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

Eric Snow
PyCon 2019

https://bit.ly/2UMMJey
@ericsnowcrntly

Today we'll be talking about CPython's GIL and how we can move past it.
Follow that URL to get to this slide deck.
I'll announce any updates to the slides or other resources on twitter.

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

You may or may not recognize some of the things I've done as a Python core dev.

So, why is the GIL a topic in which I have any interest?

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

In late 2014 I had an experience that motivated me to work on fixing Python's multi-core story.

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

Along those lines, we’ll work on an understanding of the situation and how it can be changed.

Let me be clear before we start, I’m not going to be perfect and I don't have time to go into every detail.

If have time at the end then I'll take questions.

Context

OK...Brace yourselves!
The firehose is coming...
We're going to zip through a lot of these slides.

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

These are effectively the layers of CPython.

There is one runtime in the process.
There are one or more interpreters in the runtime.
There are one or more Python threads per interpreter.
There is one call stack per Python thread.
There is one eval loop per frame in the call stack.

Let's see how that applies in practice...

What Happens When Python Runs?

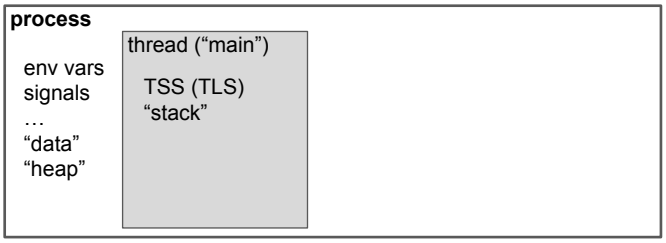
1. process initializes

https://bit.ly/2UMMJey

The next bunch of slides are a less-than-exact representation, but they're good enough!

When a process starts there are certain resources it has which are global to the process.

What Happens When Python Runs?

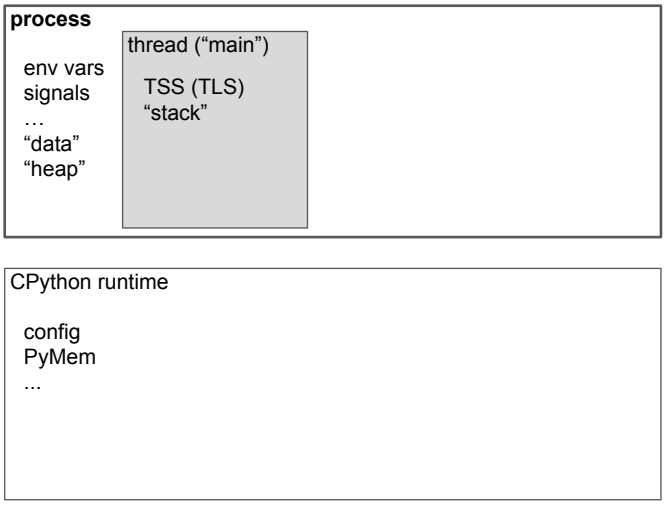


1. process initializes
2. main thread starts

https://bit.ly/2UMMJey

Likewise each OS thread has certain exclusive resources.

What Happens When Python Runs?

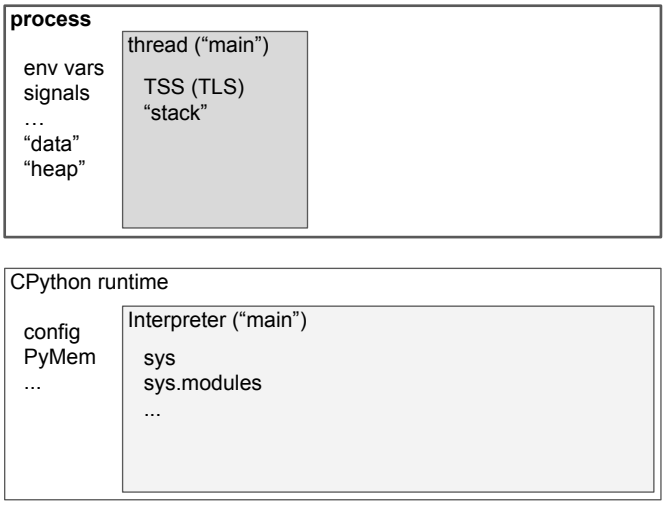


1. process initializes
2. main thread starts
3. Python runtime initializes

https://bit.ly/2UMMJey

The runtime is effectively everything Python-related in the process.

What Happens When Python Runs?



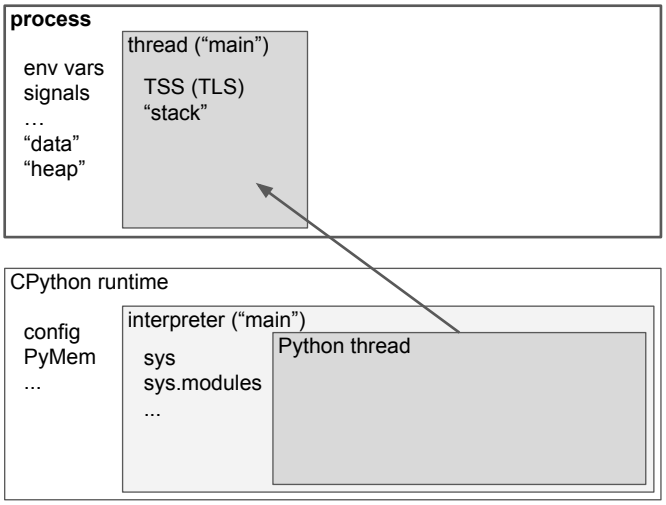
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes

https://bit.ly/2UMMJey

An interpreter is all runtime state that Python threads share in common.
We usually use the term a bit more generally. In this talk we're using a specific meaning.

(In most existing code this is the same as the global runtime state. There may be more than one interpreter in the runtime, but we'll talk more about that later.)
(In single-threaded code there is no difference from the Python thread state.)

What Happens When Python Runs?

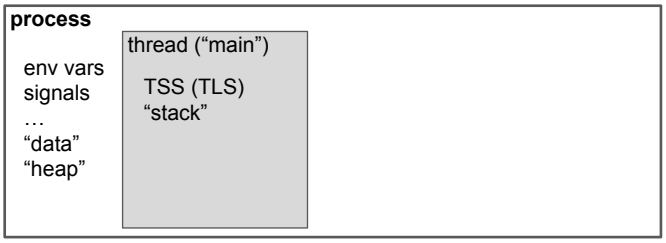


1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes

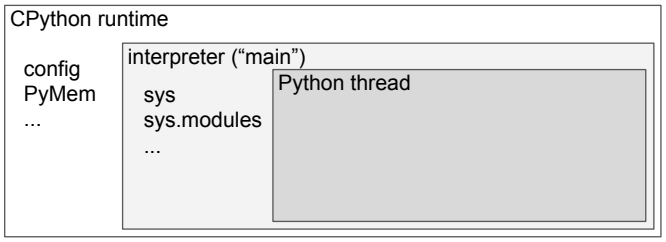
https://bit.ly/2UMMJey

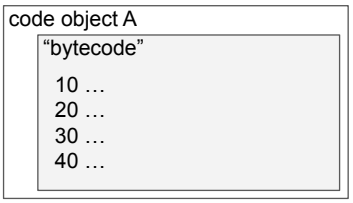
The Python thread is associated with the OS thread and keeps thread-specific runtime state.

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

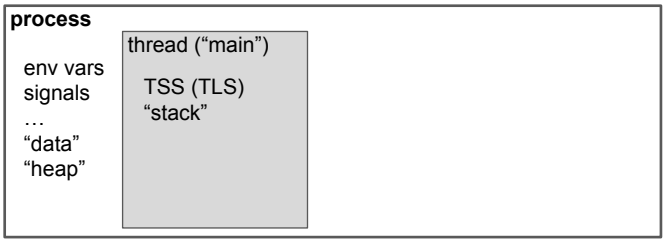




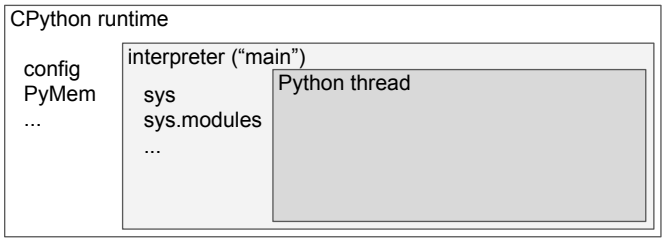
https://bit.ly/2UMMJey

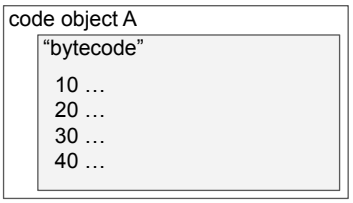
When loaded, scripts, modules, functions, and classes compile to code objects. A key part is the sequence of bytecode instructions to be executed by the Python interpreter.

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

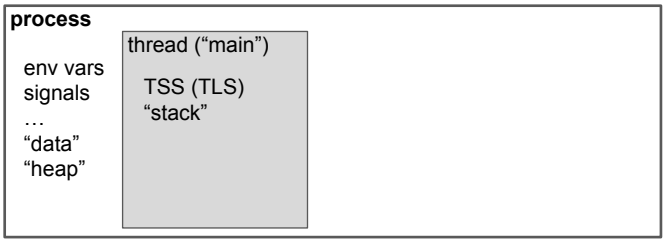




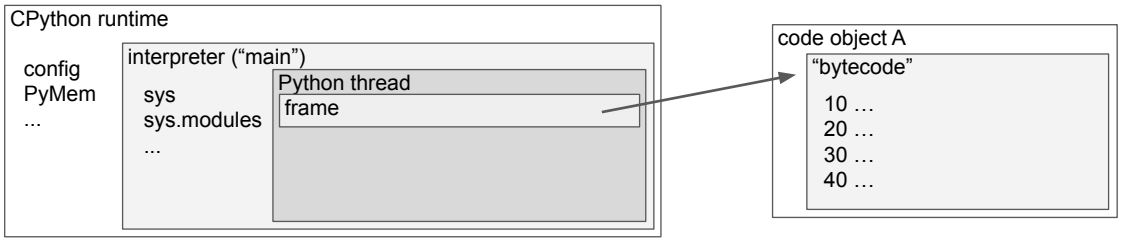
https://bit.ly/2UMMJey

For simplicity let's say that our imaginary Python program compiles to only a handful of instructions when loaded.
A real program would have many more..

What Happens When Python Runs?



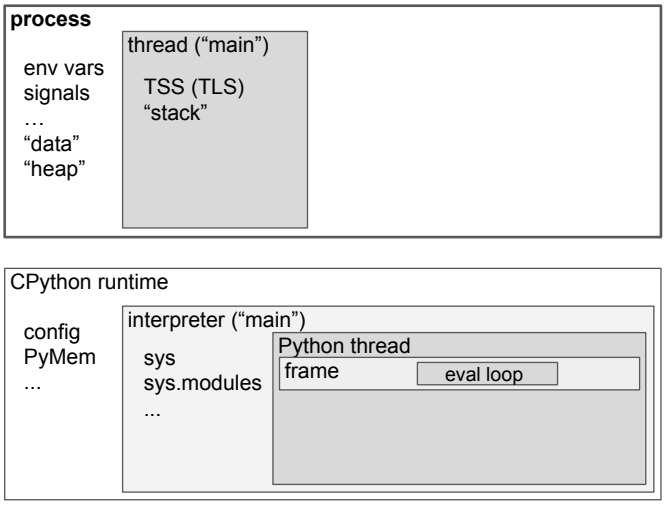
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes



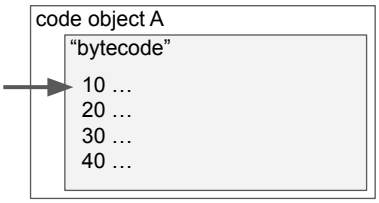
https://bit.ly/2UMMJey

The frame holds the execution state of a running code object.

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 executes one instruction each time through.

As we step through the program, watch how we visit each instruction in a deterministic order.
That is the nature of a single thread.

The Eval Loop

<set up>

for instruction in code object:

    <maybe side-channel stuff>

    <occasionally release & re-acquire the GIL>

    <execute next instruction>

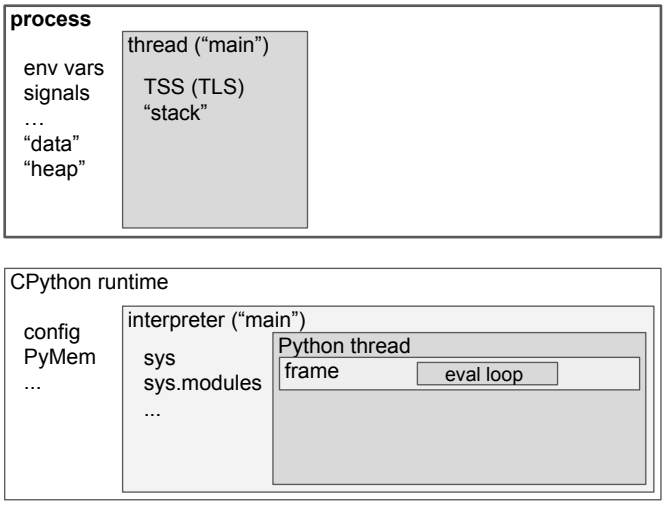
https://bit.ly/2UMMJey

Here's a *very* rough sketch of what happens in the eval loop.

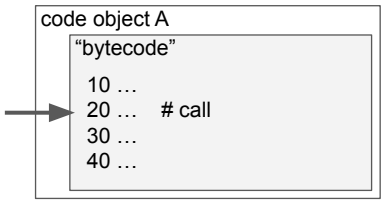
At a high-level it's pretty simple.

You even get a sneak peek here at the driver for this talk: the GIL.

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

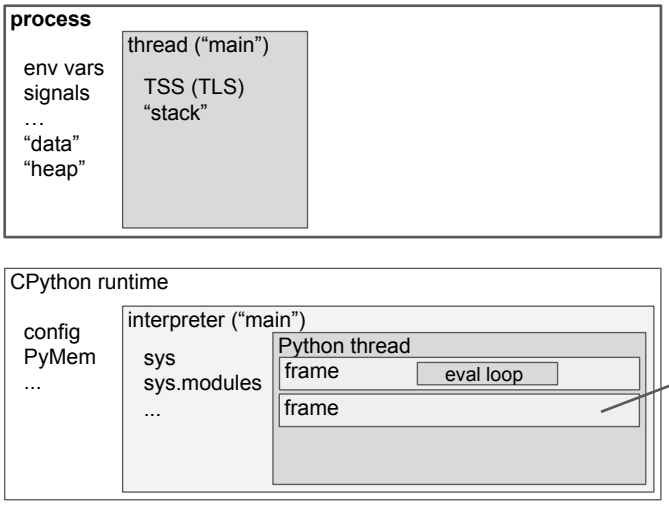


https://bit.ly/2UMMJey

At this point in our code let’s say there is a function call.

Again, there would be a lot more instructions involved in the real sequence of bytecode.

What Happens When Python Runs?

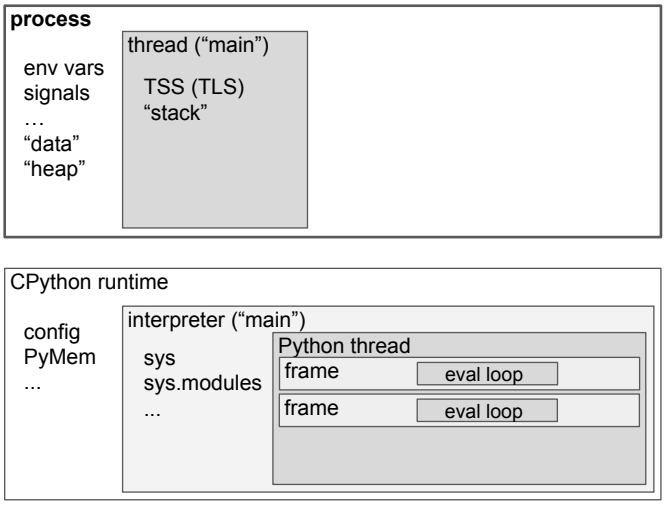


1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

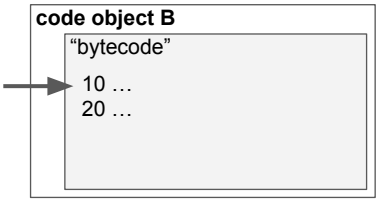
https://bit.ly/2UMMJey

For the call we push another frame onto the call stack in the thread.

What Happens When Python Runs?



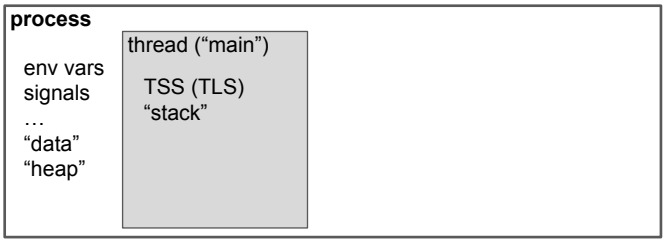
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

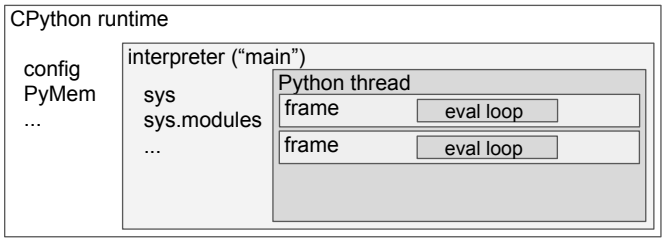


https://bit.ly/2UMMJey

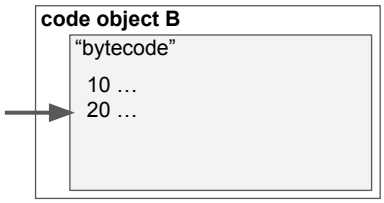
Now we walk through the code for the called function.

What Happens When Python Runs?





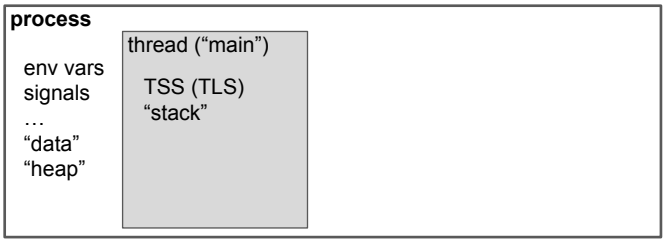
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode

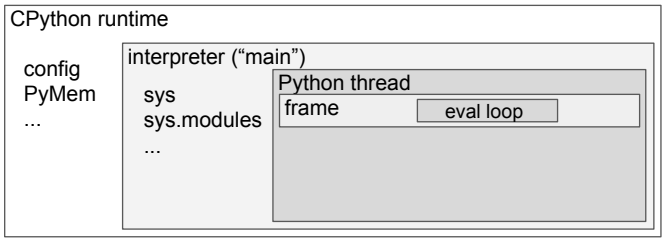


https://bit.ly/2UMMJey

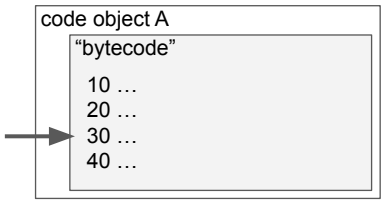
...and this is the last one...

What Happens When Python Runs?





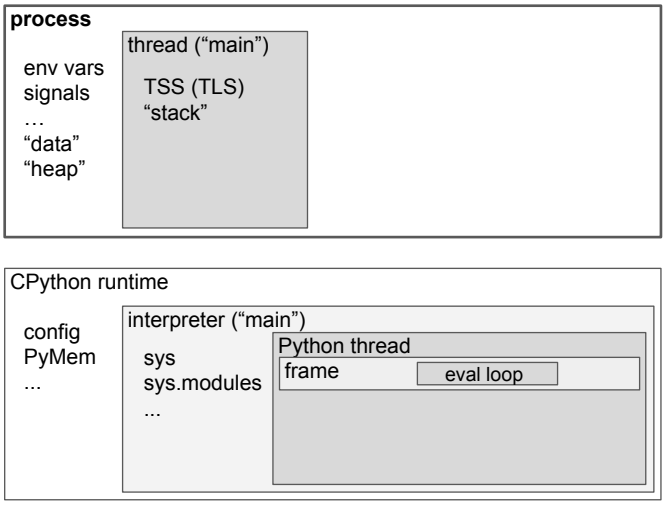
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



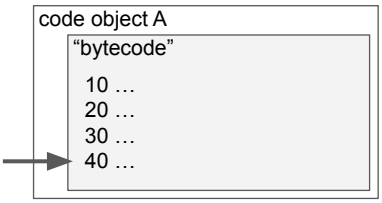
https://bit.ly/2UMMJey

...and now we’re back.

What Happens When Python Runs?



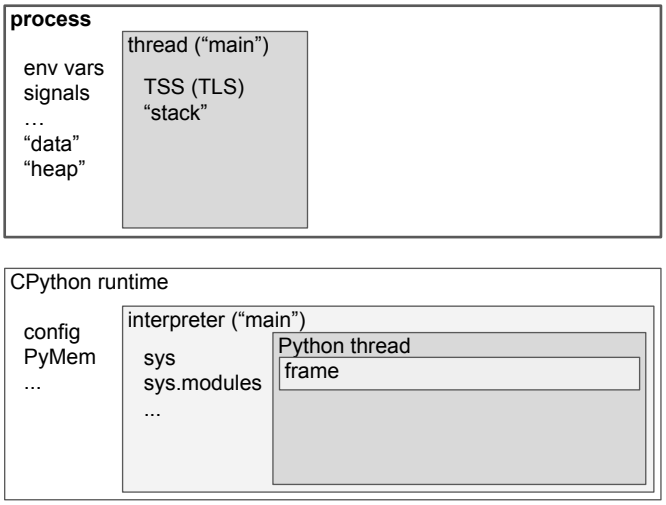
1. process initializes
2. main thread starts
3. Python runtime initializes
   a. main interpreter initializes
   b. main Py thread initializes
4. Python program loads
5. Python frame initializes
6. eval loop steps through bytecode



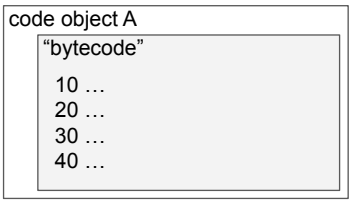
https://bit.ly/2UMMJey

...and we end here...

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

...and now we’re done.
Everything gets cleaned up at this point and the process ends.

What Happens When Python Runs?

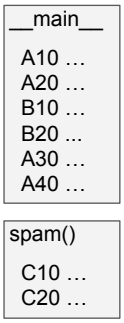
https://bit.ly/2UMMJey

As hinted earlier, we could have deterministically flatten all that execution into a single linear list.

But what happens if there is more than one thread?

Multi-threading!

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

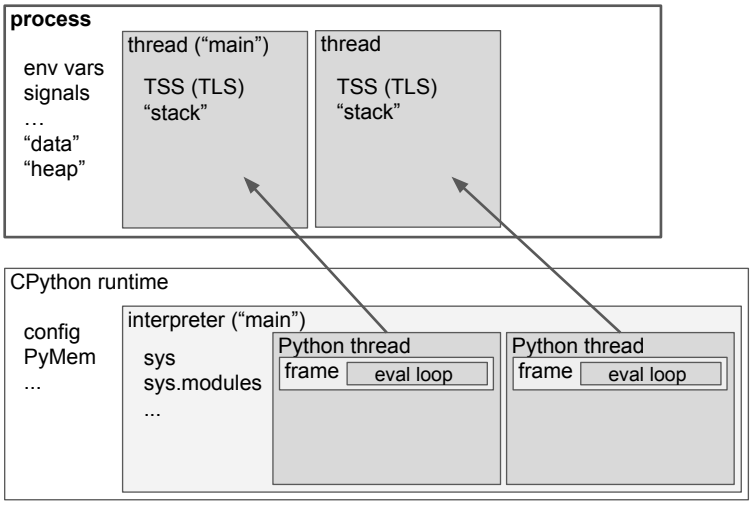


https://bit.ly/2UMMJey

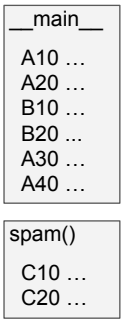
Again, we have a simplified fake program, similar to the first.

Pretend that the program matches the two lists of instructions shown.

Multi-threading!




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

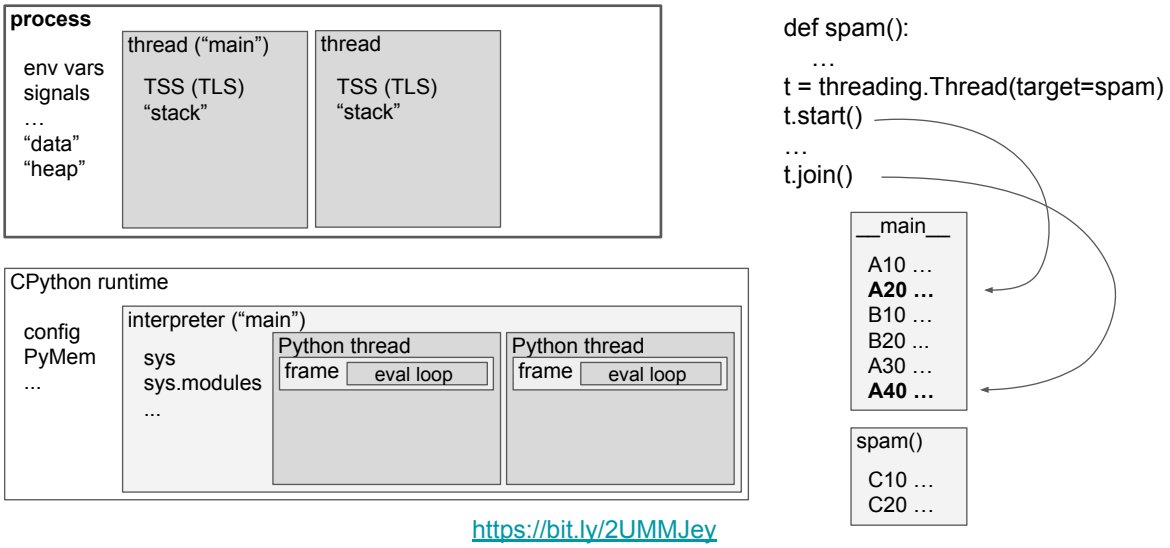


https://bit.ly/2UMMJey

Once the second thread is running we willll see it both in the process and in the Python runtime state.

Each Python thread will be associated with its own OS thread.

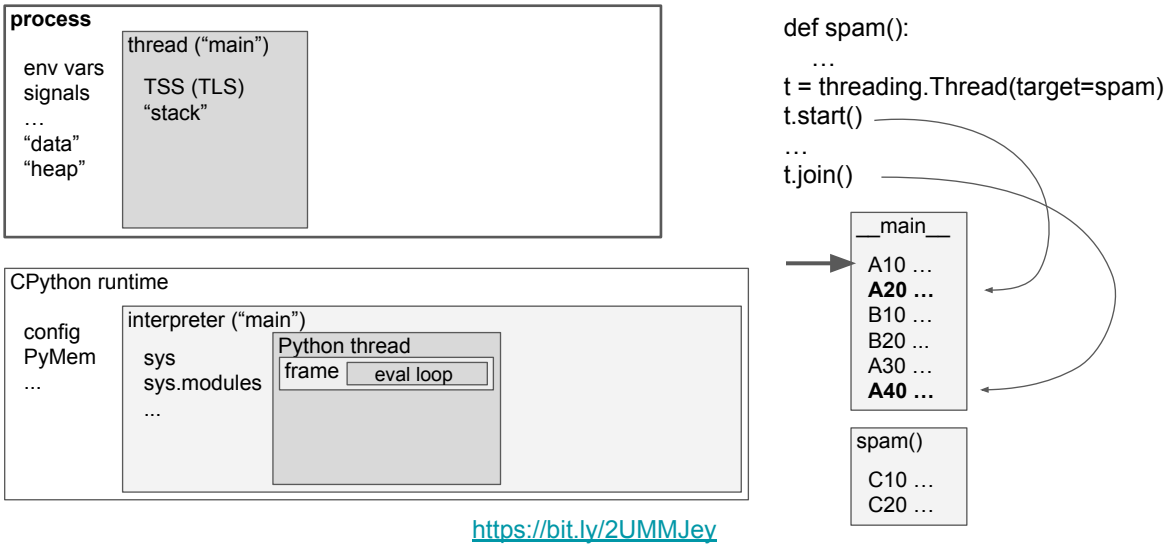
Multi-threading!



https://bit.ly/2UMMJey

In our code the new thread will split off at A20 and we will wait for it at A40.

Multi-threading!

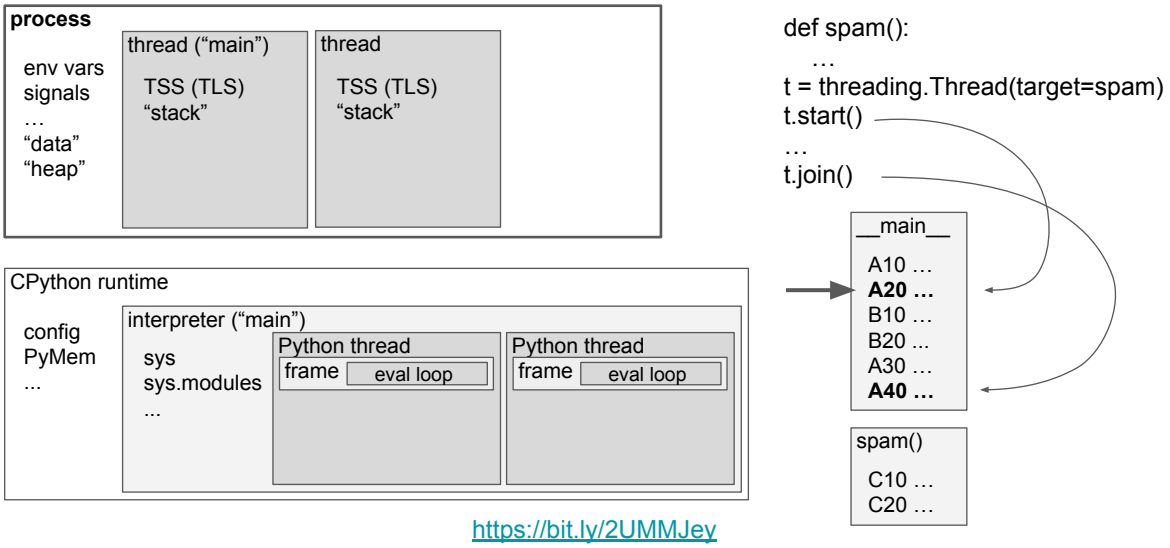


https://bit.ly/2UMMJey

Let's walk through execution.

As we do, try to imagine how we might flatten the code.

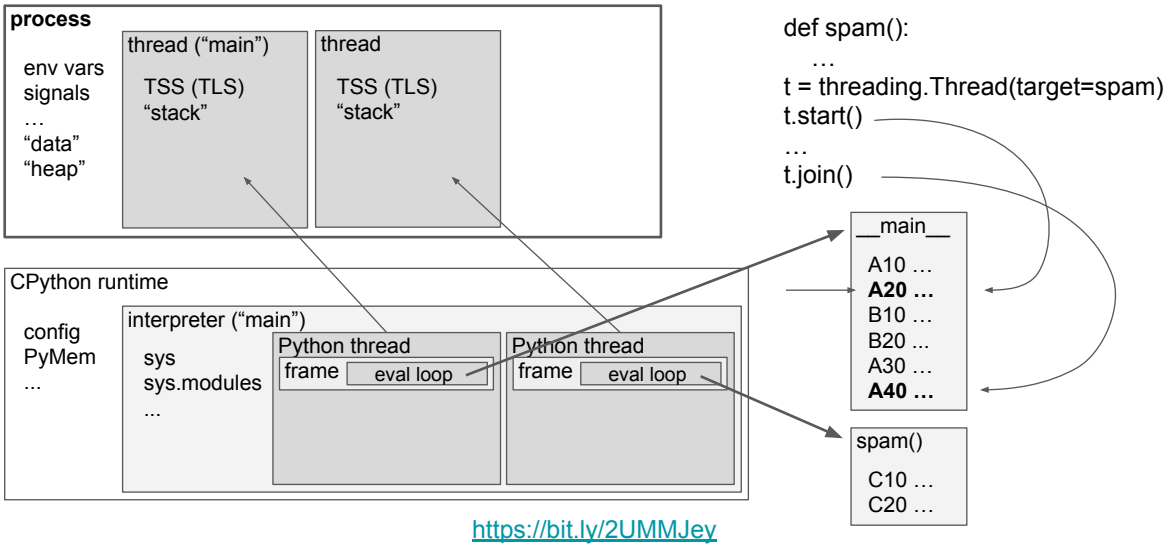
Multi-threading!



https://bit.ly/2UMMJey

At this point the new thread is created.

Multi-threading!

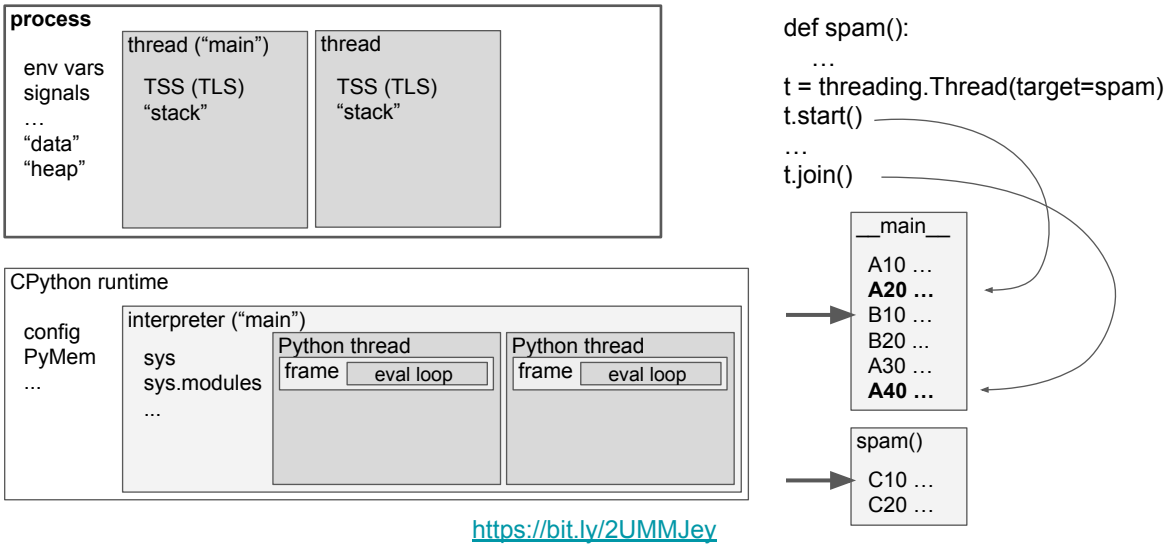


https://bit.ly/2UMMJey

Note that the eval loop in each Python thread is running different code.

On a multi-core processor they will probably be running in parallel (not just concurrently).

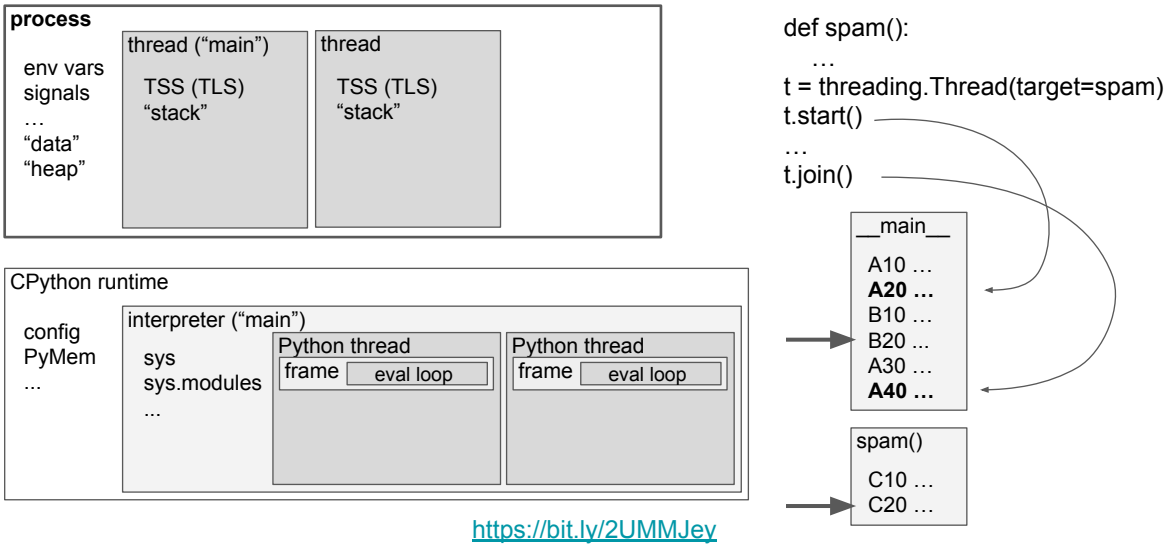
Multi-threading!



https://bit.ly/2UMMJey

So now we get to walk through both instruction sequences at the same time.

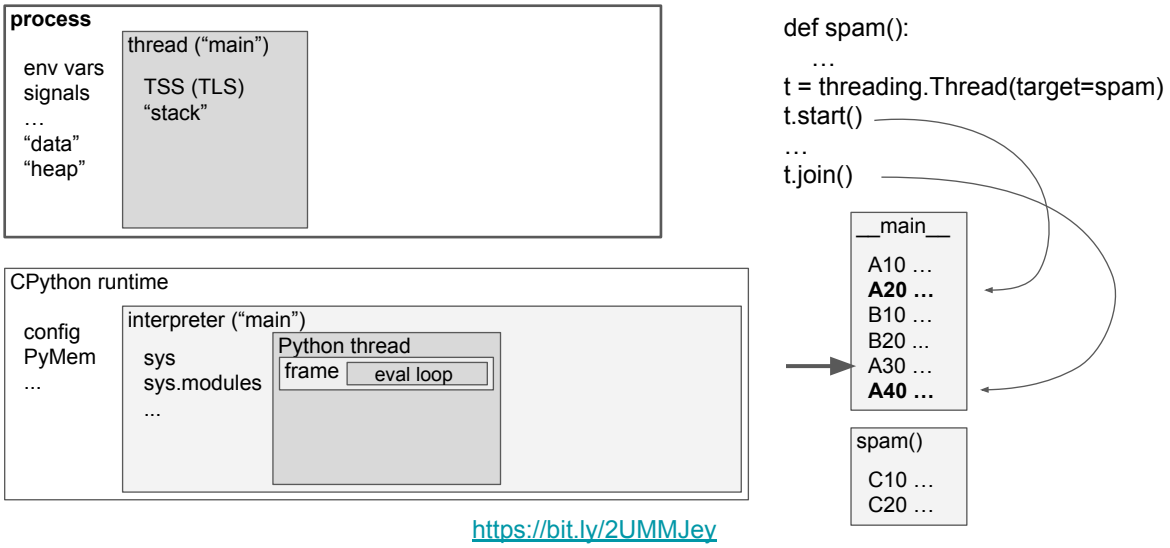
Multi-threading!



https://bit.ly/2UMMJey

Here the second thread is about to finish...

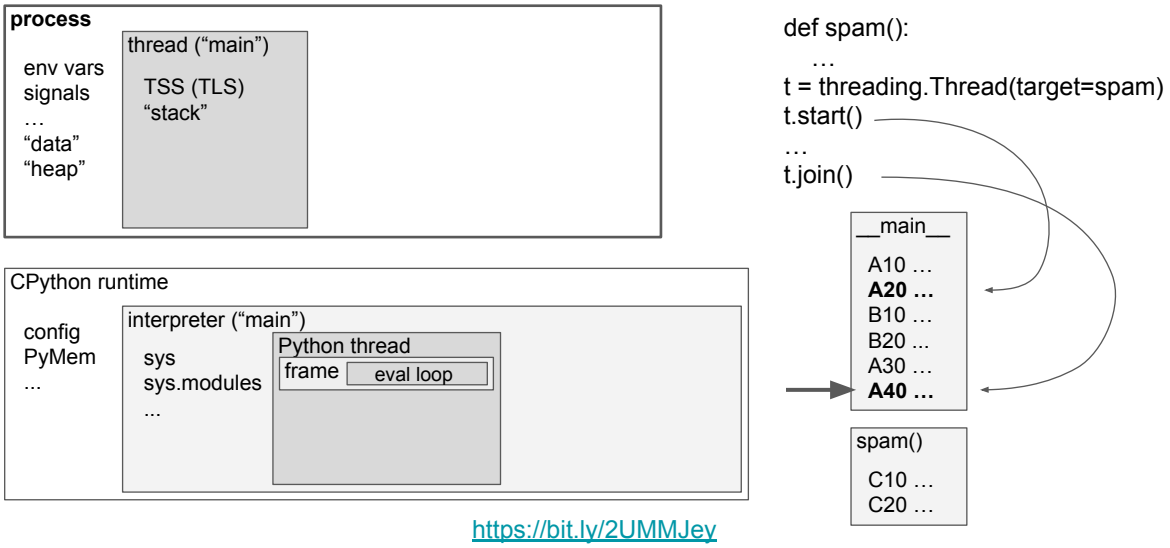
Multi-threading!



https://bit.ly/2UMMJey

The second thread has finished and cleaned up.
However, the original thread keeps going.

Multi-threading!

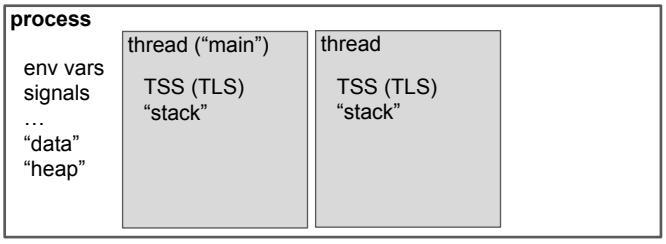


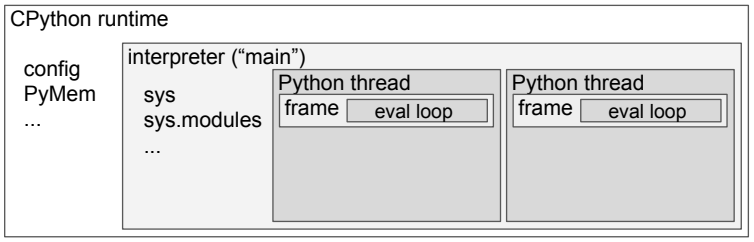
https://bit.ly/2UMMJey

The new thread has already finished so the join has nothing to wait for.

...and we're done.

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

Let's look back at how execution unfolded…

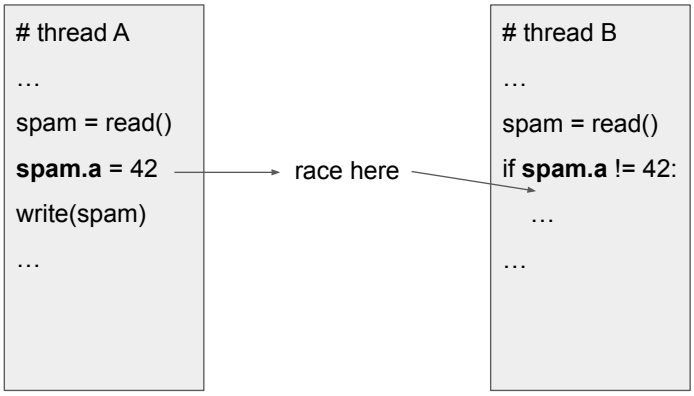
You can’t flatten the code deterministically like you could with the single-threaded code.

Why? Threading is non-deterministic concurrency. So there are many permutations of the instructions. Here we see just four of them.

There is a closely related problem...

“Race Condition”

A.K.A. “Resource Contention”



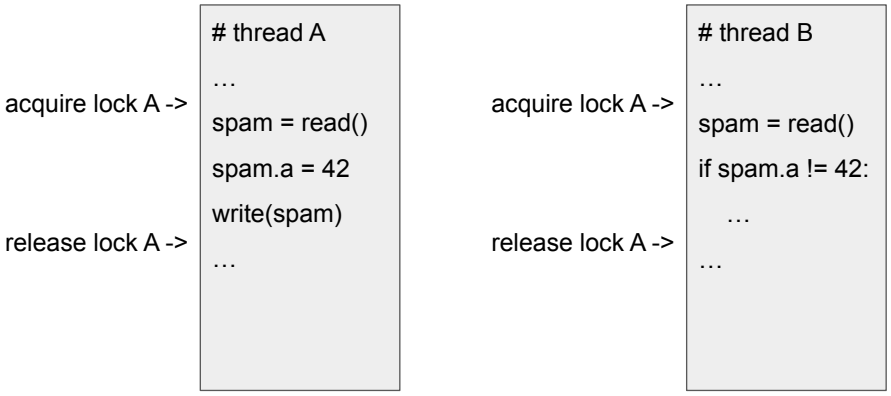
https://bit.ly/2UMMJey

Because of that non-determinism, code in one thread may modify code in another thread unexpectedly.
This is known as a race condition.

Notice how it's possible for thread A to make the assumptions of thread B invalid here.

“Race Condition”

A.K.A. “Resource Contention”



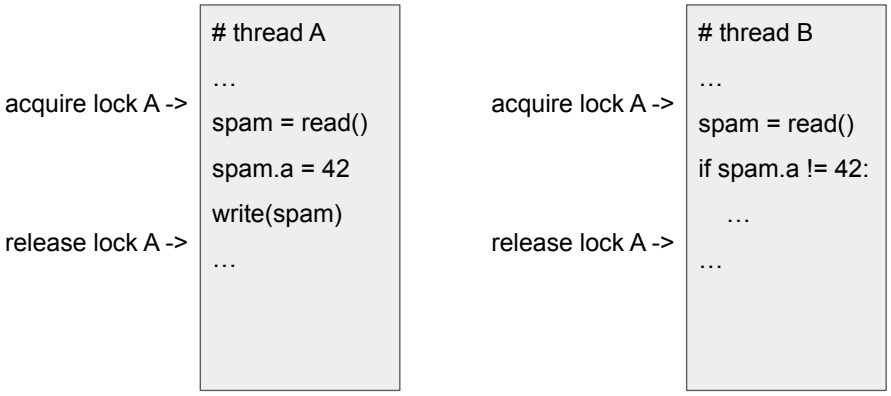
https://bit.ly/2UMMJey

Mostly we solve this by using something called a lock, or mutex, around the critical sections of code!

Locks allow you to synchronize between two threads, thus giving you deterministic order for the locked code.

“Race Condition”

A.K.A. “Resource Contention”



https://bit.ly/2UMMJey

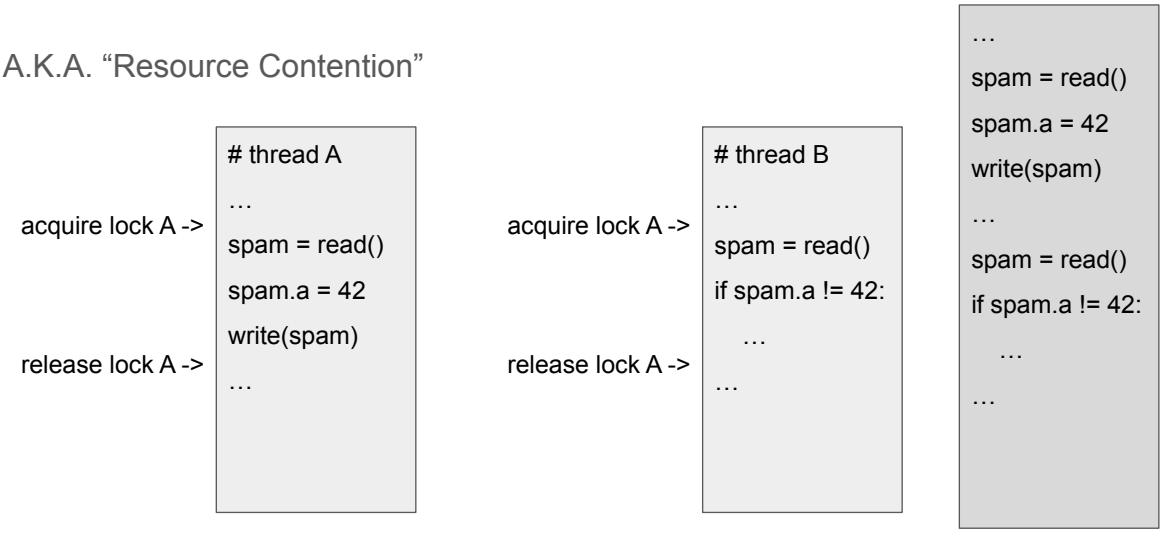
Only one thread can "hold" the lock at a time, which is from the time it is acquired until it is released.

If another thread tries to acquire the lock while it is held then the thread blocks until the lock gets released…

(even if it's the thread that already holds the lock).

“Race Condition”

A.K.A. “Resource Contention”



https://bit.ly/2UMMJey

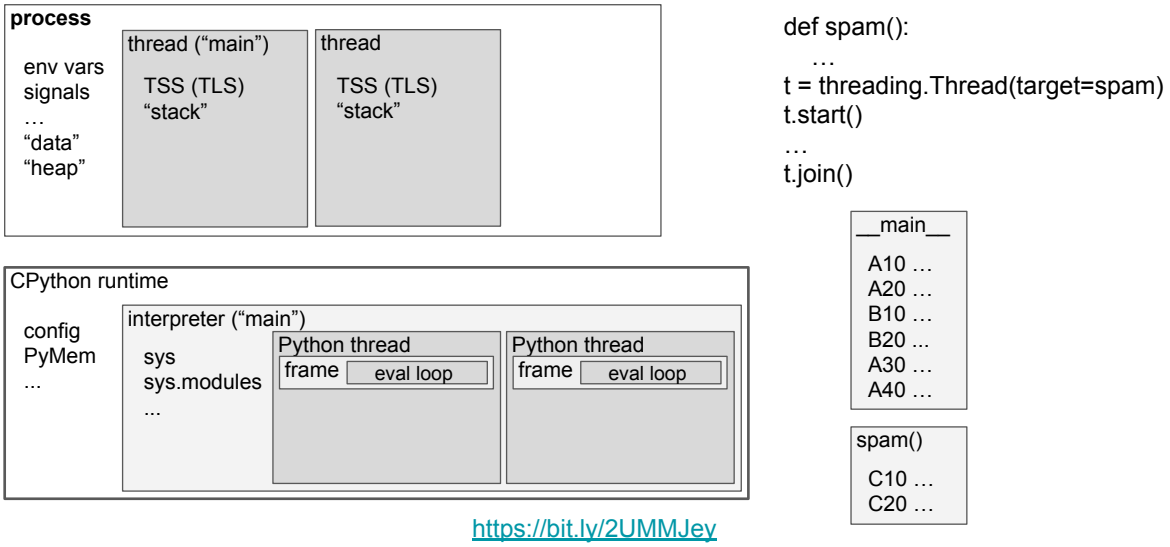
Assuming thread A acquires the lock first, we end up with this actual execution order there on the right.
Notice how the logic of thread A no longer interferes with that of thread B.

If thread B had gotten the lock first then that section would have run first, but the same protection would have taken place.

The GIL

That was a lot. Take a breather for a sec! You've earned it!

The GIL (“Global Interpreter Lock”)




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


https://bit.ly/2UMMJey

The GIL is the lock that protects CPython's runtime resources from races.

If the GIL wasn't there then CPython would not be thread-safe.

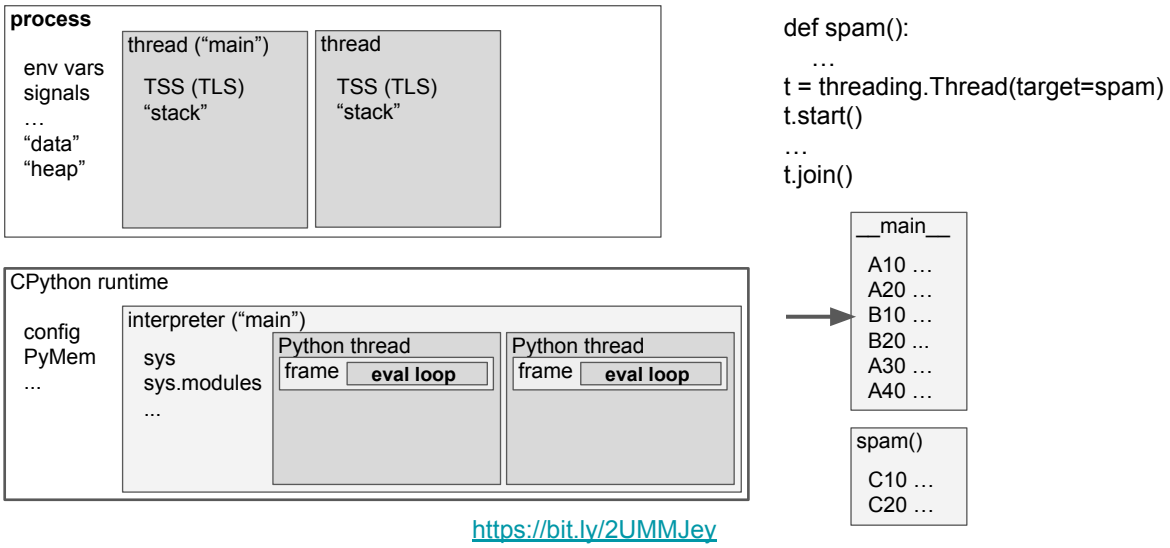
State At Different Layers

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

Here's a look at a lot of the state the GIL is protecting from races...

The GIL (“Global Interpreter Lock”)



https://bit.ly/2UMMJey

The eval loop is the driver for how the GIL guards those resources.

Only one eval loop runs at a time!
See...only one arrow!

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

The Python threads take turns executing a few bytecode instructions in their eval loops.

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

The GIL is also released in blocking situations.

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

Now that we know what the GIL is, let's talk about why we care.

I had trouble thinking of more than this one cost.

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

On the other hand, I could think of several benefits.

Effect and Perception

Who does it really affect?

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

https://bit.ly/2UMMJey

Aside from technical considerations, there's also an unfair perception that the GIL hurts everyone.

I don't want to downplay the pain it inflicts on those that are really impacted, but is the GIL really a problem?

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

So why is it not as big a problem as many people suggest?
Again, the GIL gets released in key situations!

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

Unfortunately, as long as there's a GIL, folk will complain.

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

Folks have a few ways of dealing with the GIL.

(async/await mostly doesn't help here.)

Past Efforts to Remove the GIL

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

https://bit.ly/2UMMJey

There have been a number of attempts to remove the GIL.
Each ran into its own problems, mostly related to the performance of single-threaded code.

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

Also, CPython isn”t the only Python implementation.
Some of the others do not have a GIL and hence already support multi-core Python code!

The Future

Phew!

...

Let's talk about what we're doing about the GIL.

A New C-API

- the history
- the problem
- the solutions

https://bit.ly/2UMMJey

Changes to the C-API are one major way to deal with the GIL.

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 C-API is a great thing, but it's got a handful of problems; some that we've worked at resolving and some we are still working on.

(PEP 384)
(Language summit)

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

To a large extent, the C-API is the main blocker to removing the GIL.
The hard part is that we can't just change the C-API in any way we want.
Backward-compatibility is very important.

The same problem impacts efforts to improve CPython’s performance

The Causes

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

https://bit.ly/2UMMJey

Why do we have these problems?
It’s not like anyone messed up.
We just couldn’t see so far into the future!

Let's look at possible solutions...

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

Just to be clear, nothing is going to change on its own.
This is the nature of open-source. :)

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

Two things that can help are things that we are already working on improving right now.

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

specifically...

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

Here’s how we are starting to layer the C-API so things don’t “leak” as easily.

(
internal - very low-level CPython implementation details
private - names with leading underscore and/or undocumented
unstable - the bulk of the public C-API; may change between major releases
stable - will never change; binary compatible (i.e. memory layout)
)

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

And here are some of the ways we might resolve the C-API problems, particularly for the sake of the GIL.

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>

https://bit.ly/2UMMJey

Quite a few people are involved in the effort to fix problems with the C-API.
Here are some of the projects with which I'm most familiar or involved.

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>

https://bit.ly/2UMMJey

A lot of the discussion happens on capi-sig and python-dev.

Beyond the C-API...

So what else is there?

Subinterpreters !!!

If any of you have ever talked to me then you should have expected this. :)

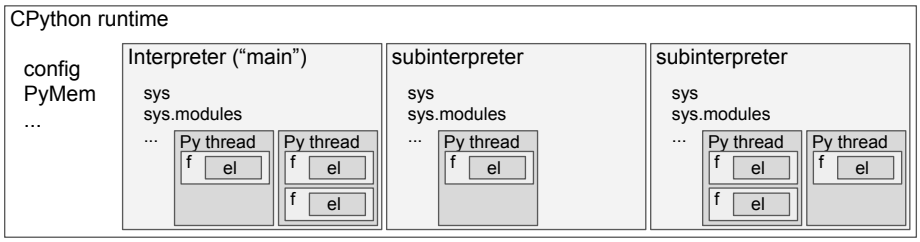
Interpreters in a Single Process

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

“What are subinterpreters?” you ask.
Remember that “interpreter” here is the broader definition.
A subinterpreter is any extra interpreter beyond the main one in the process.

Interpreters in a Single Process

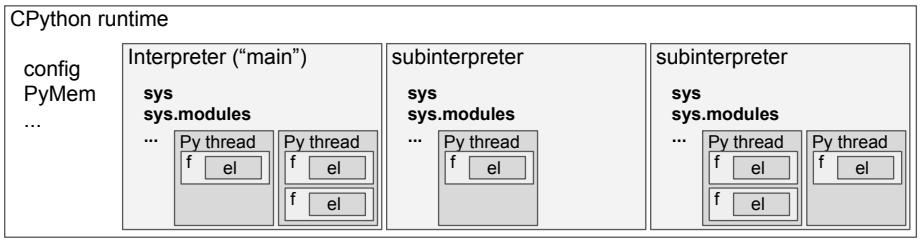
- initial interpreter: “main”
  - has certain responsibilities
- “subinterpreter”: any other interpreter created within the runtime
- isolated-ish



Here i have adapted our earlier diagram.

Interpreters in a Single Process

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



All interpreters are mostly, effectively isolated from one another.
We're working on fixing some corner cases.

It is also worth noting that not all C-extension modules will be able to run in subinterpreters yet, for a number of reasons.

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

Subinterpreters are not a new feature.

The new thing is that we may expose them in the stdlib. :)

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

My PEP, 554, is focused on exposing the existing subinterpreter C-API in a useful, but minimal way.

The proposal is still under consideration for Python 3.9 (in about 2 years) and there's no guarantee it will be accepted...but I'm hopeful.

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

First off, we add a new module to the stdlib that exposes the essential functionality from the C-API.

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""
    print('spam')
"""))
```

https://bit.ly/2UMMJey

Here's a simple example of executing code in a subinterpreter.

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""
    print('spam')
"""))
```

https://bit.ly/2UMMJey

First we create a new subinterpreter.
Nothing gets executed yet.

PEP 554: Example 1

```
import interpreters

interp = interpreters.create()

interp.run(dedent("""
    print('spam')
"""))
```

https://bit.ly/2UMMJey

Then we execute some code.

Note that, for now, we can only execute a text string.

Also note that the code runs in the subinterpreter's __main__ module.

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

Subinterpreters will be most useful when combined with threads.

Here's an example of running in a separate thread.

PEP 554: Example 3

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

https://bit.ly/2UMMJey

In this example we use the same subinterpreter multiple times.

State is preserved between runs in the __main__ module, which can be super useful.

PEP 554: Example 3

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

https://bit.ly/2UMMJey

In the first run we set up some state.

PEP 554: Example 3

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

https://bit.ly/2UMMJey

In the second run we use that state.

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

The C-API only covers creating subinterpreters and running code in them.

At first PEP 554 exposed this functionality and added nothing else.
I quickly found this wasn't quite enough.

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

Isolated interpreters are useful but not nearly as useful as when you can pass data between them safely.
So the PEP adds a minimal mechanism that I've called "channels", based on prior art.

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

Currently only simple, immutable builtin types and singletons are supported in channels.
Also, for now we pass serialized raw data between the interpreters rather than the actual objects.
Finally, channels are currently unbuffered.

Each of these things are addressed by plans for future iterations.

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

Here's an example that generates some data in a thread under the subinterpreter, and processes that data in the main interpreter.

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

First we create the interpreter and the channel.
We get the two ends of the channel as distinctly typed objects.

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

Next we start a thread that will execute some simple code in our subinterpreter.

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

Notice how, when we call "run()", we specify the channel end to inject into the subinterpreter's __main__ module.
We use that channel end in the executed code.

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

Finally we pop the data off the channel in the main interpreter so we can process it there.

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

An important feature of channels is that sending and receiving are both blocking operations.

This is the main way that interpreters can be synchronized.

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

Async/await doesn't fit my brain and I suspect I'm not the only one.
So I'm happy to say that subinterpreters facilitate an alternate concurrency model that I find fits my brain quite well.

I'm talking about Tony Hoare's CSP, "Communicating Sequential Processes", which is the result of significant academic research spanning many decades.

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

A key here is that, unlike traditional threading with its unbounded data sharing, with subinterpreters you get opt-in sharing,

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

Next, even though it isn't perfect, and we're getting better, I love this analogy.
It really matters in a lot of situations.

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

Finally, and to the point for this talk, subinterpreters are a way we can move beyond the GIL.

But how?

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

Remember this chart?
Currently the GIL is part of Python's global runtime state.
If we move it to per-interpreter state then...

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

...we can get multi-core parallelism!

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

I'm a bit hand-wavy here, but making this happen shouldn't be an invasive change. That's actually an important requirement for what I've been working on.

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

All this sounds almost too good to be true!

No one has done it yet because either they didn't know about subinterpreters or they weren't important enough to them.
…
That changed with me!

Regarding the blockers...

the blockers

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

(ノ °□°)ノ ⌒ ┻━┻

https://bit.ly/2UMMJey

There are a couple of blockers that are relatively straight-forward.

the blockers

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

(ノ °□°)ノ ⌒ ┻━┻

https://bit.ly/2UMMJey

Next, if we don't have the GIL to protect global runtime state, what do we do?
We have to minimize global runtime state and guard the remainder with granular locks as appropriate.

the blockers

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

(ノ °□°)ノ ⌒ ┻━┻

https://bit.ly/2UMMJey

Unfortunately, my project has advanced slowly because I simply haven't had enough time.
That happens a lot to open-source volunteers.

I'm grateful to my employer, Microsoft, for giving me significant time each week for open-source work.
This is a relatively recent change for me, but I'm hopeful for some acceleration.

the blockers

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

(ノ °□°)ノ ⌒ ┻━┻

https://bit.ly/2UMMJey

Finally, my old enemy...globals.

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

In C you can have process-global variables.
This applies to both C extensions and CPython itself.

The problem, among others, is that these globals violate the isolation between interpreters, sometimes leading to significant failures during execution.
This is a tricky problem to solve for C-extensions, but we're working on it.

the project

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

https://bit.ly/2UMMJey

Anyway, I’m personally working toward multi-core parallelism!
I'm always looking for folks willing to help.
It also helps that many of my underlying needs coincide with those of other core devs and those of other projects. :)

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

My project entails these three initial efforts….

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

...and most importantly, moving the GIL and other related state into the interpreter level.

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

Here's a rough view of the state that will need to move.

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

One of the neat things about this project is that 90% of the things I need to do are a good idea regardless and overlap with the needs of several other projects.
So even if it doesn't work out, a lot of good things will still come out of it.

What's Next?

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

https://bit.ly/2UMMJey

Here is my plan from here on out.

Once we get the fundamental capability then we'll move on to improving efficiency and adding functionality.

All are welcome to join me!

Thanks!

Thanks!

Questions?

Please join me in the hall there if you'd like to talk. Or find me later. I'll be here until Thursday.

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