C API Extension Support for Free Threading

Version 3.14.0rc3

Guido van Rossum and the Python development team

octobre 01, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

1 Identifying the Free-Threaded Build in C **2**

2 Module Initialization **2**
2.1 Multi-Phase Initialization . . . 2
2.2 Single-Phase Initialization . . . 2

3 General API Guidelines **3**
3.1 Container Thread Safety . . . 3

4 Borrowed References **3**

5 Memory Allocation APIs **4**

6 Thread State and GIL APIs **4**

7 Protecting Internal Extension State **4**

8 Critical Sections **5**
8.1 What Are Critical Sections ? . . . 5
8.2 Using Critical Sections . . . 5
8.3 How Critical Sections Work . . . 5
8.4 Deadlock Avoidance . . . 5
8.5 Important Considerations . . . 6

9 Building Extensions for the Free-Threaded Build **6**
9.1 Limited C API and Stable ABI . . . 6
9.2 Windows . . . 6

Starting with the 3.13 release, CPython has support for running with the global interpreter lock (GIL) disabled in a configuration called free threading. This document describes how to adapt C API extensions to support free threading.

1 Identifying the Free-Threaded Build in C

The CPython C API exposes the `Py_GIL_DISABLED` macro : in the free-threaded build it's defined to `1`, and in the regular build it's not defined. You can use it to enable code that only runs under the free-threaded build :

```
#ifdef Py_GIL_DISABLED
/* code that only runs in the free-threaded build */
#endif
```

Note

On Windows, this macro is not defined automatically, but must be specified to the compiler when building. The `sysconfig.get_config_var()` function can be used to determine whether the current running interpreter had the macro defined.

2 Module Initialization

Extension modules need to explicitly indicate that they support running with the GIL disabled ; otherwise importing the extension will raise a warning and enable the GIL at runtime.

There are two ways to indicate that an extension module supports running with the GIL disabled depending on whether the extension uses multi-phase or single-phase initialization.

2.1 Multi-Phase Initialization

Extensions that use multi-phase initialization (i.e., `PyModuleDef_Init()`) should add a `Py_mod_gil` slot in the module definition. If your extension supports older versions of CPython, you should guard the slot with a `PY_VERSION_HEX` check.

```
static struct PyModuleDef_Slot module_slots[] = {
    ...
#if PY_VERSION_HEX >= 0x030D0000
    {Py_mod_gil, Py_MOD_GIL_NOT_USED},
#endif
    {0, NULL}
};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    .m_slots = module_slots,
    ...
};
```

2.2 Single-Phase Initialization

Extensions that use single-phase initialization (i.e., `PyModule_Create()`) should call `PyUnstable_Module_SetGIL()` to indicate that they support running with the GIL disabled. The function is only defined in the free-threaded build, so you should guard the call with `#ifdef Py_GIL_DISABLED` to avoid compilation errors in the regular build.

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    ...
};

PyMODINIT_FUNC
```

(suite sur la page suivante)

(suite de la page précédente)

```
PyInit_mymodule(void)
{
    PyObject *m = PyModule_Create(&moduledef);
    if (m == NULL) {
        return NULL;
    }
#ifdef Py_GIL_DISABLED
    PyUnstable_Module_SetGIL(m, Py_MOD_GIL_NOT_USED);
#endif
    return m;
}
```

3 General API Guidelines

Most of the C API is thread-safe, but there are some exceptions.

- **Struct Fields** : Accessing fields in Python C API objects or structs directly is not thread-safe if the field may be concurrently modified.
- **Macros** : Accessor macros like `PyList_GET_ITEM`, `PyList_SET_ITEM`, and macros like `PySequence_Fast_GET_SIZE` that use the object returned by `PySequence_Fast()` do not perform any error checking or locking. These macros are not thread-safe if the container object may be modified concurrently.
- **Borrowed References** : C API functions that return borrowed references may not be thread-safe if the containing object is modified concurrently. See the section on *borrowed references* for more information.

3.1 Container Thread Safety

Containers like `PyListObject`, `PyDictObject`, and `PySetObject` perform internal locking in the free-threaded build. For example, the `PyList_Append()` will lock the list before appending an item.

PyDict_Next

A notable exception is `PyDict_Next()`, which does not lock the dictionary. You should use `Py_BEGIN_CRITICAL_SECTION` to protect the dictionary while iterating over it if the dictionary may be concurrently modified :

```
Py_BEGIN_CRITICAL_SECTION(dict);
PyObject *key, *value;
Py_ssize_t pos = 0;
while (PyDict_Next(dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

4 Borrowed References

Some C API functions return borrowed references. These APIs are not thread-safe if the containing object is modified concurrently. For example, it's not safe to use `PyList_GetItem()` if the list may be modified concurrently.

The following table lists some borrowed reference APIs and their replacements that return strong references.

Borrowed reference API	Strong reference API
`PyList_GetItem()`	`PyList_GetItemRef()`
`PyList_GET_ITEM()`	`PyList_GetItemRef()`
`PyDict_GetItem()`	`PyDict_GetItemRef()`
`PyDict_GetItemWithError()`	`PyDict_GetItemRef()`
`PyDict_GetItemString()`	`PyDict_GetItemStringRef()`
`PyDict_SetDefault()`	`PyDict_SetDefaultRef()`
`PyDict_Next()`	none (see *PyDict_Next*)
`PyWeakref_GetObject()`	`PyWeakref_GetRef()`
`PyWeakref_GET_OBJECT()`	`PyWeakref_GetRef()`
`PyImport_AddModule()`	`PyImport_AddModuleRef()`
`PyCell_GET()`	`PyCell_Get()`

Not all APIs that return borrowed references are problematic. For example, `PyTuple_GetItem()` is safe because tuples are immutable. Similarly, not all uses of the above APIs are problematic. For example, `PyDict_GetItem()` is often used for parsing keyword argument dictionaries in function calls; those keyword argument dictionaries are effectively private (not accessible by other threads), so using borrowed references in that context is safe.

Some of these functions were added in Python 3.13. You can use the pythoncapi-compat package to provide implementations of these functions for older Python versions.

5 Memory Allocation APIs

Python's memory management C API provides functions in three different allocation domains : "raw", "mem", and "object". For thread-safety, the free-threaded build requires that only Python objects are allocated using the object domain, and that all Python object are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

> **Note**
>
> Search for uses of `PyObject_Malloc()` in your extension and check that the allocated memory is used for Python objects. Use `PyMem_Malloc()` to allocate buffers instead of `PyObject_Malloc()`.

6 Thread State and GIL APIs

Python provides a set of functions and macros to manage thread state and the GIL, such as :

- `PyGILState_Ensure()` and `PyGILState_Release()`
- `PyEval_SaveThread()` and `PyEval_RestoreThread()`
- `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`

These functions should still be used in the free-threaded build to manage thread state even when the GIL is disabled. For example, if you create a thread outside of Python, you must call `PyGILState_Ensure()` before calling into the Python API to ensure that the thread has a valid Python thread state.

You should continue to call `PyEval_SaveThread()` or `Py_BEGIN_ALLOW_THREADS` around blocking operations, such as I/O or lock acquisitions, to allow other threads to run the cyclic garbage collector.

7 Protecting Internal Extension State

Your extension may have internal state that was previously protected by the GIL. You may need to add locking to protect this state. The approach will depend on your extension, but some common patterns include :

- **Caches** : global caches are a common source of shared state. Consider using a lock to protect the cache or disabling it in the free-threaded build if the cache is not critical for performance.
- **Global State** : global state may need to be protected by a lock or moved to thread local storage. C11 and C++11 provide the `thread_local` or `_Thread_local` for thread-local storage.

8 Critical Sections

In the free-threaded build, CPython provides a mechanism called ”critical sections” to protect data that would otherwise be protected by the GIL. While extension authors may not interact with the internal critical section implementation directly, understanding their behavior is crucial when using certain C API functions or managing shared state in the free-threaded build.

8.1 What Are Critical Sections ?

Conceptually, critical sections act as a deadlock avoidance layer built on top of simple mutexes. Each thread maintains a stack of active critical sections. When a thread needs to acquire a lock associated with a critical section (e.g., implicitly when calling a thread-safe C API function like `PyDict_SetItem()`, or explicitly using macros), it attempts to acquire the underlying mutex.

8.2 Using Critical Sections

The primary APIs for using critical sections are :

- `Py_BEGIN_CRITICAL_SECTION` and `Py_END_CRITICAL_SECTION` - For locking a single object
- `Py_BEGIN_CRITICAL_SECTION2` and `Py_END_CRITICAL_SECTION2` - For locking two objects simultaneously

These macros must be used in matching pairs and must appear in the same C scope, since they establish a new local scope. These macros are no-ops in non-free-threaded builds, so they can be safely added to code that needs to support both build types.

A common use of a critical section would be to lock an object while accessing an internal attribute of it. For example, if an extension type has an internal count field, you could use a critical section while reading or writing that field :

```
// read the count, returns new reference to internal count value
PyObject *result;
Py_BEGIN_CRITICAL_SECTION(obj);
result = Py_NewRef(obj->count);
Py_END_CRITICAL_SECTION();
return result;

// write the count, consumes reference from new_count
Py_BEGIN_CRITICAL_SECTION(obj);
obj->count = new_count;
Py_END_CRITICAL_SECTION();
```

8.3 How Critical Sections Work

Unlike traditional locks, critical sections do not guarantee exclusive access throughout their entire duration. If a thread would block while holding a critical section (e.g., by acquiring another lock or performing I/O), the critical section is temporarily suspended—all locks are released—and then resumed when the blocking operation completes.

This behavior is similar to what happens with the GIL when a thread makes a blocking call. The key differences are :

- Critical sections operate on a per-object basis rather than globally
- Critical sections follow a stack discipline within each thread (the ”begin” and ”end” macros enforce this since they must be paired and within the same scope)
- Critical sections automatically release and reacquire locks around potential blocking operations

8.4 Deadlock Avoidance

Critical sections help avoid deadlocks in two ways :

1. If a thread tries to acquire a lock that’s already held by another thread, it first suspends all of its active critical sections, temporarily releasing their locks
2. When the blocking operation completes, only the top-most critical section is reacquired first

This means you cannot rely on nested critical sections to lock multiple objects at once, as the inner critical section may suspend the outer ones. Instead, use `Py_BEGIN_CRITICAL_SECTION2` to lock two objects simultaneously.

Note that the locks described above are only `PyMutex` based locks. The critical section implementation does not know about or affect other locking mechanisms that might be in use, like POSIX mutexes. Also note that while blocking on any `PyMutex` causes the critical sections to be suspended, only the mutexes that are part of the critical sections are released. If `PyMutex` is used without a critical section, it will not be released and therefore does not get the same deadlock avoidance.

8.5 Important Considerations

- Critical sections may temporarily release their locks, allowing other threads to modify the protected data. Be careful about making assumptions about the state of the data after operations that might block.
- Because locks can be temporarily released (suspended), entering a critical section does not guarantee exclusive access to the protected resource throughout the section's duration. If code within a critical section calls another function that blocks (e.g., acquires another lock, performs blocking I/O), all locks held by the thread via critical sections will be released. This is similar to how the GIL can be released during blocking calls.
- Only the lock(s) associated with the most recently entered (top-most) critical section are guaranteed to be held at any given time. Locks for outer, nested critical sections might have been suspended.
- You can lock at most two objects simultaneously with these APIs. If you need to lock more objects, you'll need to restructure your code.
- While critical sections will not deadlock if you attempt to lock the same object twice, they are less efficient than purpose-built reentrant locks for this use case.
- When using `Py_BEGIN_CRITICAL_SECTION2`, the order of the objects doesn't affect correctness (the implementation handles deadlock avoidance), but it's good practice to always lock objects in a consistent order.
- Remember that the critical section macros are primarily for protecting access to *Python objects* that might be involved in internal CPython operations susceptible to the deadlock scenarios described above. For protecting purely internal extension state, standard mutexes or other synchronization primitives might be more appropriate.

9 Building Extensions for the Free-Threaded Build

C API extensions need to be built specifically for the free-threaded build. The wheels, shared libraries, and binaries are indicated by a `t` suffix.

- pypa/manylinux supports the free-threaded build, with the `t` suffix, such as `python3.13t`.
- pypa/cibuildwheel supports the free-threaded build if you set CIBW_ENABLE to cpython-freethreading.

9.1 Limited C API and Stable ABI

The free-threaded build does not currently support the Limited C API or the stable ABI. If you use setuptools to build your extension and currently set `py_limited_api=True` you can use `py_limited_api=not sysconfig.get_config_var("Py_GIL_DISABLED")` to opt out of the limited API when building with the free-threaded build.

> **Note**
>
> You will need to build separate wheels specifically for the free-threaded build. If you currently use the stable ABI, you can continue to build a single wheel for multiple non-free-threaded Python versions.

9.2 Windows

Due to a limitation of the official Windows installer, you will need to manually define `Py_GIL_DISABLED=1` when building extensions from source.

Voir aussi

Porting Extension Modules to Support Free-Threading : A community-maintained porting guide for extension authors.