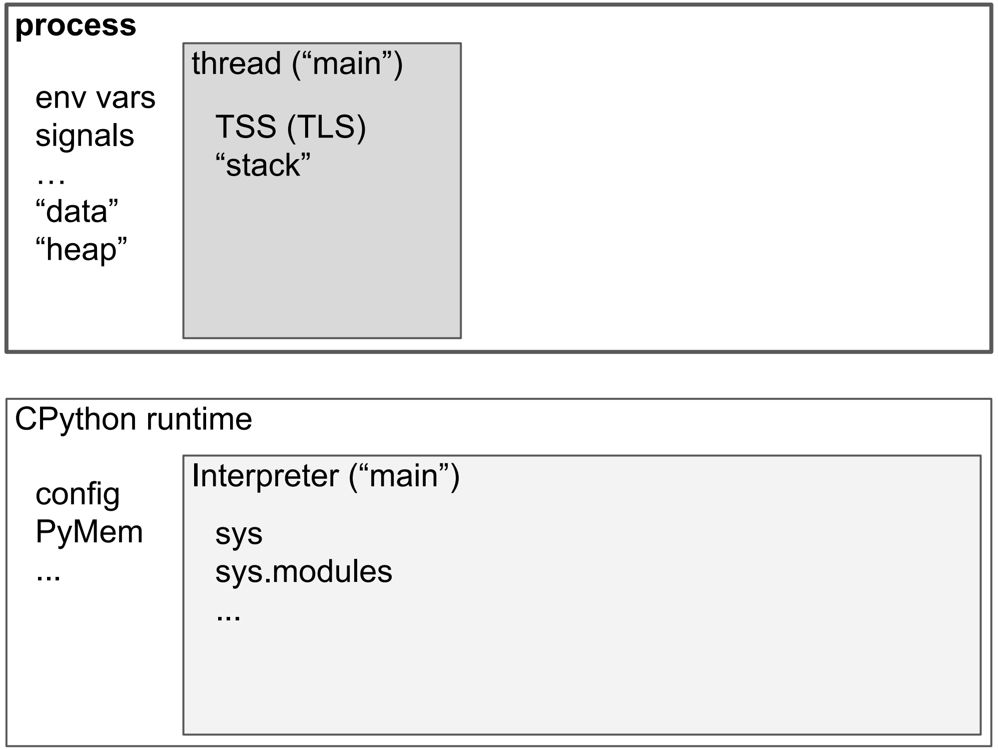
What Happens When Python Runs?



1. process initializes
2. main thread starts
3. Python runtime initializes

https://bit.ly/2UMMJey

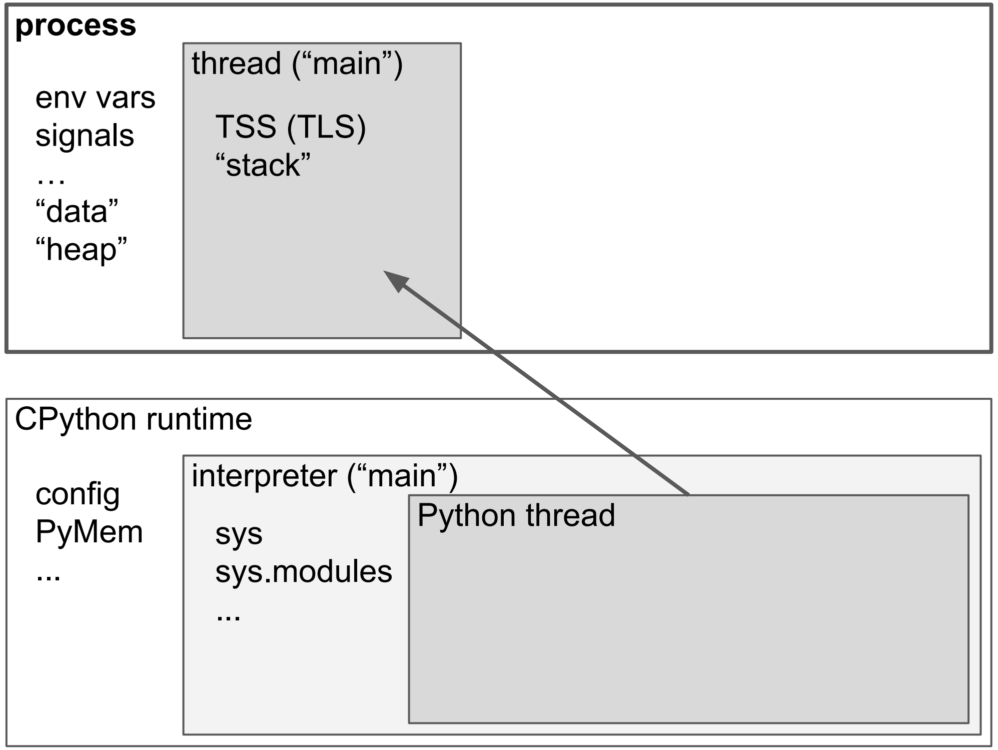
What Happens When Python Runs?



1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes

https://bit.ly/2UMMJey

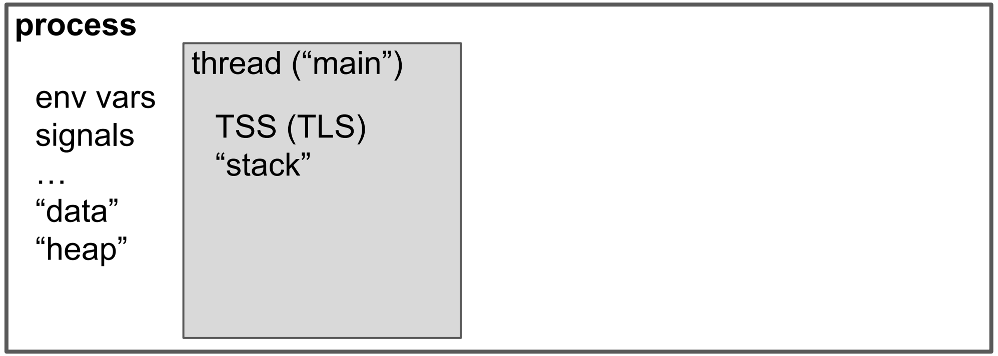
What Happens When Python Runs?



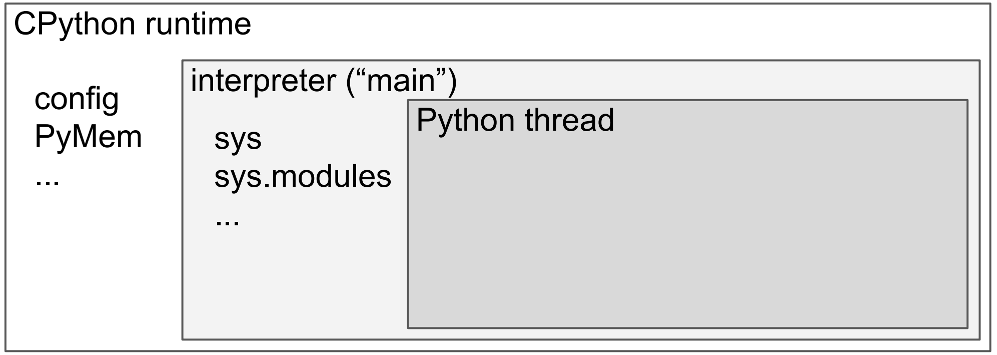
1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes

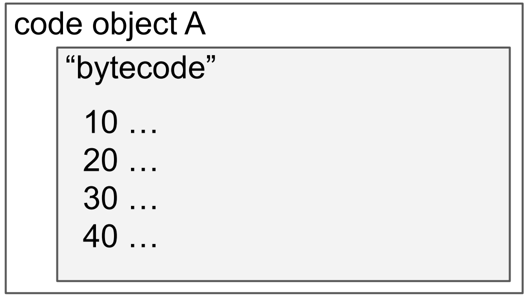
https://bit.ly/2UMMJey

What Happens When Python Runs?



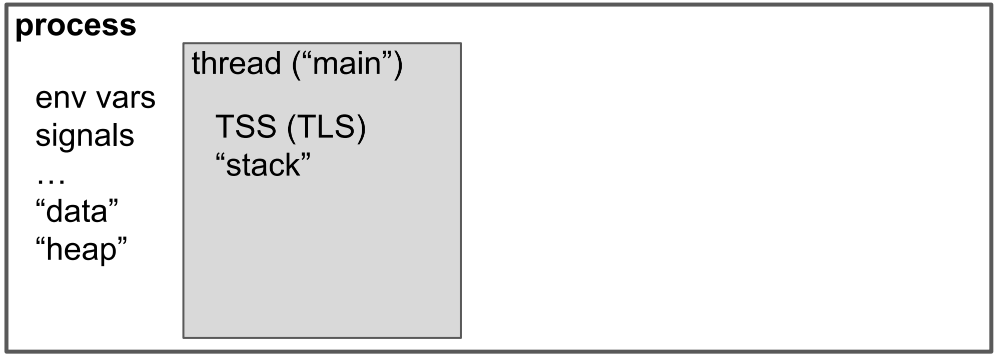
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads



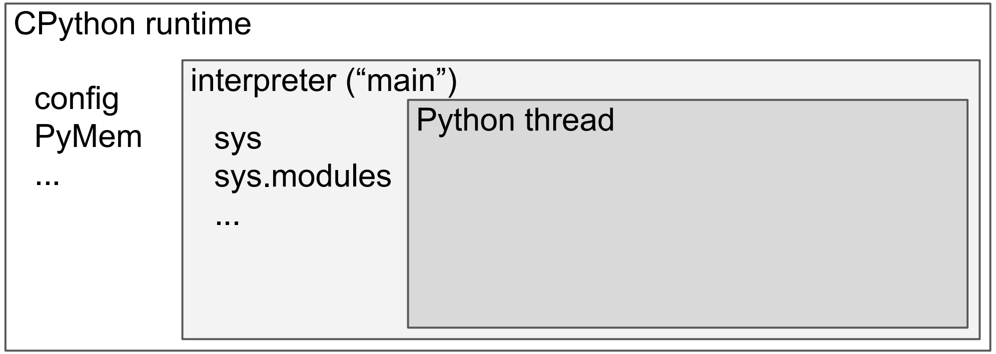


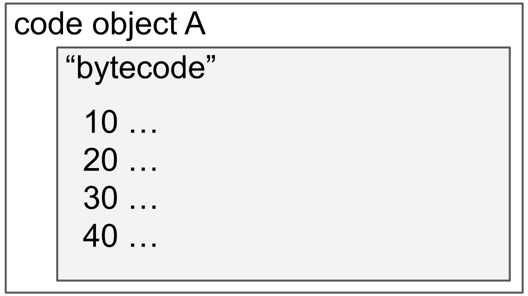
https://bit.ly/2UMMJey

What Happens When Python Runs?



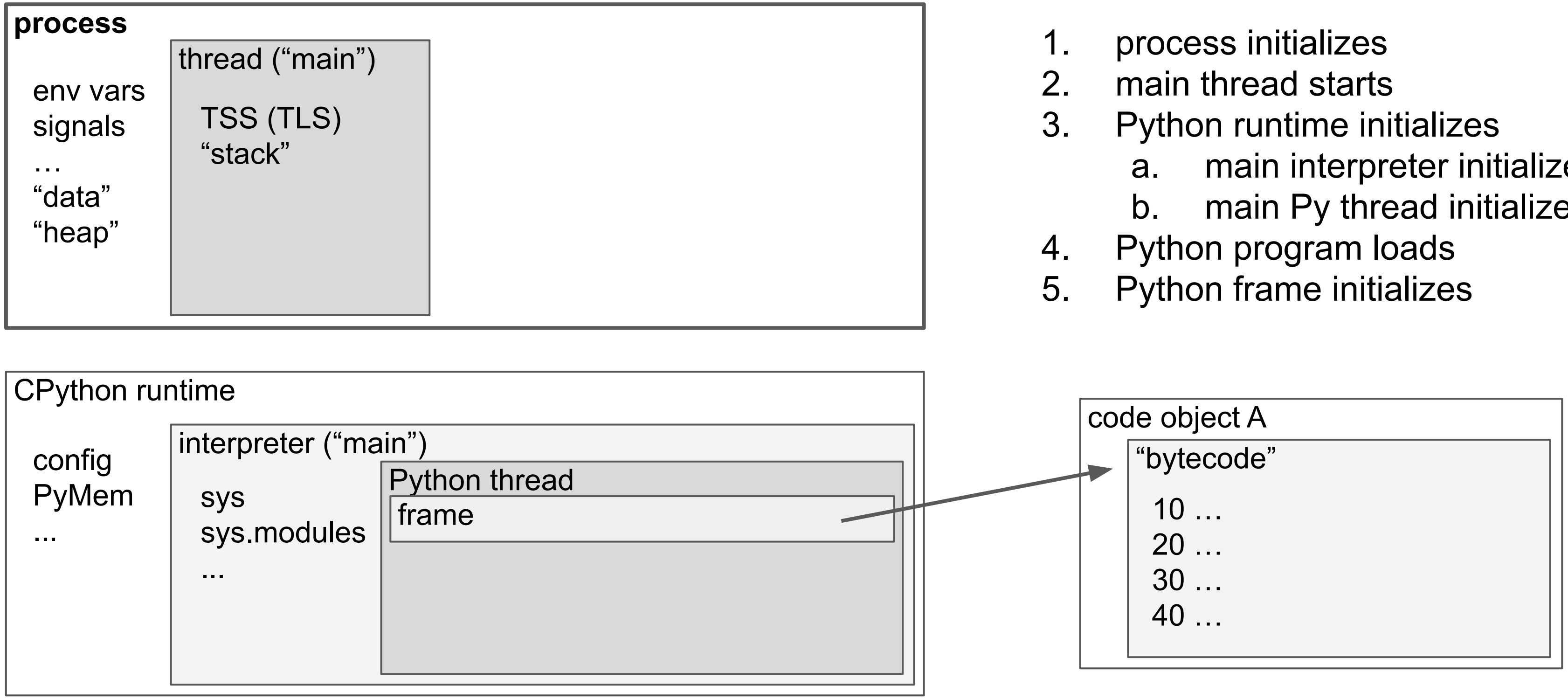
1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads





https://bit.ly/2UMMJey

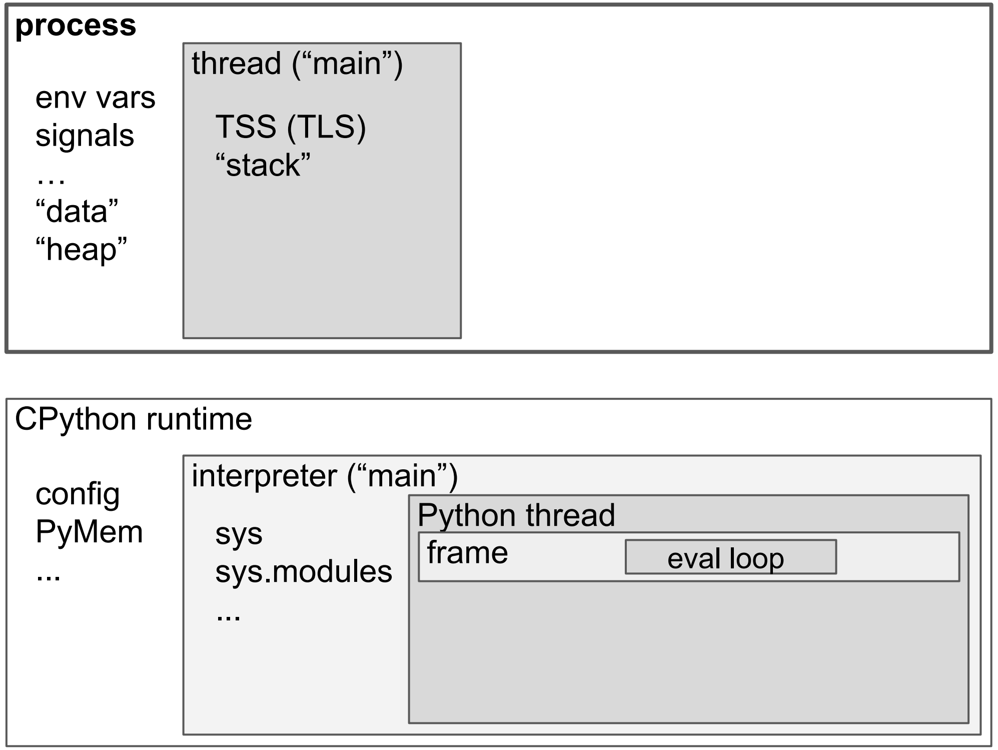
What Happens When Python Runs?



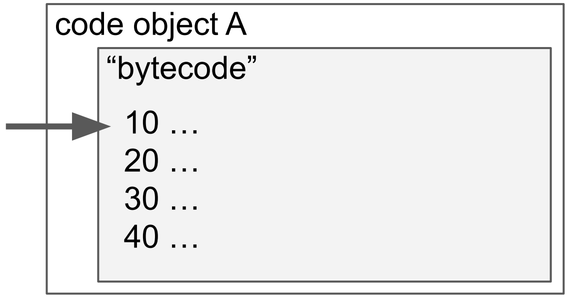
1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads
5. Python frame initializes

https://bit.ly/2UMMJey

What Happens When Python Runs?



1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



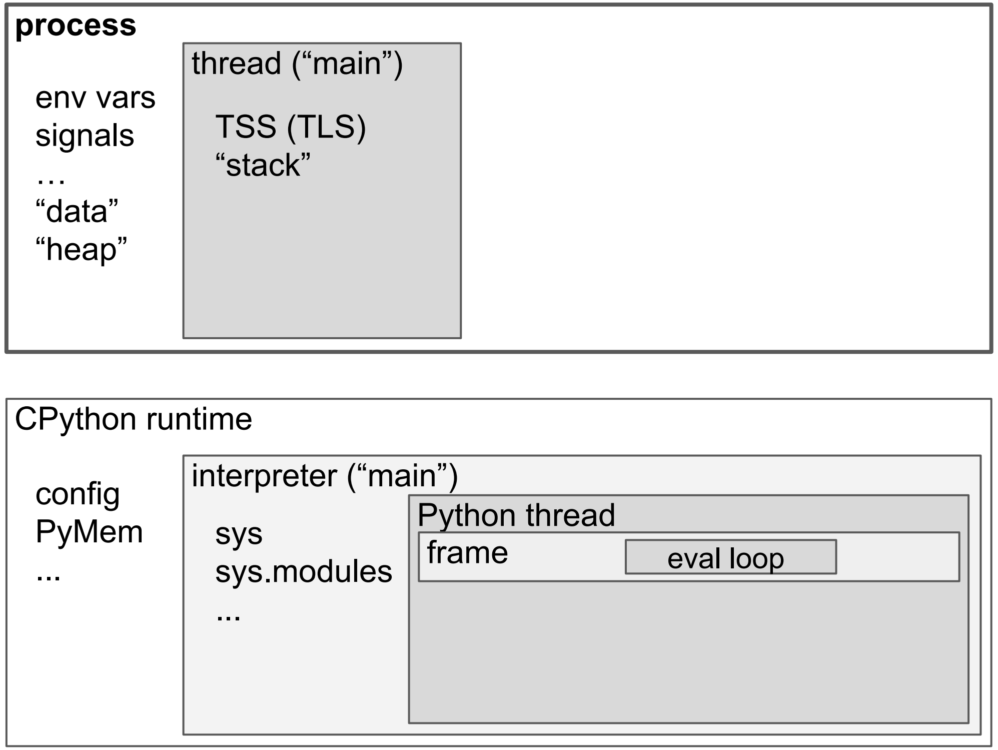
https://bit.ly/2UMMJey

The Eval Loop

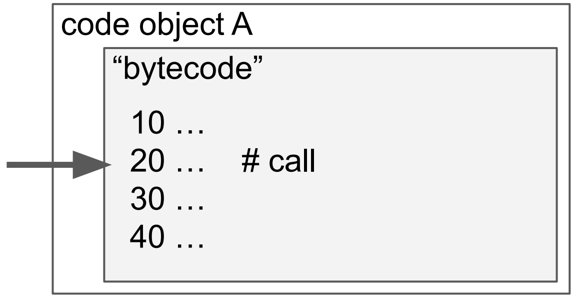
```
<set up>

for instruction in code object:

    <maybe side-channel stuff>

    <occasionally release & re-acquire the GIL>

    <execute next instruction>
```

https://bit.ly/2UMMJey

What Happens When Python Runs?

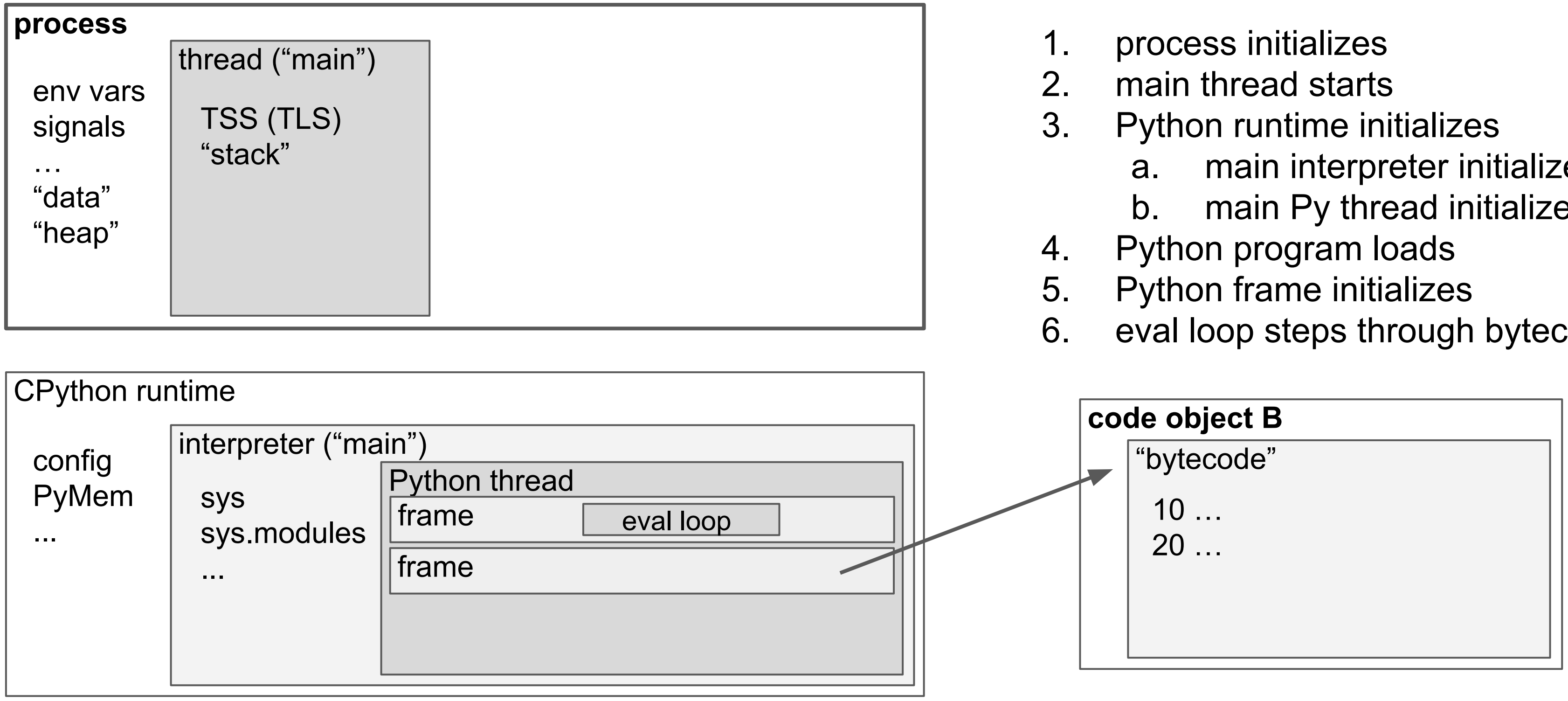


1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



https://bit.ly/2UMMJey

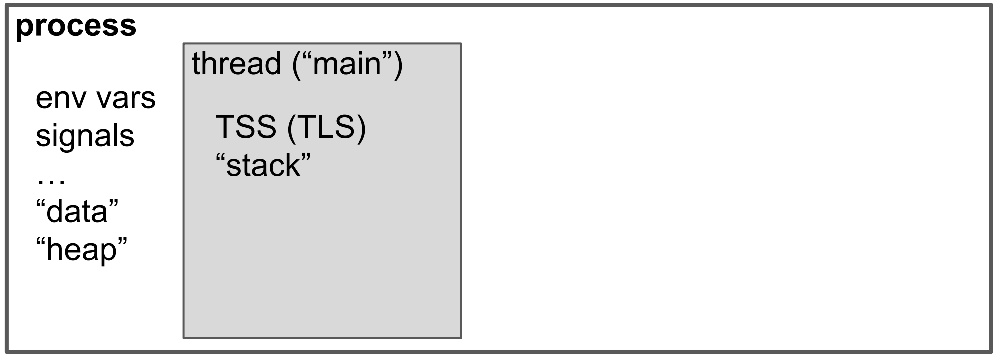
What Happens When Python Runs?



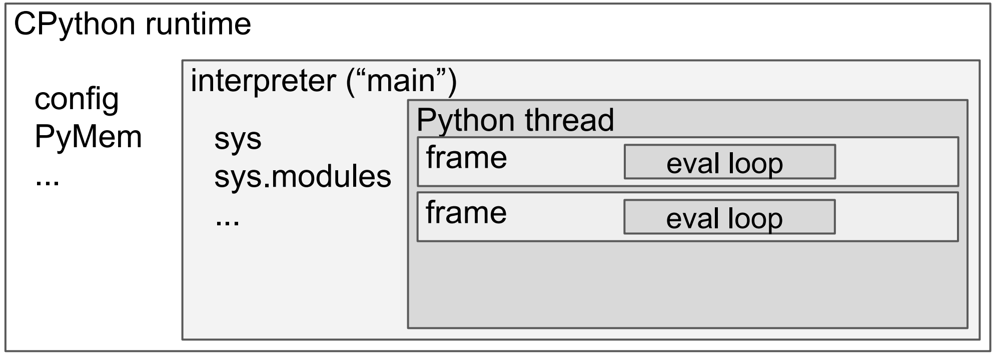
1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

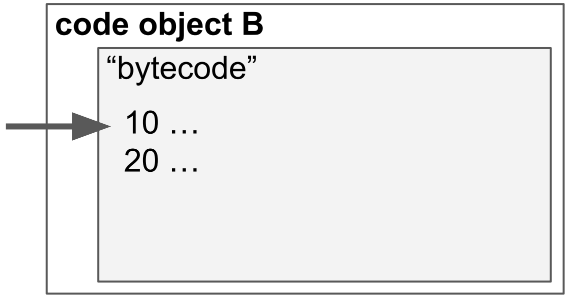
https://bit.ly/2UMMJey

What Happens When Python Runs?



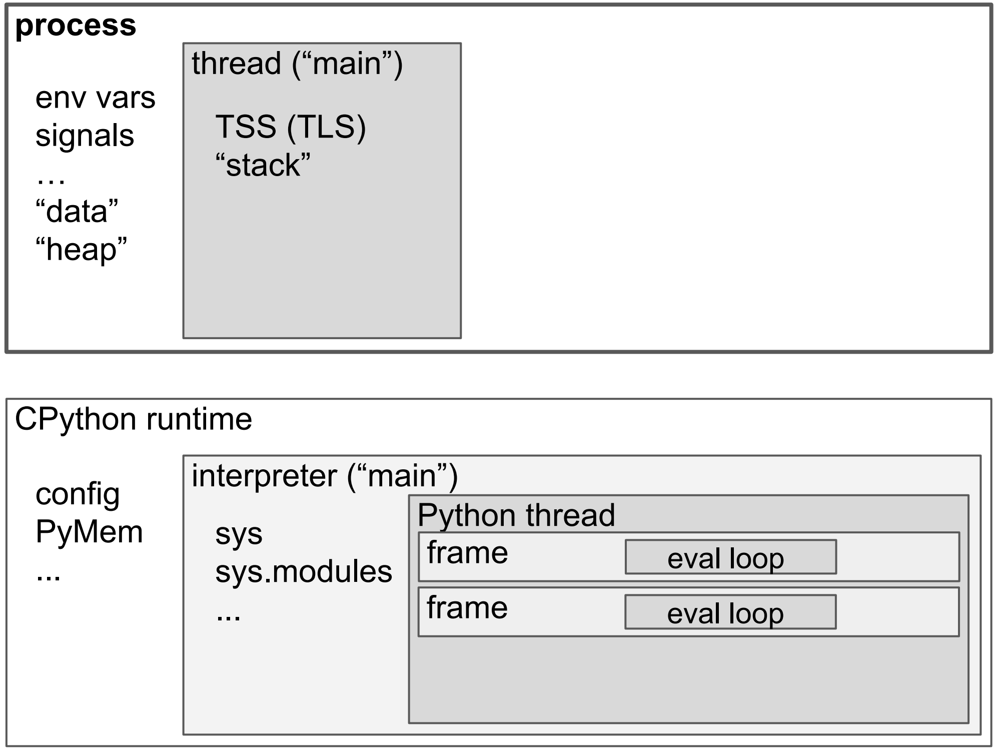
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



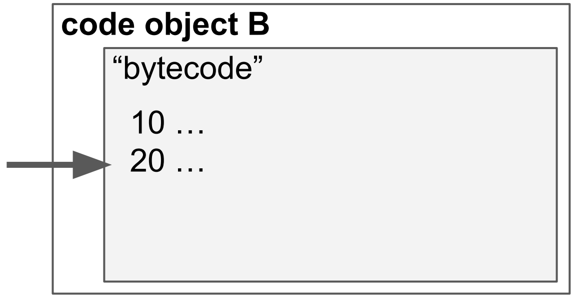


https://bit.ly/2UMMJey

What Happens When Python Runs?

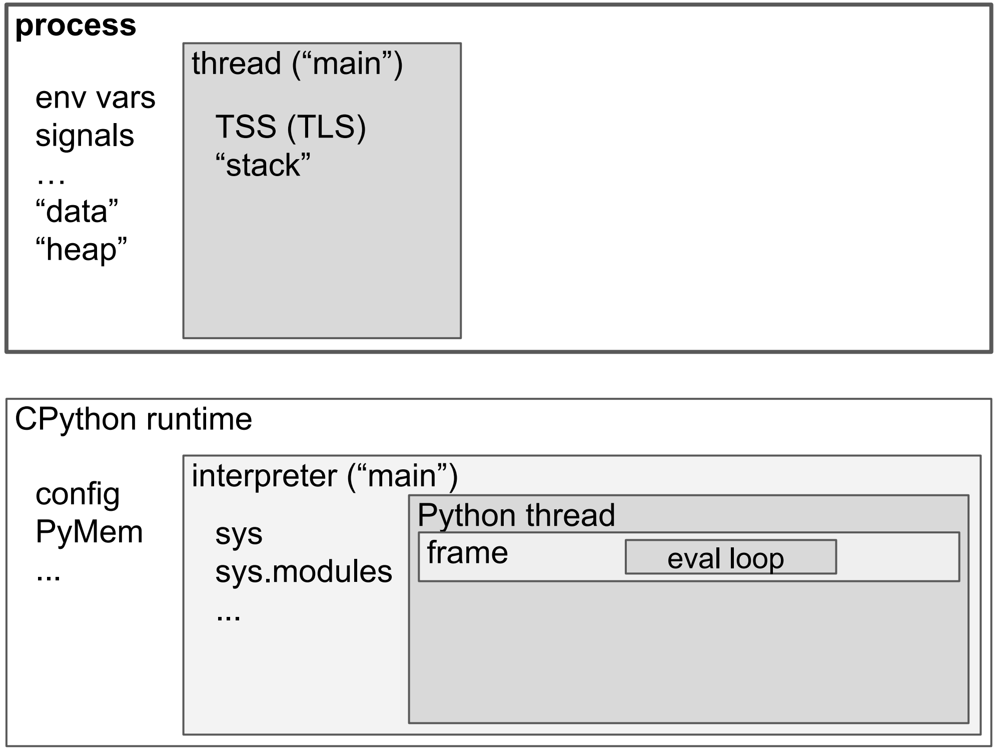


1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

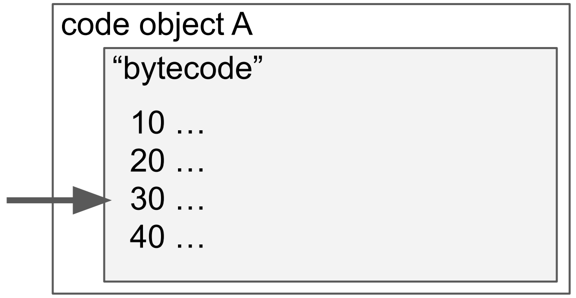


https://bit.ly/2UMMJey

What Happens When Python Runs?

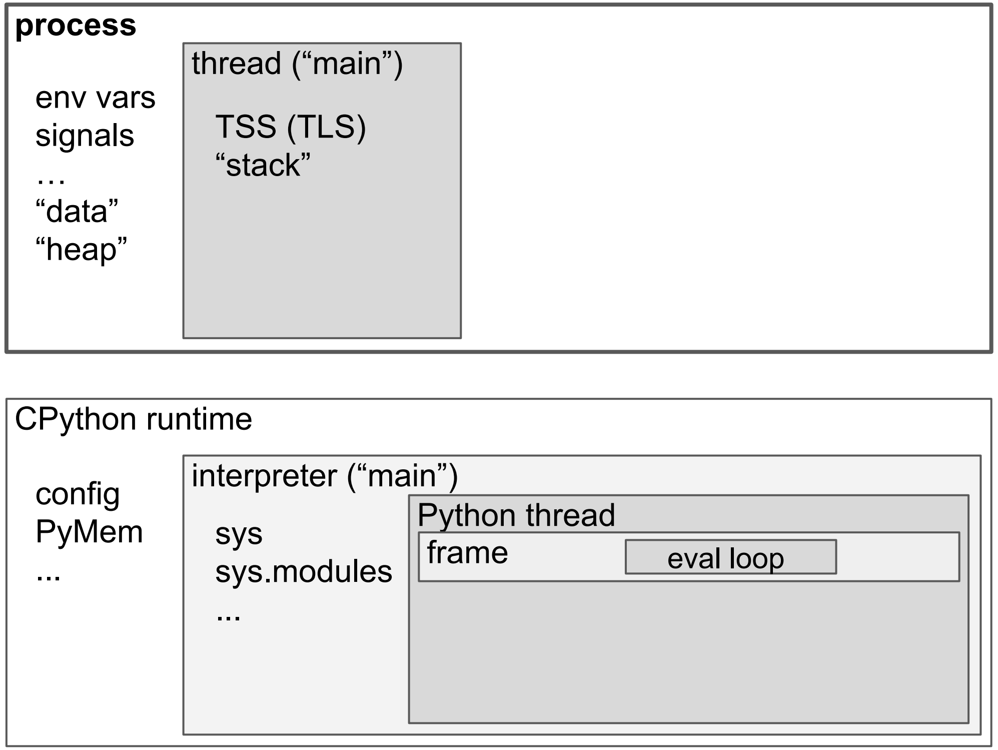


1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

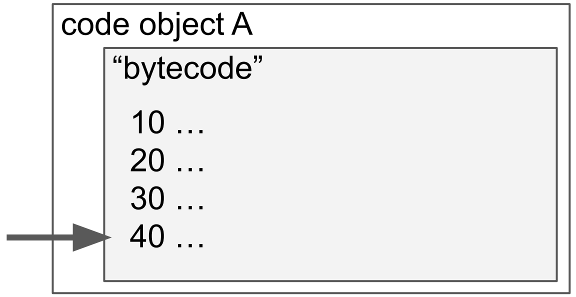


https://bit.ly/2UMMJey

What Happens When Python Runs?

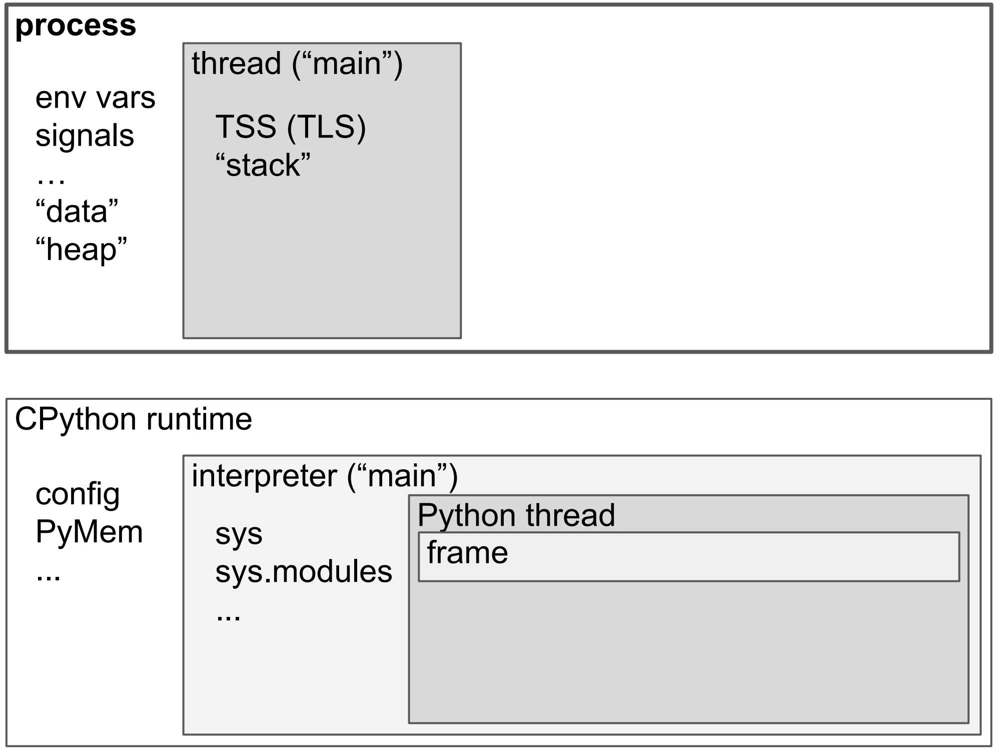


1. process initializes
2. main thread starts
3. Python runtime initializes
    a. main interpreter initializes
    b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

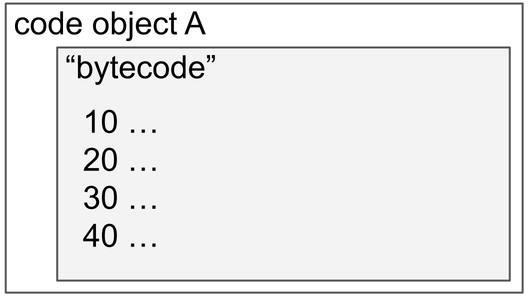


https://bit.ly/2UMMJey

What Happens When Python Runs?

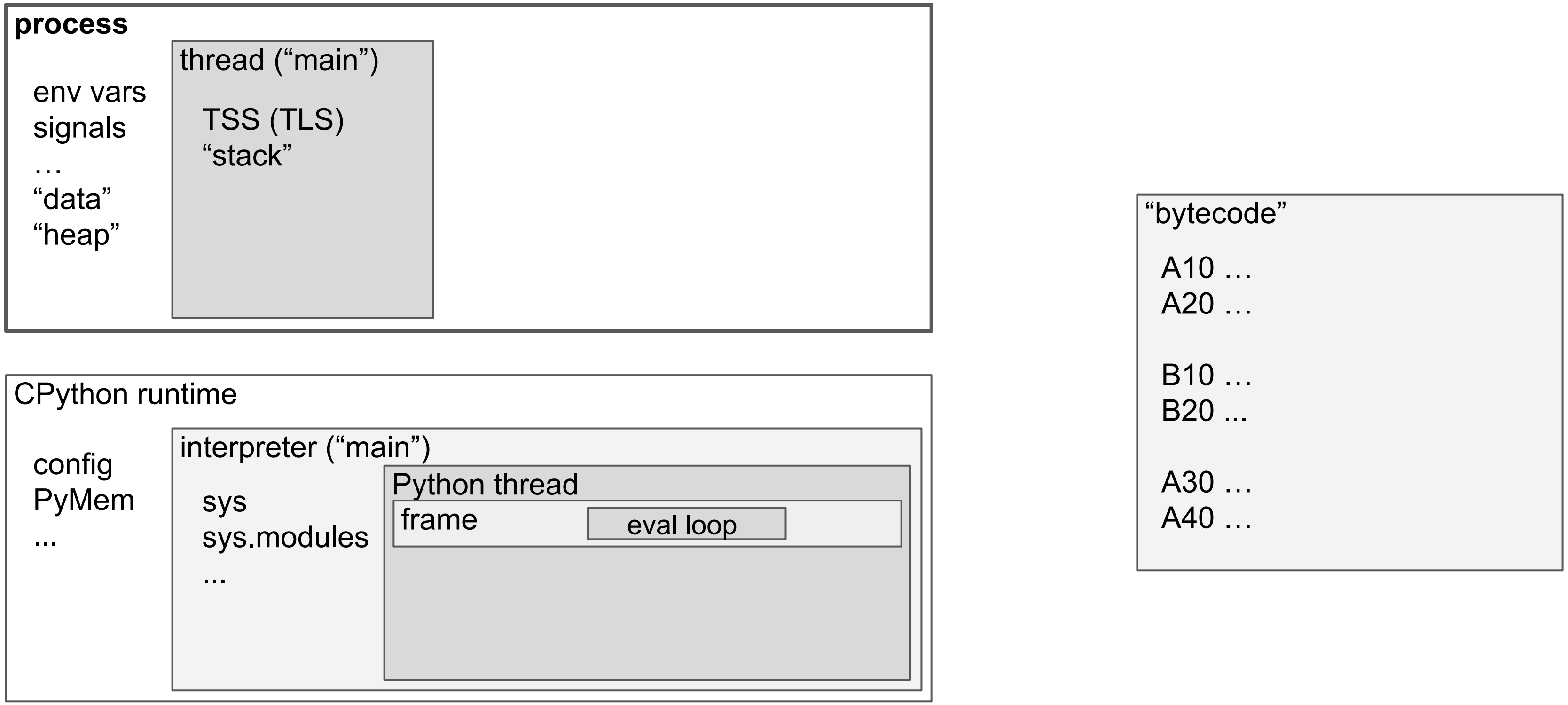


1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



https://bit.ly/2UMMJey

What Happens When Python Runs?



https://bit.ly/2UMMJey

Multi-threading!

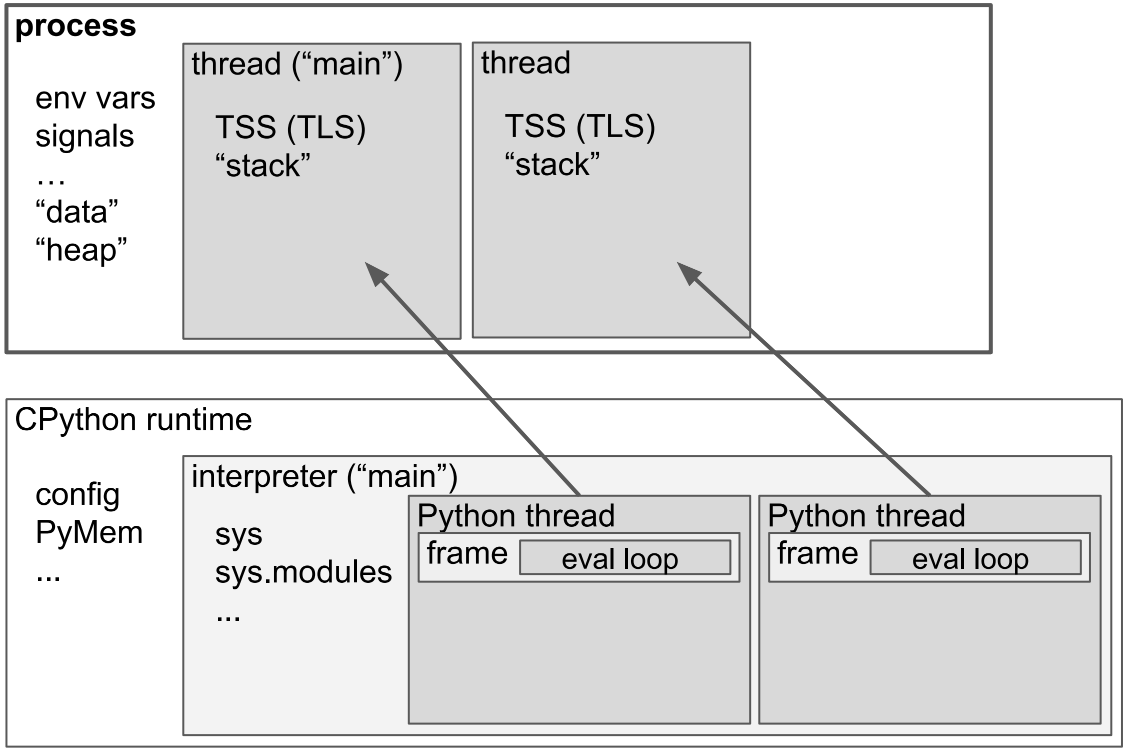
```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

```
__main__

  A10 …
  A20 …
  B10 …
  B20 ...
  A30 …
  A40 …
```

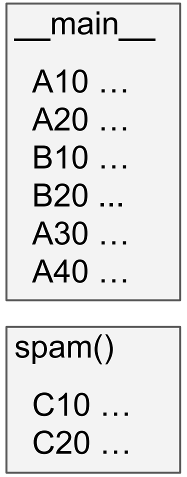
```
spam()

  C10 …
  C20 …
```

https://bit.ly/2UMMJey

Multi-threading!

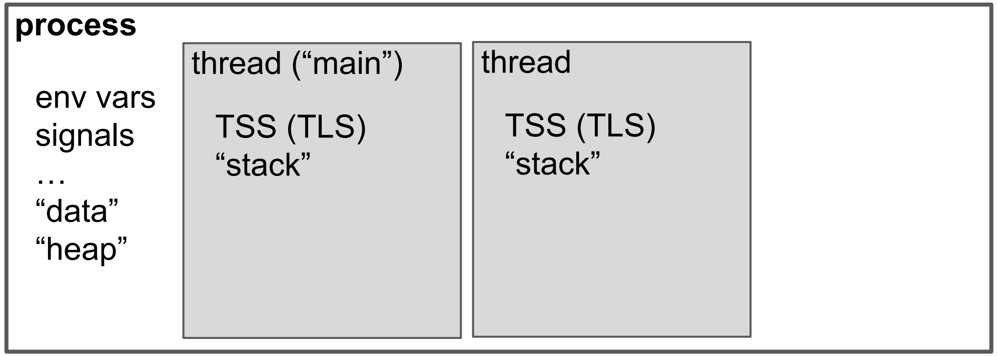


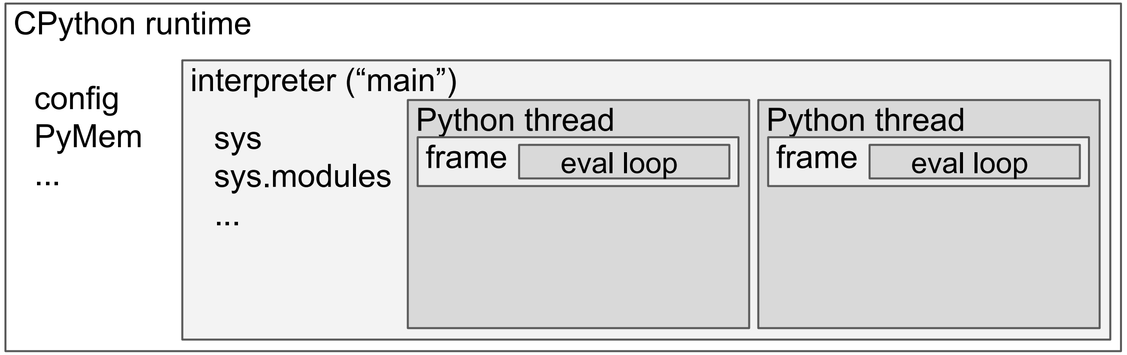
```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



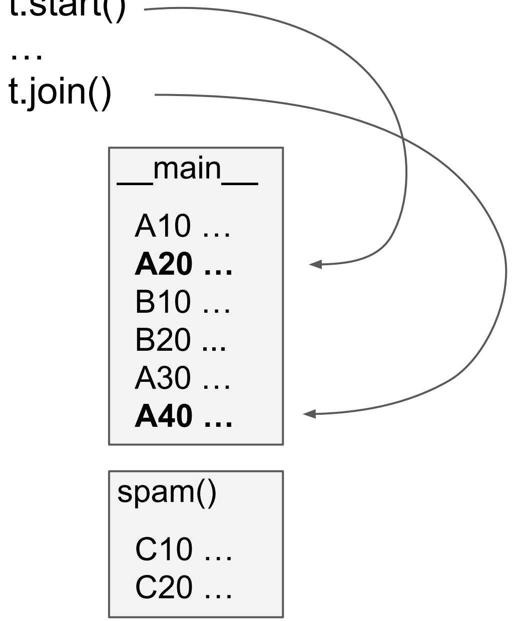
https://bit.ly/2UMMJey

Multi-threading!



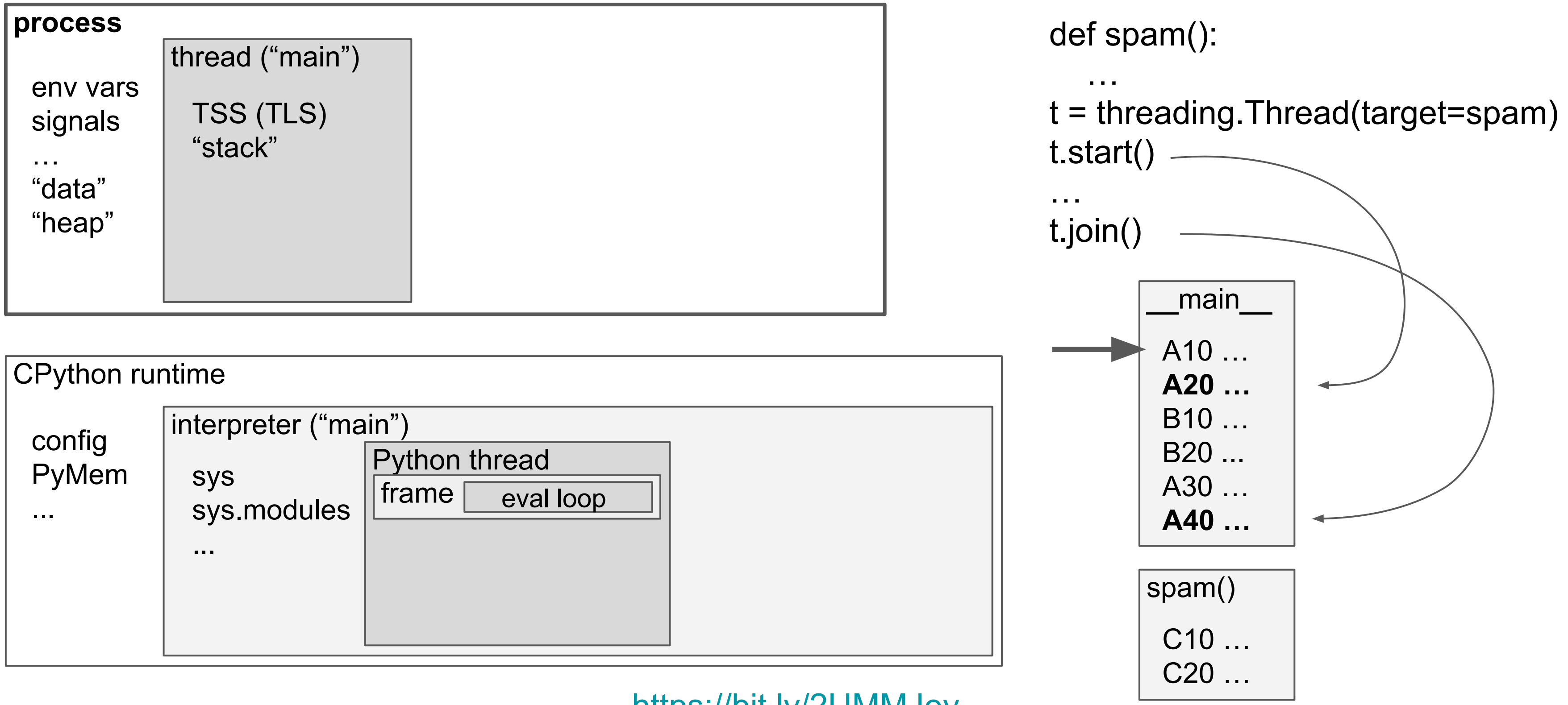


```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



https://bit.ly/2UMMJey

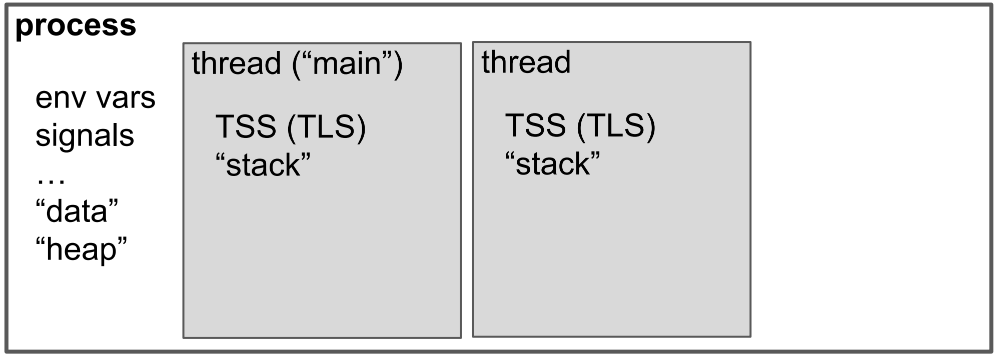
Multi-threading!

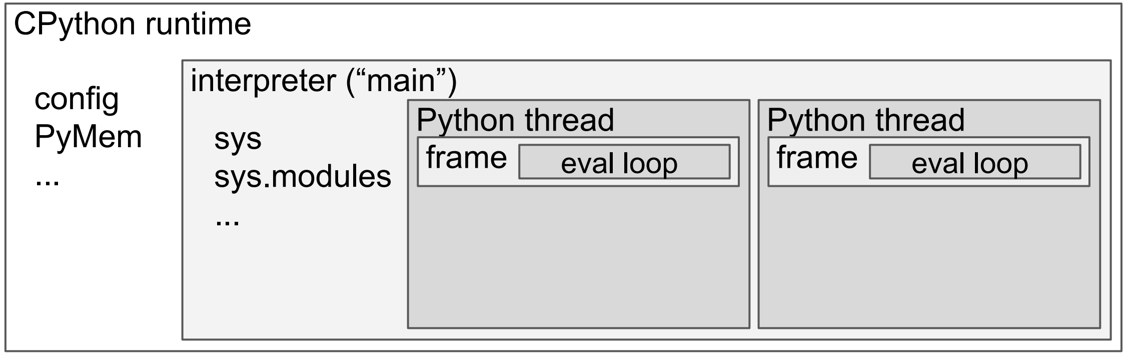


```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

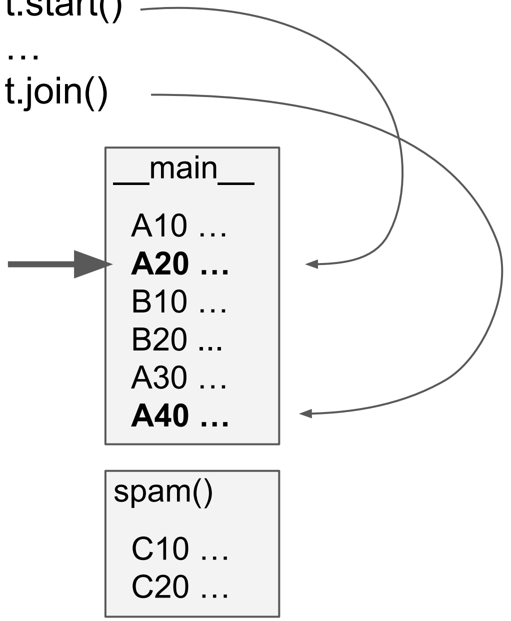
https://bit.ly/2UMMJey

Multi-threading!



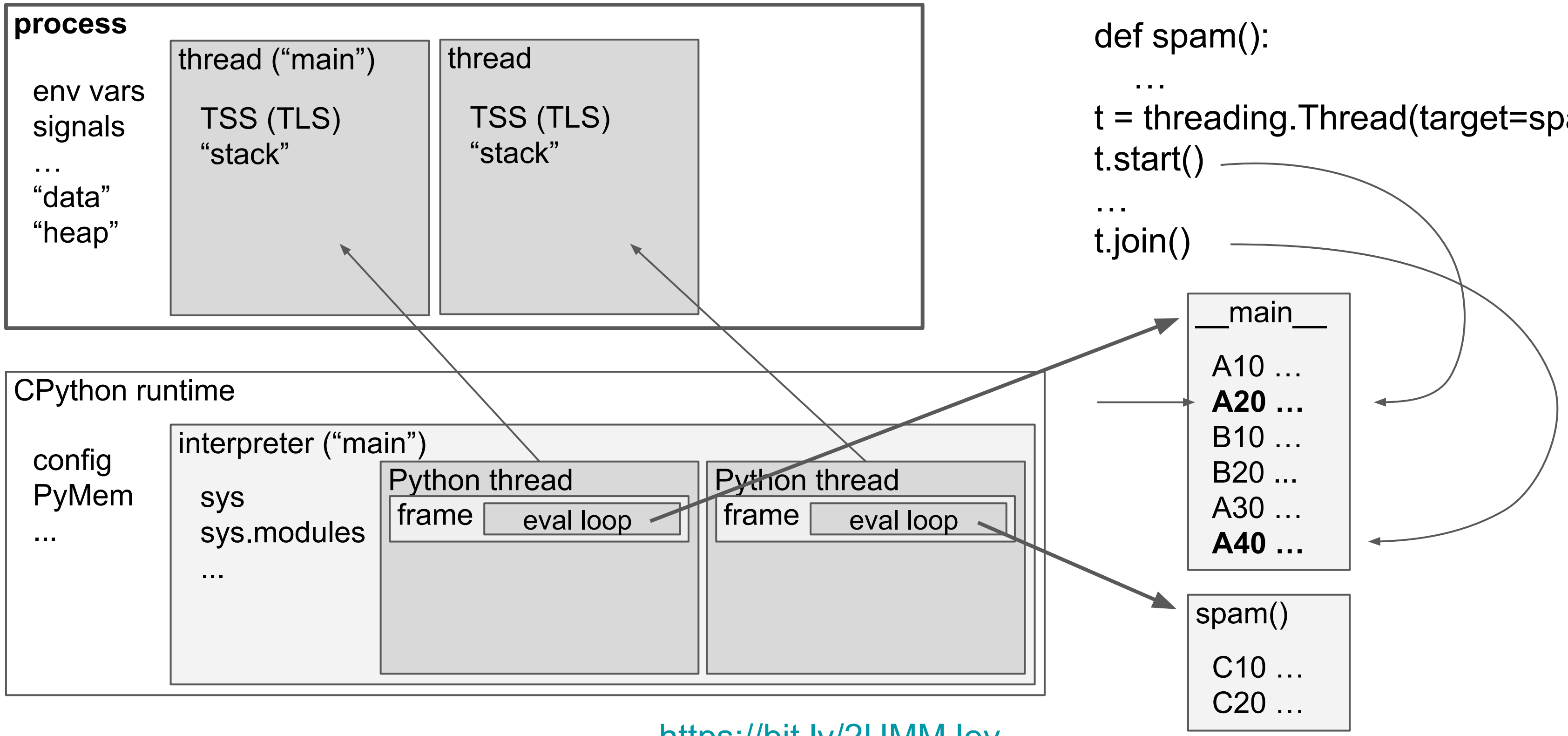


```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



https://bit.ly/2UMMJey

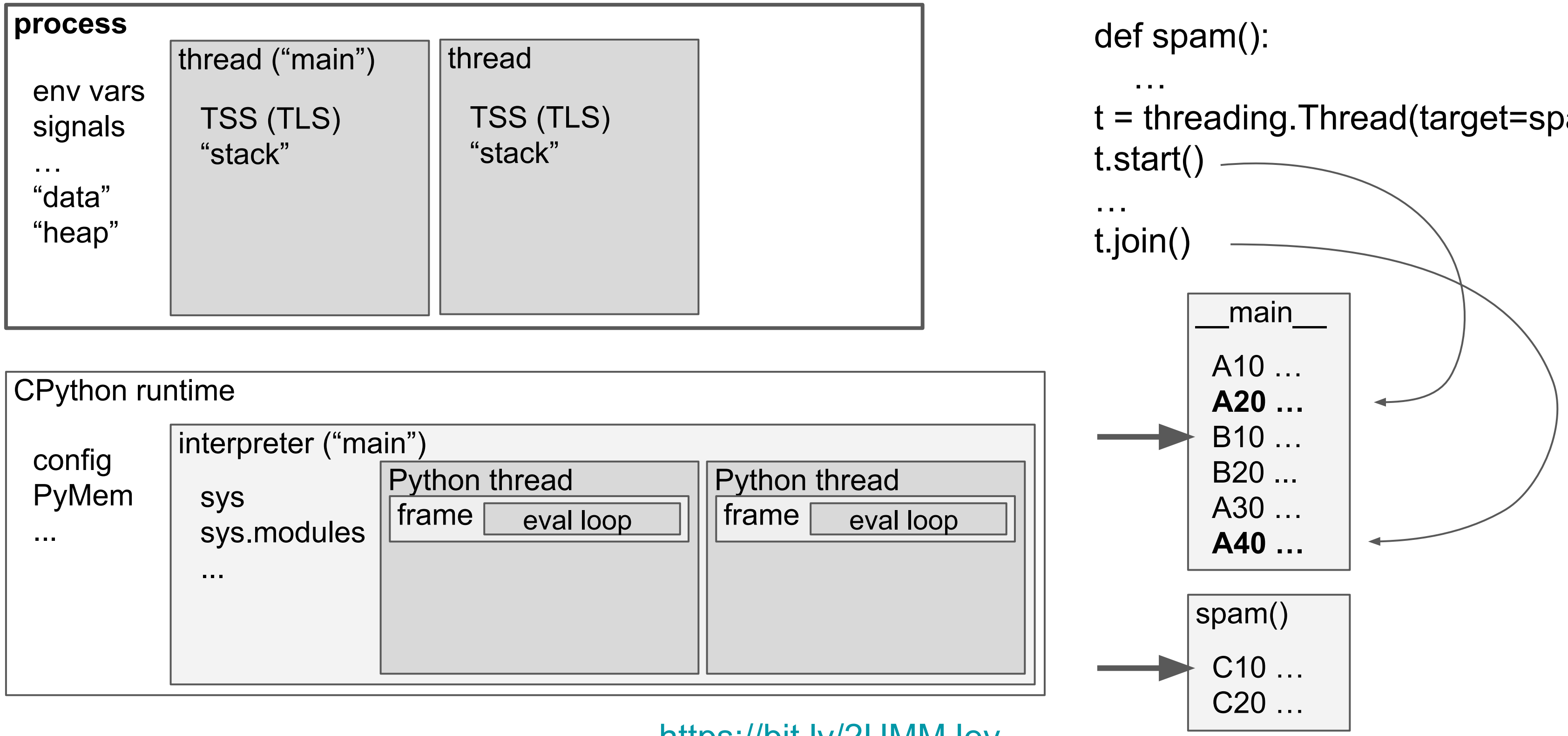
Multi-threading!



```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

https://bit.ly/2UMMJey

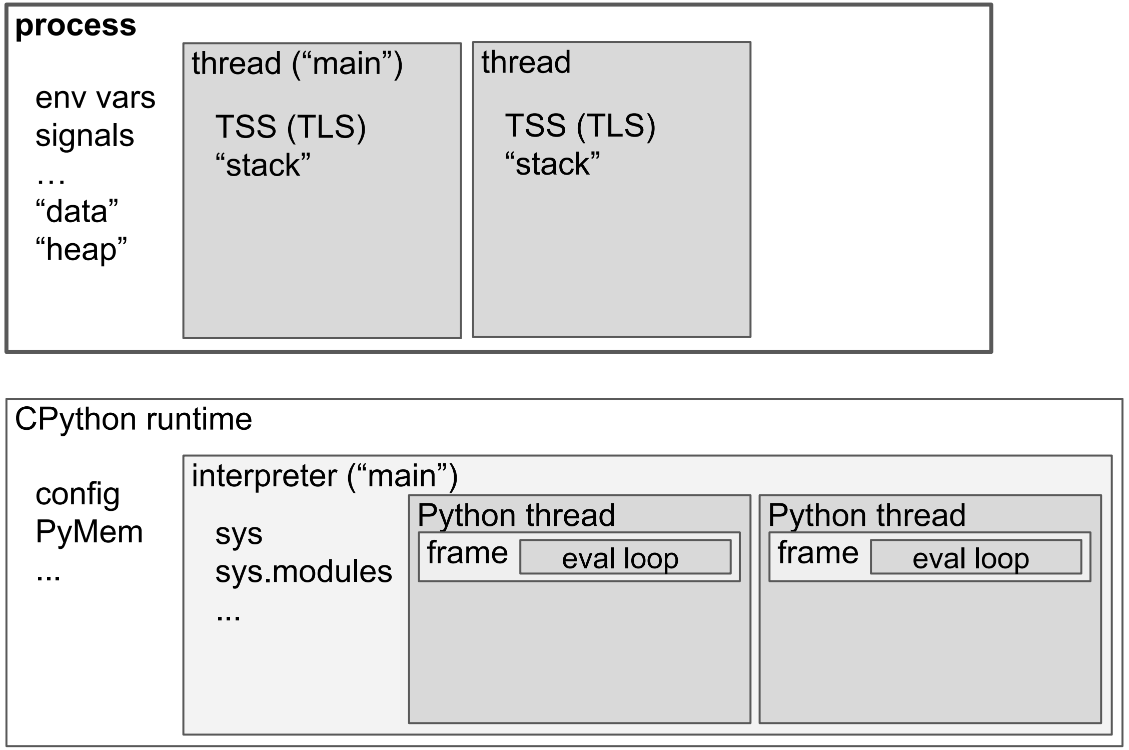
Multi-threading!



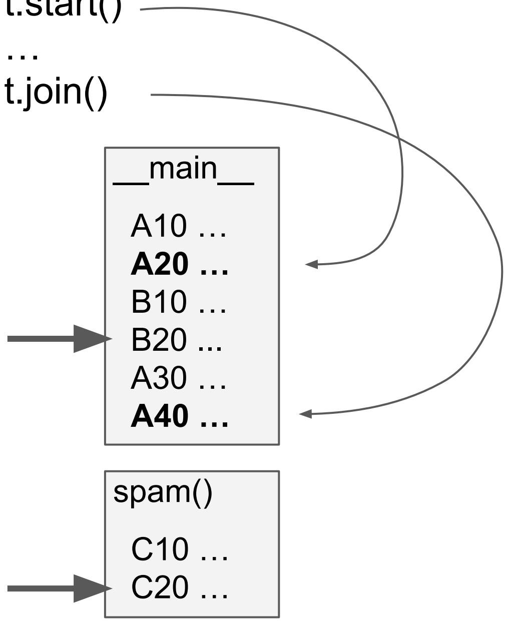
```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

https://bit.ly/2UMMJey

Multi-threading!

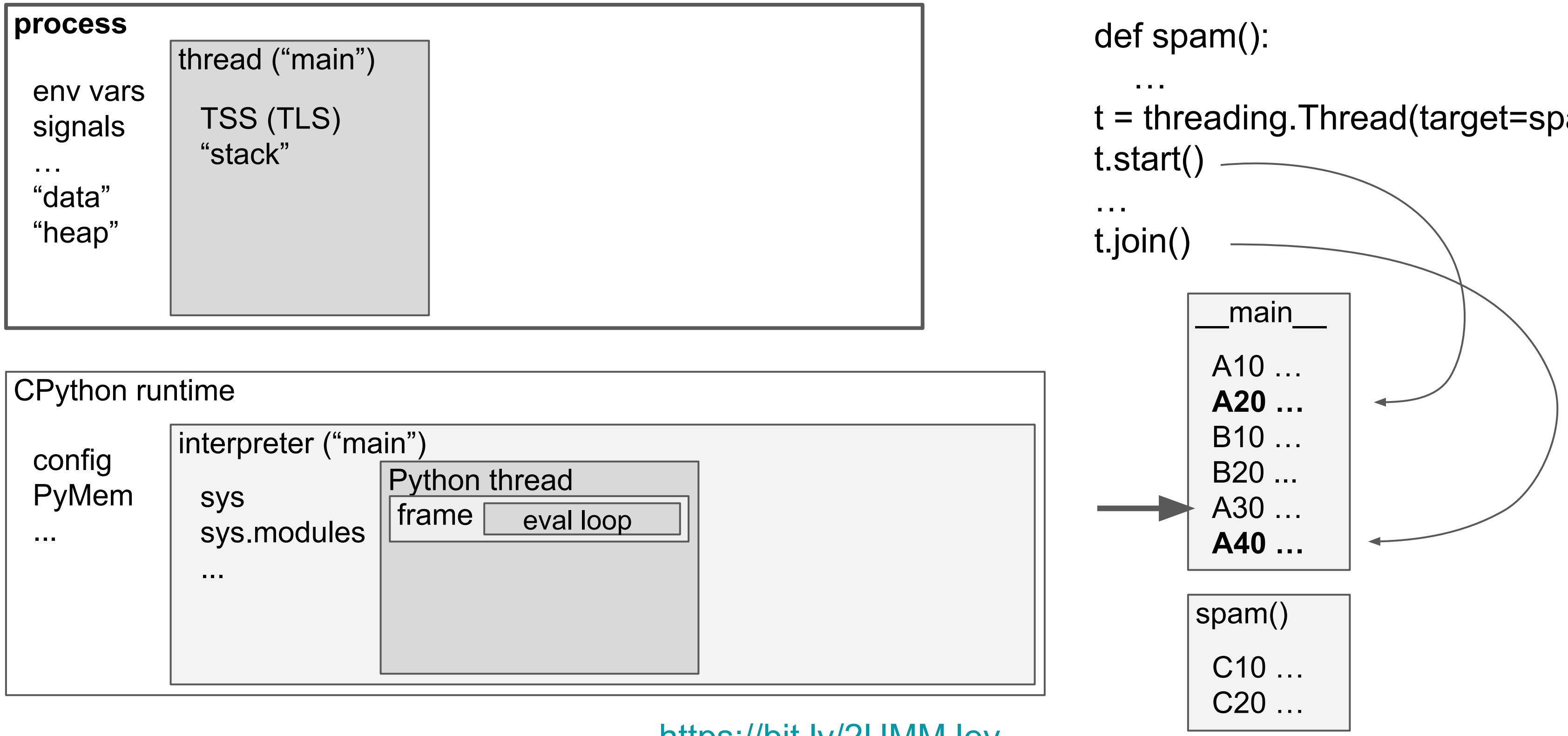


```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



https://bit.ly/2UMMJey

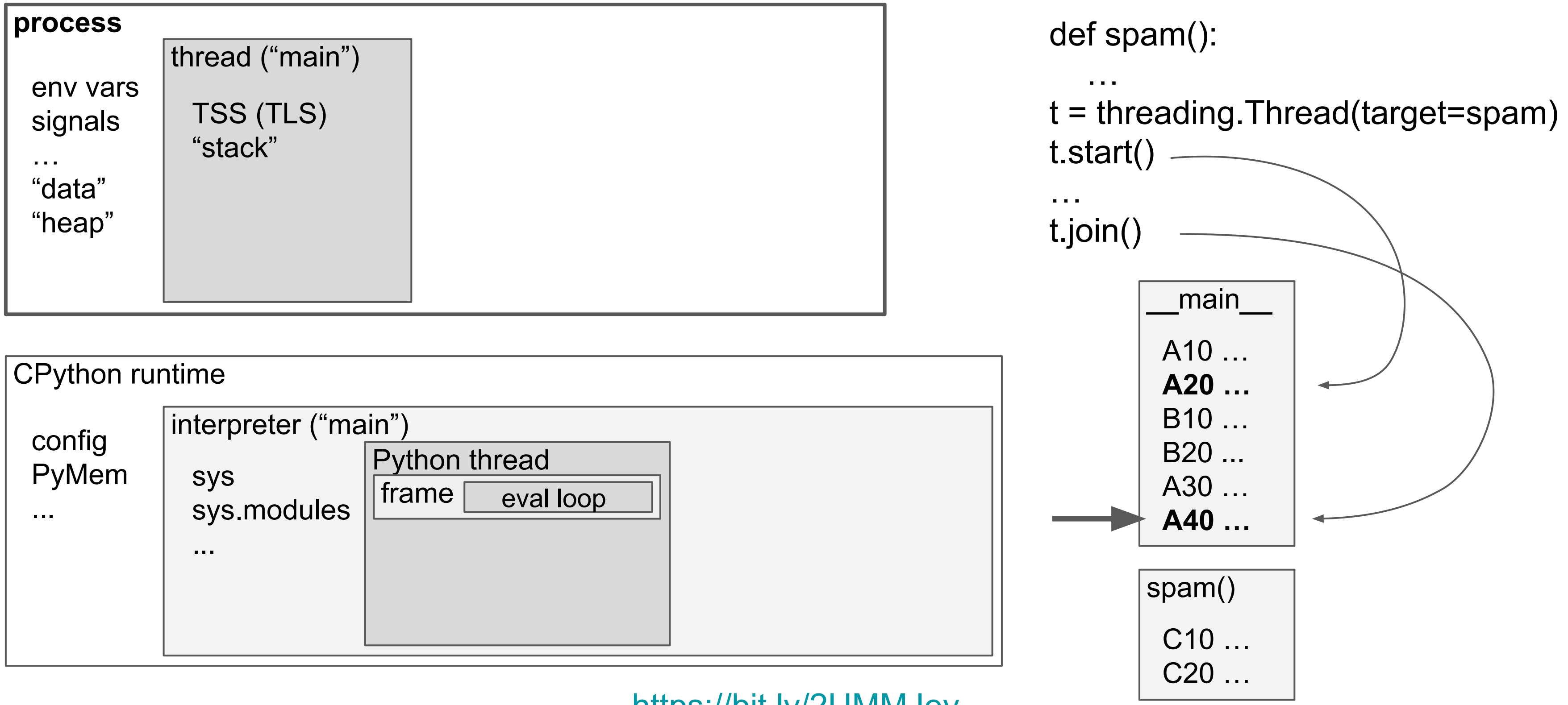
Multi-threading!



```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

https://bit.ly/2UMMJey

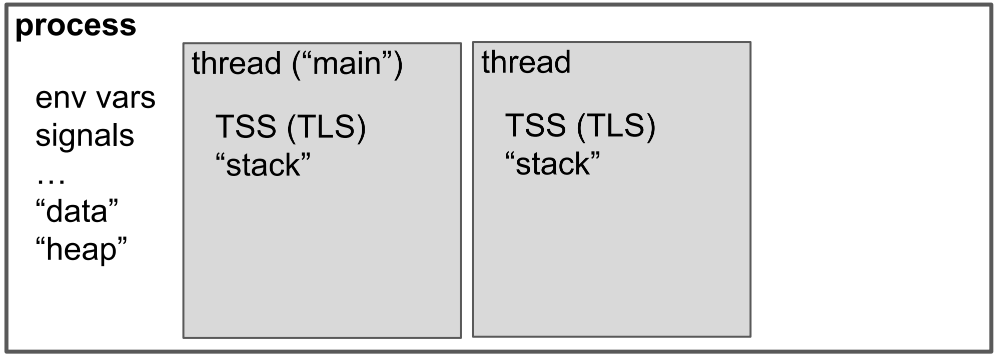
Multi-threading!



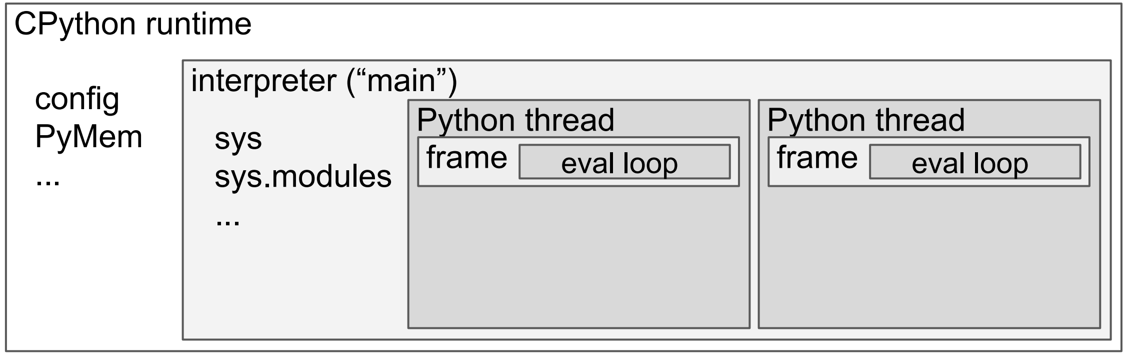
```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

https://bit.ly/2UMMJey

Multi-threading!



```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```

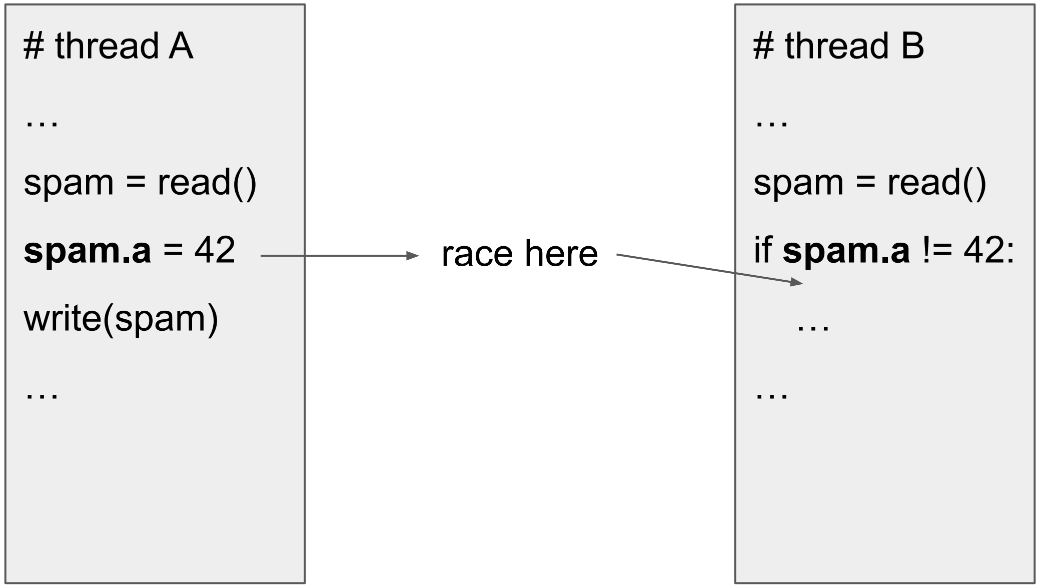


```
"bytecode"
 A10
 A20      # t.start()
 ----------------------------
 B10   C10   B10   C10
 B20   C20   C10   B10
 A30   B10   B20   B20
 C10   B20   C20   A30
 C20   A30   A30   C20
 ----------------------------
 A40      # t.join()
```

https://bit.ly/2UMMJey

“Race Condition”

A.K.A. “Resource Contention”



```
# thread A
…
spam = read()
spam.a = 42
write(spam)
…
```

```
# thread B
…
spam = read()
if spam.a != 42:
    …
…
```

https://bit.ly/2UMMJey

“Race Condition”

A.K.A. “Resource Contention”

```
                    # thread A
                    …
acquire lock A ->
                    spam = read()
                    spam.a = 42
                    write(spam)
release lock A ->
                    …
```

```
                    # thread B
                    …
acquire lock A ->
                    spam = read()
                    if spam.a != 42:
                        …
release lock A ->
                    …
```

https://bit.ly/2UMMJey

“Race Condition”

A.K.A. “Resource Contention”

```
                        # thread A
                        …
acquire lock A ->       spam = read()
                        spam.a = 42
                        write(spam)
release lock A ->       …
```

```
                        # thread B
                        …
acquire lock A ->       spam = read()
                        if spam.a != 42:
                            …
release lock A ->       …
```

https://bit.ly/2UMMJey

“Race Condition”

A.K.A. “Resource Contention”

```
                    # thread A
                    …
acquire lock A ->
                    spam = read()
                    spam.a = 42
                    write(spam)
release lock A ->
                    …
```

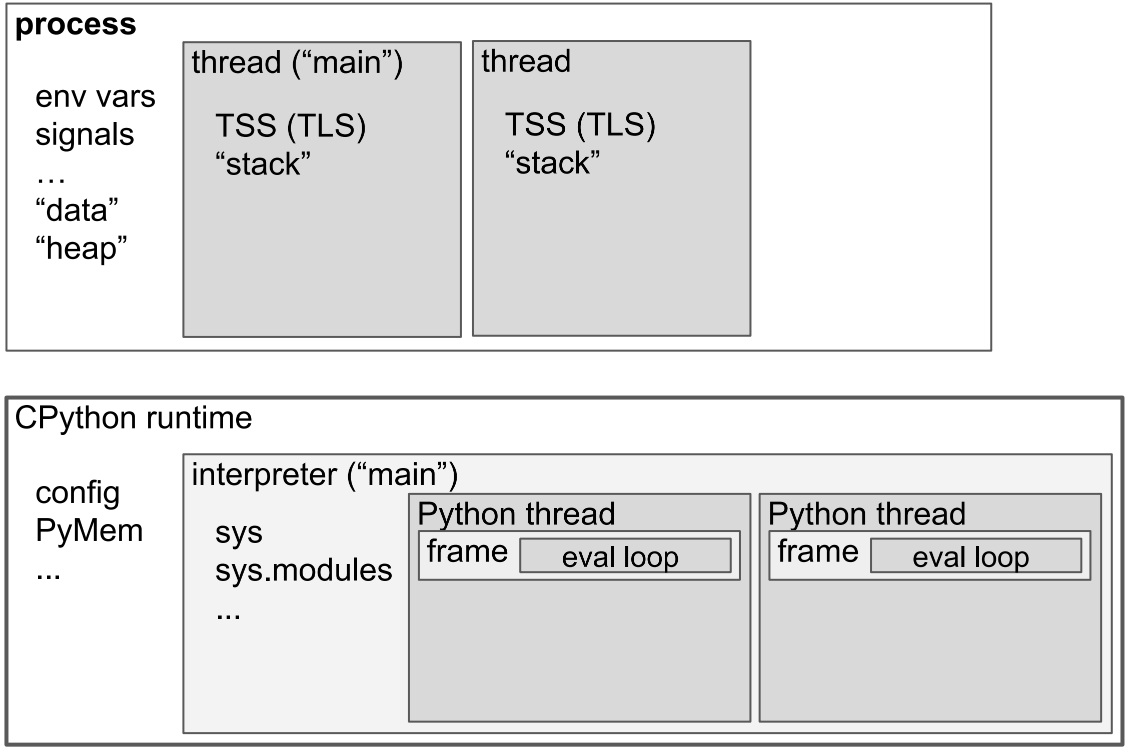
```
                    # thread B
                    …
acquire lock A ->
                    spam = read()
                    if spam.a != 42:
                        …
release lock A ->
                    …
```

```
…
spam = read()
spam.a = 42
write(spam)
…
spam = read()
if spam.a != 42:
    …
…
```

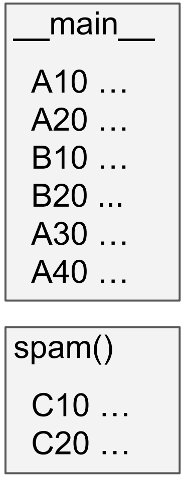
https://bit.ly/2UMMJey

The GIL

The GIL ("Global Interpreter Lock")



```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



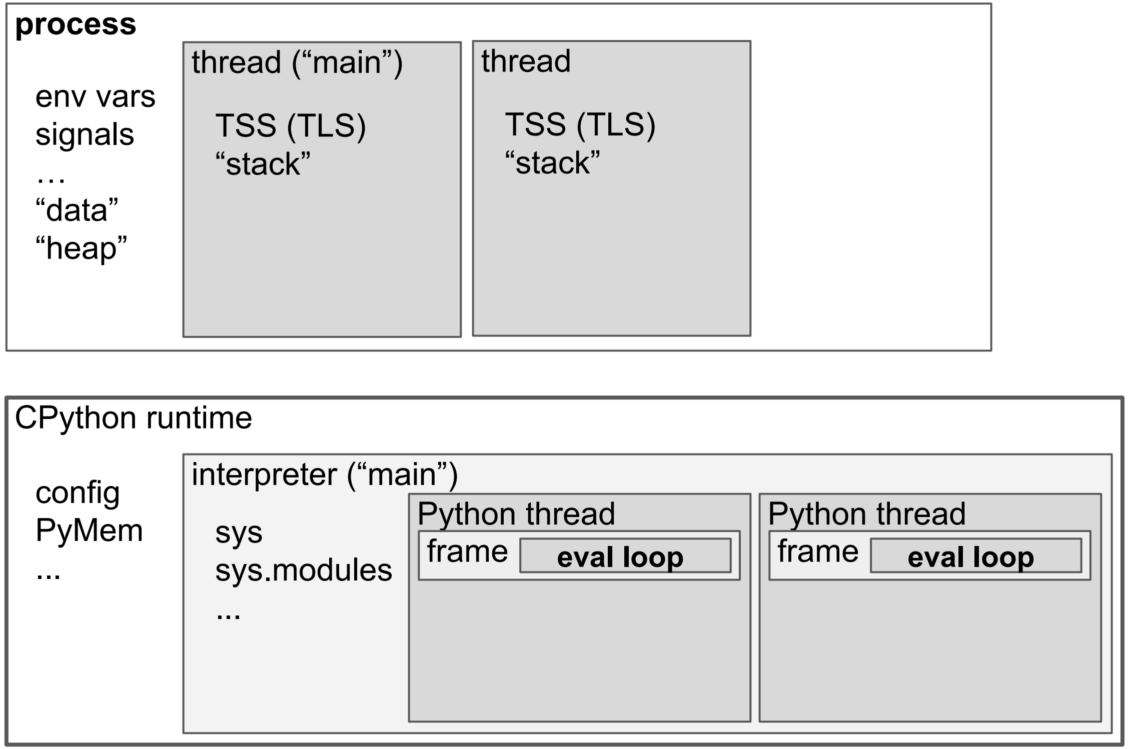
https://bit.ly/2UMMJey

State At Different Layers

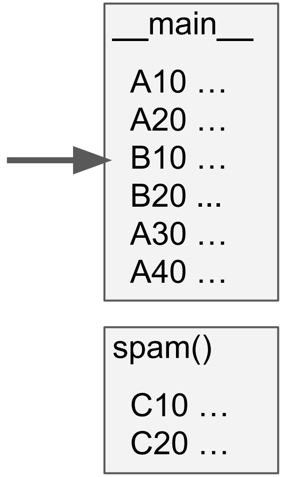
process ->	**global runtime ->**	**interpreter ->**	**thread / stack / ceval**
env vars	GIL	sys module	current frame
sockets	signal handlers	modules	stack depth
file handles	Py_AtExit() funcs	atexit handlers	“tracing”
signals	GC	fork handlers	hook: trace
(thread-local storage)	allocator (mem)	hook: eval_frame	hook: profile
...	objects (w/ refcounts)	codecs	current exception
	pending calls		context
	“eval breaker”		...

https://bit.ly/2UMMJey

The GIL (“Global Interpreter Lock”)



```
def spam():
    …
t = threading.Thread(target=spam)
t.start()
…
t.join()
```



https://bit.ly/2UMMJey

The Eval Loop

```
<set up>

for instruction in code object:

    <maybe side-channel stuff>

    <occasionally release & re-acquire the GIL>

    <execute next instruction>
```

https://bit.ly/2UMMJey

When is the GIL Released?

- eval loop: every few instructions
- around C code that does not touch runtime resources
- around IO operations
- (by C extensions)

https://bit.ly/2UMMJey

Costs and Benefits of the GIL

- Multi-core parallelism of Python code
- ???

- Cheaper than fine-grained locks
- Simpler eval loop implementation
- Simpler object implementation
- Simpler C-API implementation

https://bit.ly/2UMMJey

Costs and **Benefits** of the GIL

- Multi-core parallelism of Python code
- ???

- Cheaper than fine-grained locks
- Simpler implementation
  - eval loop
  - object system
  - C-API

https://bit.ly/2UMMJey

Effect and Perception

Who does it really affect?

- Users with threaded, CPU-bound *Python* code (relatively few people)
- Basically no one else

Effect and Perception

Who does it really affect?

- Users with threaded, CPU-bound *Python* code (relatively few people)
- Basically no one else

Why? **C implementation releases the GIL around IO and CPU-intensive code.**

https://bit.ly/2UMMJey

Effect and Perception

Who does it really affect?

- Users with threaded, CPU-bound *Python* code (relatively few people)
- Basically no one else

So why does the GIL get such a bad wrap?

- Lack of understanding
- Experience with other programming languages
- Haters gonna hate

https://bit.ly/2UMMJey

Working Around the GIL

- C-extension modules
  - rewrite CPU-bound code in C
  - release the GIL around that code
- multi-processing
- (async / await)

https://bit.ly/2UMMJey

Past Efforts to Remove the GIL

- 1999 Greg Stein
- Larry Hastings' Gilectomy (on hold)
- other Python implementations
  - unladen swallow
  - …

https://bit.ly/2UMMJey

Other Python Implementations

	GIL?	C-API?	latest Py version
CPython	yes	yes	3.7
Jython	no	JyNI	2.7
IronPython	no	yes?	2.7
PyPy (& w/STM)	yes (no)	cffi, cpyext	3.6
MicroPython	~yes	no?	~3.4+

https://bit.ly/2UMMJey

The Future

A New C-API

- the history
- the problem
- the solutions

https://bit.ly/2UMMJey

The C-API

- historically fundamental to Python’s success
- organic growth
- early efforts to simplify
- core devs: growing concerns
- core devs: increasing efforts

https://bit.ly/2UMMJey

The Problem

- getting rid of GIL needs low-level changes
- parts of public C-API expose low-level details (e.g. refcounts)
- so...
- getting rid of GIL requires breaking parts of C-API

https://bit.ly/2UMMJey

The Causes

- didn’t think 20+ years into future
- “consenting adults”
- accidental leaks

https://bit.ly/2UMMJey

The Solutions

- someone has to care enough to do the work
- physically separate the categories of C-API
- more opaque structs
- Python (C)FFI
- (maybe) break compatibility in a few places
- deprecate C-API in favor of something like Cython (official)
- ...

https://bit.ly/2UMMJey

The Solutions

- someone has to care enough to do the work
- physically separate the categories of C-API
- more opaque structs
- Python (C)FFI
- (maybe) break compatibility in a few places
- deprecate C-API in favor of something like Cython (official)
- ...

https://bit.ly/2UMMJey

The Solutions

- someone has to care enough to do the work
- **physically separate the categories of C-API**
- more opaque structs
- Python (C)FFI
- (maybe) break compatibility in a few places
- deprecate C-API in favor of something like Cython (official)
- ...

Categorizing the C-API

“internal”	“Do not touch!”
“private”	“Use at your own risk!”
“unstable”	“Go for it (but rebuild your extension each Python release)!”
“stable”	“Worry-free!”

https://bit.ly/2UMMJey

The Solutions

- someone has to care enough to do the work
- physically separate the categories of C-API
- more opaque structs
- Python (C)FFI
- (maybe) break compatibility in a few places
- move toward something like Cython (official)
- ...

https://bit.ly/2UMMJey

The Projects

- Victor Stinner’s, well, all of it
  - https://pythoncapi.readthedocs.io/roadmap.html
- Steve Dower's efforts
  - https://mail.python.org/archives/list/capi-sig@python.org/thread/B2VDVLABM4RQ4ATEJXFZYWEGTBZPUBKW/
- Petr Viktorin's projects
- bringing sanity to runtime initialization and finalization
- others

<capi-sig@python.org>

The Projects

- Victor Stinner’s, well, all of it
    - https://pythoncapi.readthedocs.io/roadmap.html
- Steve Dower's efforts
    - https://mail.python.org/archives/list/capi-sig@python.org/thread/B2VDVLABM4RQ4ATEJXFZYWEGTBZPUBKW/
- Petr Viktorin's projects
- bringing sanity to runtime initialization and finalization
- others

<capi-sig@python.org>

Beyond the C-API...

Subinterpreters

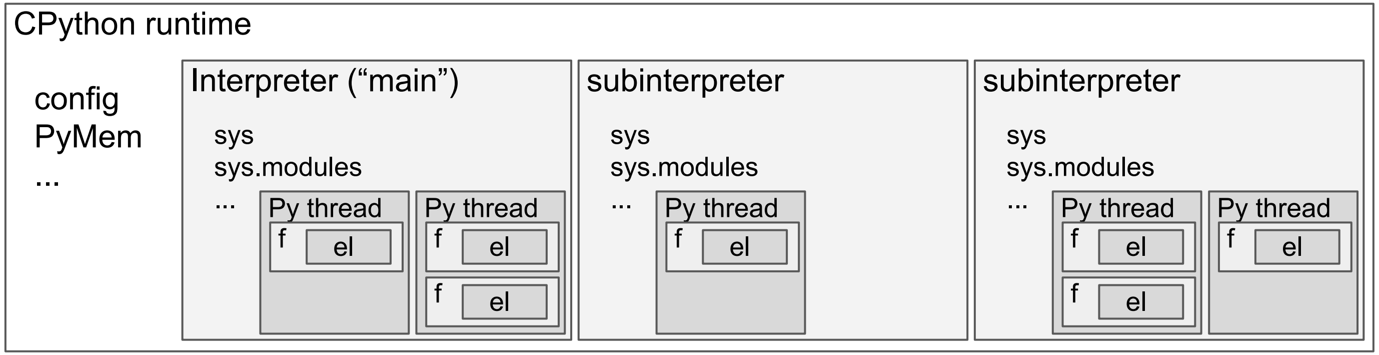
!!!

Interpreters in a Single Process

- initial interpreter: “main”
    - has certain responsibilities
- “subinterpreter”: any other interpreter created within the runtime
- isolated-ish

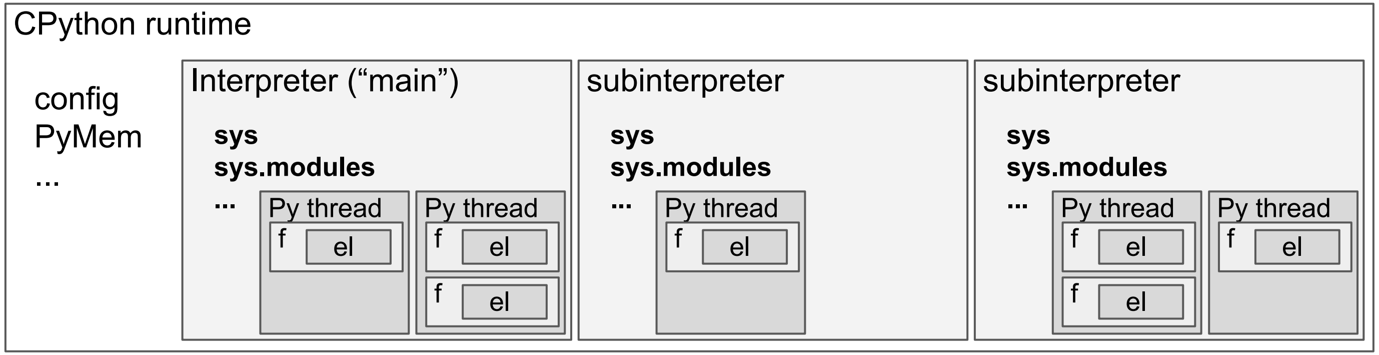
Interpreters in a Single Process

- initial interpreter: “main”
    - has certain responsibilities
- “subinterpreter”: any other interpreter created within the runtime
- isolated-ish



Interpreters in a Single Process

- initial interpreter: “main”
    - has certain responsibilities
- “subinterpreter”: any other interpreter created within the runtime
- isolated-ish



Subinterpreters

- initial interpreter: “main”
  - has certain responsibilities
- “subinterpreter”: any other interpreter created within the runtime
- isolated-ish
- C-API for over 20 years
- PEP 554: stdlib module

https://bit.ly/2UMMJey

PEP 554 - “Multiple Interpreters in the Stdlib”

- https://www.python.org/dev/peps/pep-0554/
- new “interpreters” module
    - create(), list_all(), etc.
    - Interpreter class
    - create_channel()
    - RecvChannel, SendChannel

https://bit.ly/2UMMJey

PEP 554 - “Multiple Interpreters in the Stdlib”

- https://www.python.org/dev/peps/pep-0554/
- new “interpreters” module
    - create(), list_all(), etc.
    - Interpreter class
    - create_channel()
    - RecvChannel, SendChannel

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""

    print('spam')

"""))
```

https://bit.ly/2UMMJey

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""

    print('spam')

"""))
```

https://bit.ly/2UMMJey

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""

    print('spam')

"""))
```

https://bit.ly/2UMMJey

PEP 554: Example 2

```
interp = interpreters.create()

def func():
    interp.run(dedent("""
        print('spam')
    """))
t = threading.Thread(target=func)
t.start()
```

https://bit.ly/2UMMJey

PEP 554: Example 3

```
interp = interpreters.create()
interp.run(dedent("""
    x = 'spam'
"""))
interp.run(dedent("""
    print(x)
"""))
```

https://bit.ly/2UMMJey

PEP 554: Example 3

```
interp = interpreters.create()
interp.run(dedent("""
    x = 'spam'
"""))
interp.run(dedent("""
    print(x)
"""))
```

https://bit.ly/2UMMJey

PEP 554: Example 3

```
interp = interpreters.create()
interp.run(dedent("""
    x = 'spam'
"""))
interp.run(dedent("""
    print(x)
"""))
```

https://bit.ly/2UMMJey

PEP 554 - “Multiple Interpreters in the Stdlib”

- https://www.python.org/dev/peps/pep-0554/
- new “interpreters” module
    - create(), list_all(), etc.
    - Interpreter class
    - create_channel()
    - RecvChannel, SendChannel

PEP 554 - “Multiple Interpreters in the Stdlib”

- https://www.python.org/dev/peps/pep-0554/
- new “interpreters” module
  - create(), list_all(), etc.
  - Interpreter class
  - create_channel()
  - RecvChannel, SendChannel

PEP 554 - “Multiple Interpreters in the Stdlib”

- https://www.python.org/dev/peps/pep-0554/
- new “interpreters” module
  - create(), list_all(), etc.
  - Interpreter class
  - create_channel()
  - RecvChannel, SendChannel

For now:

- limited supported types
  - str, int, None, etc.
  - PEP 3118 buffers
- actual objects not shared
- no buffering

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

PEP 554: Example 4

```
(rchan, schan
 ) = interpreters.create_channel()
interp = interpreters.create()

def func():
    interp.run(dedent("""
        import spam
        data = spam.do_something()
        ch.send(data)    # blocks
    """, channels={ch: schan})
```

```
t = threading.Thread(target=func)
t.start()

data = rchan.recv()    # blocks
process_data(data)
```

https://bit.ly/2UMMJey

Who Cares?

- a human-oriented concurrency model (IMHO)
    - "opt-in sharing"
- “the isolation of processes, with the efficiency of threads”
- gateway to multi-core CPython

https://bit.ly/2UMMJey

Who Cares?

- a human-oriented concurrency model (IMHO)
  - "opt-in sharing"
- “the isolation of processes, with the efficiency of threads”
- gateway to multi-core CPython

Who Cares?

- a human-oriented concurrency model (IMHO)
    - "opt-in sharing"
- “the isolation of processes, with the efficiency of threads”
- gateway to multi-core CPython

https://bit.ly/2UMMJey

Who Cares?

- a human-oriented concurrency model (IMHO)
  - "opt-in sharing"
- “the isolation of processes, with the efficiency of threads”
- gateway to multi-core CPython

Stop Sharing the GIL!!!

State At Different Layers

process ->	**global runtime ->**	**interpreter ->**	**thread / stack / ceval**
env vars	**GIL**	sys module	current frame
sockets	signal handlers	modules	stack depth
file handles	Py_AtExit() funcs	atexit handlers	“tracing”
signals	GC	fork handlers	hook: trace
(thread-local storage)	allocator (mem)	hook: eval_frame	hook: profile
...	objects	codecs	current exception
	pending calls		context
	“eval breaker”		...

https://bit.ly/2UMMJey

Stop Sharing the GIL!

- allow each interpreter to execute independently
- threads within an interpreter would still share a “GIL”
- shouldn’t require wide-spread changes
- no change to single-threaded (or single-interpreter) performance

https://bit.ly/2UMMJey

Stop Sharing the GIL!

- allow each interpreter to execute independently
- threads within an interpreter would still share a “GIL”
- shouldn’t require wide-spread changes
- no change to single-threaded (or single-interpreter) performance

https://bit.ly/2UMMJey

Why Hasn’t It Been Done Already?

- forgotten feature
- no one interested enough (to do the work)
- “good enough” alternatives
- scary! (or not)
- blockers...

https://bit.ly/2UMMJey

the blockers

- lingering bugs
- subinterpreters only in C-API
- how to guard against races between interpreters?
- enough time to do the work!
- C globals

(╯°□°)╯︵ ┻━┻

https://bit.ly/2UMMJey

the blockers

- lingering bugs
- subinterpreters only in C-API
- how to guard against races between interpreters?
- enough time to do the work!
- C globals

(╯°□°)╯︵ ┻━┻

https://bit.ly/2UMMJey

the blockers

- lingering bugs
- subinterpreters only in C-API
- how to guard against races between interpreters?
- enough time to do the work!
- C globals

(╯°□°)╯︵ ┻━┻

https://bit.ly/2UMMJey

the blockers

- lingering bugs
- subinterpreters only in C-API
- how to guard against races between interpreters?
- enough time to do the work!
- C globals

(╯°□°)╯︵ ┻━┻

https://bit.ly/2UMMJey

C "Globals"

- “static globals”, "static locals"
- TSS/TLS (Thread-Specific Storage)
- in the CPython code base
- in extension modules
  - static types, exceptions, singletons; etc.
  - C globals in included shared libraries (e.g. OpenSSL in cryptography)
  - efforts to fix: PEPs 3121, 384, 489, (573), ~~575~~, (579), (580); Cython; Red Hat; Instagram
  - (type “slots”)

https://bit.ly/2UMMJey

the project

https://github.com/ericsnowcurrently/multi-core-python

the project

https://github.com/ericsnowcurrently/multi-core-python

- PEP 554
- resolve bugs
- deal with C globals
- move some runtime state into the interpreter state
    - including the GIL

https://bit.ly/2UMMJey

the project

https://github.com/ericsnowcurrently/multi-core-python

- PEP 554
- resolve bugs
- deal with C globals
- move some runtime state into the interpreter state
  - including the GIL

State At Different Layers

process ->	**global runtime ->**	**interpreter ->**	**thread / stack / ceval**
env vars	**GIL**	sys module	current frame
sockets	signal handlers	modules	stack depth
file handles	Py_AtExit() funcs	atexit handlers	“tracing”
signals	**GC**	fork handlers	hook: trace
(thread-local storage)	**allocator (mem)**	hook: eval_frame	hook: profile
...	**objects**	codecs	current exception
	pending calls		context
	“eval breaker”		...

https://bit.ly/2UMMJey

Beneficial Side Effects

- find bugs and deficiencies in runtime (e.g. init/fini)
- motivation to fix them
- clean-up in runtime implementation (incl. globals, C-API, header files)
- reduce coupling between components in runtime implementation
- encourage fewer static globals in C extension modules
- (improve interpreter startup performance)
- (improve object isolation (e.g. in memory))
- ...

https://bit.ly/2UMMJey

What’s Next?

1. PEP 554, blockers, and per-interpreter GIL
2. low-hanging fruit (optimization)
3. deferred functionality

Thanks!

Thanks!

Questions?

Resources

- https://docs.google.com/presentation/d/1BuU6e-CKdZxDL5z9VBp19LAaIY8Ys2-jlcz-mD0Vr3c/
    - https://bit.ly/2UMMJey
- https://github.com/ericsnowcurrently/multi-core-python
- https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock
- https://wiki.python.org/moin/GlobalInterpreterLock#Eliminating_the_GIL
- twitter: @ericsnowcrntly