C API Extension Support for Free Threading

Release 3.13.0rc2

Guido van Rossum and the Python development team

September 25, 2024

Python Software Foundation
Email: docs@python.org

Contents

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 **5**

8 Building Extensions for the Free-Threaded Build **5**
8.1 Limited C API and Stable ABI . . . 5
8.2 Windows . . . 5

Starting with the 3.13 release, CPython has experimental 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
```

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
PyInit_mymodule(void)
{
    PyObject *m = PyModule_Create(&moduledef);
    if (m == NULL) {
        return NULL;
```

(continues on next page)

(continued from previous page)

```
    }
#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` and `PyList_SET_ITEM` 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()`
`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()`

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 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_FREE_THREADED_SUPPORT.

8.1 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.

8.2 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.

See also

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