

PYCON CHINA 2026

One Bug as a Doorway into CPython

Learning the Interpreter through Free-Threading Bugs

Edward Xu

CPython Contributor / Apache Kvrocks Maintainer

PyCon China 2026

One year ago, I had never read CPython

I knew systems programming. I did not know this system.

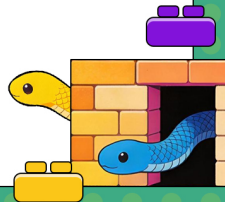
WHAT I BROUGHT

- C++ and systems programming
- Multithreading concepts
- Debugging production systems

WHAT I LACKED

- CPython internals
- CPython's C conventions
- The contributor workflow

I needed a concrete doorway, not a tour of everything.



Free-threading bugs gave me a map

Free threading makes previously GIL-protected assumptions about lifetime and synchronization visible.

Evidence + two stacks + a path into the interpreter

EVIDENCE

A reproducer and a sanitizer report make the failure concrete.

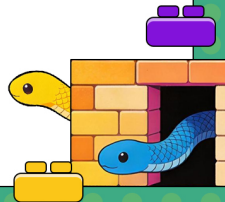
TWO STACKS

The conflicting accesses show exactly where two stories meet.

A SOURCE PATH

Python calls lead into C code, ownership, and an invariant.

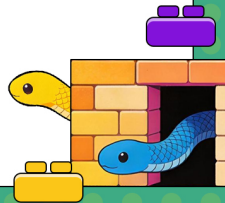
The failure chose where I should start reading.



Case study

A real ThreadSanitizer (TSAN) report from free-threaded CPython.

Let me show you one.



The bug: a race on ThreadHandle.ident

THE CASE

Data race on ThreadHandle.ident

CONFLICTING PATHS

thread_shutdown



ThreadHandle_start

[gh-154937](#) ↗

One observable failure becomes
a path through thread lifecycle
and CPython internals.

ORIGINAL PYTHON REPRODUCER

```
1 import _thread
2 import time
3 from threading import Thread
4
5 def chain1_thread():
6     for _ in range(12000):
7         try:
8             _thread._shutdown()
9         except Exception:
10             pass
11
12 def chain2_thread():
13     for _ in range(12000):
14         try:
15             _thread.start_new_thread(lambda: None, ())
16         except Exception:
17             pass
18         time.sleep(0.0001)
19
20 N_C1 = 1
21 N_C2 = 8
22 threads = [Thread(target=chain1_thread) for _ in range(N_C1)]
23 threads += [Thread(target=chain2_thread) for _ in range(N_C2)]
24 for t in threads: t.start()
25 for t in threads: t.join()
```

Start from the observed behavior

Start from behavior, not architecture.

The first clue is a pair of overlapping calls.

SHUTDOWN PATH

```
_thread._shutdown()
```

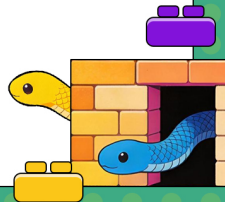


STARTUP PATH

```
_thread.start_new_thread()
```

These are suspects, not yet the root cause.

The question is: **Why does this observable behavior break?**



First, prove the race is real

Run the report unchanged. Preserve the signal before explaining it.

First, prove that the race is real.

ORIGINAL CASE

- 1 **Repeat**
Run the case in a loop
- 2 **Capture**
Keep both stack tops
- 3 **Compare**
Same address and field

THREADSANITIZER SIGNAL

```

1 WARNING: ThreadSanitizer: data race
2   Read by thread T2:
3     #0 thread_shutdown                                _threadmodule.c:2416
4
5   Previous write by thread T1:
6     #0 ThreadHandle_start                             _threadmodule.c:486
7     #1 do_start_new_thread                             _threadmodule.c:1919
8     #2 ...start_joinable_thread                       _threadmodule.c:2042
9
10 SUMMARY: data race in thread_shutdown

```

The invariant signal is one read, one write, and the path connecting each access to Python.

Reproduce now. Interpret next.

Follow both stacks

Follow both stacks before simplifying the Python.

The paths meet at ThreadHandle.ident.

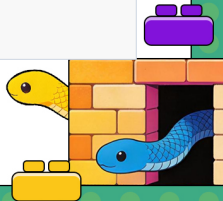
READ PATH

```
_thread._shutdown()  
↓  
thread_shutdown  
↓  
read cur->ident  
while holding HEAD_LOCK
```

WRITE PATH

```
threading.Thread.start()  
↓ _start_joinable_thread  
↓ do_start_new_thread  
↓ add non-daemon handle  
↓ ThreadHandle_start  
write self->ident  
while holding self->mutex
```

Two locks. One field. No mutual exclusion.

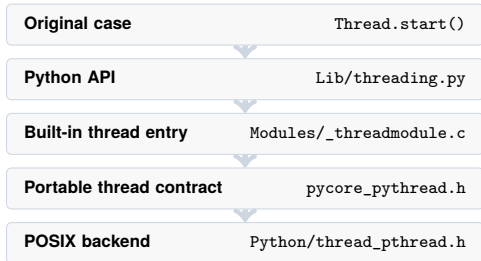


Code map: connect Python to the race

The trace gives us a route through the repository before we rewrite the case.

One Python call opens a vertical slice of CPython.

EXECUTION PATH



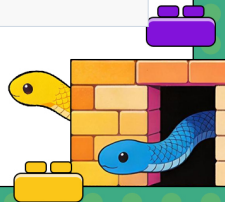
SYNCHRONIZATION SIDE PATH

PyMutex
field-level protection

↓
include/cpython/pylock.h
type and fast path

↓
Python/lock.c
parking and fairness

Why is this hidden race visible now? $\Rightarrow$ **PEP 703: optional GIL**



Validate the whole story end to end

Now the newcomer has enough source context to connect every step.

Can I explain the case from startup to the race?

ROOT PATH FROM THE PR

- 1 **Interpreter startup**
Main thread is non-daemon
- 2 **Thread.start()**
Inherits daemon=False
- 3 **do_start_new_thread**
Adds handle to shutdown_handles
- 4 **Race window**
Startup writes while shutdown reads

WHAT THE PATH GIVES US

BUG ROOT

ident: one read and one write under unrelated locks.

CORE MISMATCH

TSAN STACK

```
_thread.start_joinable_thread
```

ORIGINAL CASE

```
_thread.start_new_thread
```

The stack and the reproducer point to different entry functions.

PR #155085

Wait—this is not what the original case suggests

The Python call and the TSAN stack point to different thread-start APIs.

The two pieces of evidence name different entry functions.

ORIGINAL PYTHON CASE

```
_thread.start_new_thread(  
    lambda: None, ()  
)
```

TSAN STACK

```
#2 start_joinable_thread  
_threadmodule.c:2042
```

`start_new_thread` $\neq$ `start_joinable_thread`

Before choosing a fix, challenge the reproducer.

Something doesn't match

Compare the stack's target function with what the original Python seems to test.

The stack and the original case do not align.

INNER CALL: UNCHANGED

```
1 # Present in both versions
2 _thread.start_new_thread(
3     lambda: None, ()
4 )
```

This detached-thread call was not changed by the diagnostic experiment.

ONLY CHANGE: OUTER THREADS

```
1 # original: inherits non-daemon
2 Thread(target=chain1_thread)
3 Thread(target=chain2_thread)
4
5 # diagnostic edit from the PR
6 Thread(target=chain1_thread, daemon=True)
7 Thread(target=chain2_thread, daemon=True)
```

The later `t.start()` calls remain unchanged; daemon status is set when each `Thread` is constructed.

`Thread(..., daemon=True)` $\Rightarrow$ not added to `shutdown_handles` $\Rightarrow$ race disappears

The write came from the non-daemon `threading.Thread` path, not the unchanged `_thread.start_new_thread()` call.

PR #155085

Build the reproducer from the root cause

Build a new case from the complete root-cause path we just investigated.

FROM UNDERSTANDING TO CASE

- 1 Keep**
The two C access paths
- 2 Remove**
Unrelated Python work
- 3 Control**
The race window
- 4 Verify**
The daemon hypothesis

ROOT-CAUSE-DERIVED REPRODUCER

```
1 phase = Barrier(2)
2 def shutdown_worker():
3     phase.wait()
4     for _ in range(ITERATIONS):
5         try:
6             _thread._shutdown()
7         except RuntimeError:
8             pass
9 def startup_worker():
10    phase.wait()
11    for _ in range(ITERATIONS):
12        handle = _thread.start_joinable_thread(
13            lambda: None, daemon=False
14        )
15        handle.join()
```

Start both outer workers with `daemon=True`; only the intended inner handle enters `shutdown_handles`.

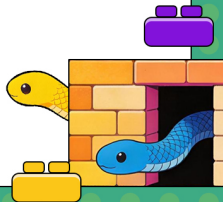
What exactly needs protection?

A race report says *something conflicts*.

It does not tell us *what CPython should guarantee*.

Useful questions:

- Is this an object lifetime problem?
- Is one variable independently shared?
- Do several fields form one invariant?
- Is this synchronization visible in Python semantics?
- Is the current behavior intentionally unsupported?



The warning is evidence, not the contract

Tempting but dangerous

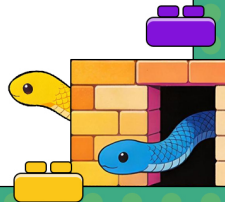
- TSAN reports a variable race
- Make the variable atomic
- The warning disappears

But the logical state may still be inconsistent.

Better question

- What must remain true?
- Who owns this object?
- Which fields change together?
- What does Python code observe?

The fix should protect the semantic invariant.



Find the ownership scope before choosing the tool

I used to ask “Which primitive?” before locating every concurrent accessor.

Who owns this state—and which threads can touch it?



Scope comes first; mutexes, atomics, and STW are choices made afterward.

This is a scope ladder, not containment. If an invariant spans the runtime, STW may be the clearest boundary.

The existing accessor already encodes the ownership rule

Make the reader use the existing protected accessor.

```
thread_shutdown(...)  
  
- if (cur->ident != ident) {  
  
+ if (ThreadHandle_ident(cur) != ident) {
```

DOCUMENTED

ident is guarded by
ThreadHandle.mutex.

REUSED

ThreadHandle_ident() already
takes that mutex.

BOUNDED

No new global lock or broader
synchronization promise.

Merged in CPython PR #155085

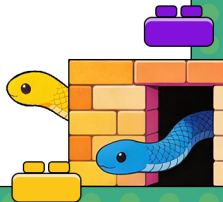
Did the patch validate our model?

A patch is also an experiment.

It tests whether my mental model was right.

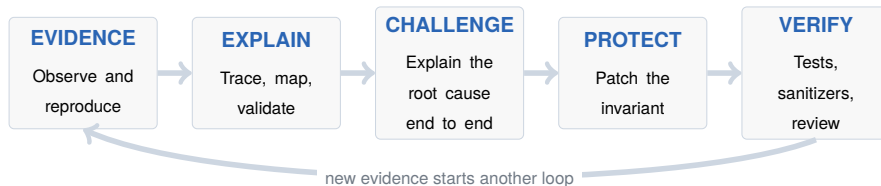
Validation layers:

- the original reproducer no longer fails;
- the regression test fails before the patch and passes after it;
- relevant CPython tests pass;
- sanitizer build is clean for this case;
- review agrees with the invariant and the fix boundary.



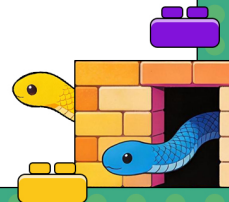
One investigation becomes a reusable method

The details change; the reasoning loop stays the same.



Review is part of verification.

"Review is where assumptions meet another perspective."



After several loops

Repeated investigations turn isolated code paths into a system map.

You start recognizing parts of the interpreter.

TAGGED POINTERS

`_PyStackRef` carries pointers or tagged integers with reference-state bits.

BIASED REFCOUNTING

The owner updates a local count; other threads use the shared count.

CYCLIC GC

Collection coordinates threads to inspect a consistent object graph.

One bug teaches a function; several bugs reveal the architecture.

During each investigation, you follow source history, PEPs, and the discussions behind them. Looking back, you realize that each question has taken you one step deeper into CPython's internals.

Next challenge: no stack trace

After several bug fixes, choose a performance problem.

This time, you have to draw the map.

WHAT DISAPPEARS

No sanitizer stack
No named conflicting field
No ready-made source path

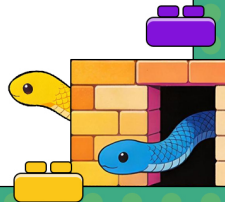
The symptom may be only latency, throughput, or contention.

WALK THE SYSTEM YOURSELF

- 1 **Benchmark** Make the cost repeatable
- 2 **Predict** Name the layer responsible
- 3 **Instrument** Profile and add counters
- 4 **Change** Move one boundary
- 5 **Validate** Accept or reject the model

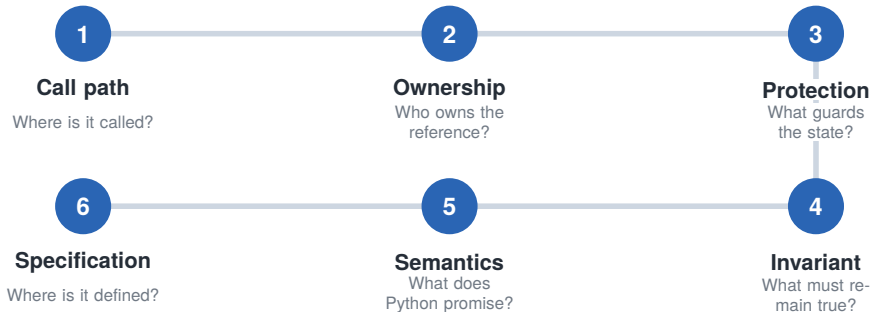
Performance work tests whether the source map is now yours.

If you are interested, see this journey in [CPython PR #141596](#).



From local bug to architecture

Follow each answer into the next, deeper question.



The questions deepen as the model improves.



My takeaway: we need clearer concurrency contracts

After several race fixes, I could explain the implementation—but not always the promise.

What execution order may Python code rely on?

IMPLEMENTATION ANSWERS

`memory_order` defines internal ordering.
Locks and STW protect selected invariants.
Clean TSAN: the observed data race is gone.

LANGUAGE-LEVEL QUESTIONS

What may cross threads?
When do writes become visible?
What establishes a happens-before relation?
Which outcomes should be deterministic?
Python promise or CPython behavior?

A POSSIBLE DESIGN DIRECTION

Provide one high-level concurrency model across GIL-enabled and free-threaded builds—without forcing most application developers to manage low-level synchronization primitives.

Draft [PEP 805—Safe Parallel Python](#) proposes race-free-by-default parallel execution and gradual migration.

The [earlier design discussion](#) shows how the proposal evolved.

A broader engineering philosophy

This is not only about CPython.

System	Observable evidence
CPython race	TSAN report
Database	incorrect query result
Distributed system	production incident
Application	user-reported bug

Different evidence. Same loop: observe, reduce, model, validate, learn.

Why this matters for newcomers

You do not need to understand all of CPython before contributing to CPython.

You need:

- one observable problem;
- enough patience to reduce it;
- enough curiosity to follow the stack;
- enough discipline to validate your model;
- enough humility to let review change your understanding.

Contribution can be the path to understanding, not only the result of understanding.

Call to action

Try the free-threaded build.

Run your tests.

Break something.

Reduce it.

Report it.

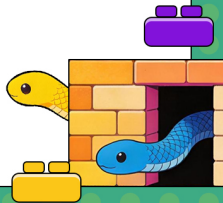
Maybe fix it.

One bug can be your doorway.

Resources

Everything here is ready to open and explore.

- **CPython Developer Guide**
- **PEP 703: Making the GIL Optional in CPython**
- **CPython issues labeled `topic-free-threading`**
- **Build CPython with ThreadSanitizer**
- **Race case study: CPython PR #155085**
- **Performance case study: CPython PR #141596**
- **Safe Parallel Python design discussion**



Thank you!

Questions?

CONTACT

Find me on GitHub.

GITHUB



github.com/LindaSummer

One bug can be your doorway into CPython.