Synchronization primitives

- Python critical section API
  - Low-level critical section API
- Legacy locking APIs

The C-API provides a basic mutual exclusion lock.

> A mutual exclusion lock. The `!PyMutex` should be initialized to zero to represent the unlocked state. For example:
>
> ```
> PyMutex mutex = {0};
> ```
>
> Instances of `!PyMutex` should not be copied or moved. Both the contents and address of a `!PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.
>
> > [!NOTE] A `!PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.
>
> 3.13

> Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily detach the `thread state <attached thread state>` if one exists.
>
> 3.13

>

> Unlock mutex *m*. The mutex must be locked --- otherwise, the function will issue a fatal error.
>
> 3.13

> Returns non-zero if the mutex *m* is currently locked, zero otherwise.
>
> > [!NOTE] This function is intended for use in assertions and debugging only and should not be used to make concurrency control decisions, as the lock state may change immediately after the check.
>
> 3.14

Python critical section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for `free-threaded <free threading>` CPython. They are intended to replace reliance on the `global interpreter lock`, and are no-ops in versions of Python with the global interpreter lock.

Critical sections are intended to be used for custom types implemented in C-API extensions. They should generally not be used with built-in types like `list` and `dict` because their public C-APIs already use critical sections internally, with the notable exception of `PyDict_Next`, which requires critical section to be acquired externally.

Critical sections avoid deadlocks by implicitly suspending active critical sections, hence, they do not provide exclusive access such as provided by traditional locks like `PyMutex`. When a critical section is started, the per-object lock for the object is acquired. If the code executed inside the critical section calls C-API functions then it can suspend the critical section thereby releasing the per-object lock, so other threads can acquire the per-object lock for the same object.

Variants that accept `PyMutex` pointers rather than Python objects are also available. Use these variants to start a critical section in a situation where there is no `PyObject` -- for example, when working with a C type that does not extend or wrap `PyObject` but still needs to call into the C API in a manner that might lead to deadlocks.

>

> [!NOTE] Operations that need to lock two objects at once must use `Py_BEGIN_CRITICAL_SECTION2`. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Example usage:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, `Py_SETREF` calls `Py_DECREF`, which can call arbitrary code through an object's deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls `PyEval_SaveThread`.

> “ Acquires the per-object lock for the object *op* and begins a critical section.
>
> In the free-threaded build, and when building for the `Stable ABI <stable-abi>`, this macro expands to:
>
> ```
> {
>     PyCriticalSection _py_cs;
>     PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
> ```
>
> In the default build, this macro expands to `{`.
>
> 3.13

> “ Locks the mutex *m* and begins a critical section.
>
> In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

Note that unlike `Py_BEGIN_CRITICAL_SECTION`, there is no cast for the argument of the macro - it must be a `PyMutex` pointer.

On the default build, this macro expands to `{`.

3.14

> Ends the critical section and releases the per-object lock.
>
> In the free-threaded build, and when building for the `Stable ABI <stable-abi>`, this macro expands to:
>
> ```
> PyCriticalSection_End(&_py_cs);
> }
> ```
>
> In the default build, this macro expands to `}`.
>
> 3.13

> Acquires the per-object locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.
>
> In the free-threaded build, this macro expands to:
>
> ```
> {
>     PyCriticalSection2 _py_cs2;
>     PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
> ```
>
> In the default build, this macro expands to `{`.
>
> 3.13

>

Locks the mutexes *m1* and *m2* and begins a critical section.

In the free-threaded build, and when building for the `Stable ABI <stable-abi>`, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

Note that unlike `Py_BEGIN_CRITICAL_SECTION2`, there is no cast for the arguments of the macro - they must be `PyMutex` pointers.

On the default build, this macro expands to `{`.

3.14

> Ends the critical section and releases the per-object locks.
>
> In the free-threaded build, and when building for the `Stable ABI <stable-abi>`, this macro expands to:
>
> ```
> PyCriticalSection2_End(&_py_cs2);
> }
> ```
>
> In the default build, this macro expands to `}`.
>
> 3.13

Low-level critical section API

The following functions and structs are exposed for cases where C macros are not available.

Legacy locking APIs

These APIs are obsolete since Python 3.13 with the introduction of `PyMutex`.

3.15 These APIs are now a simple wrapper around `PyMutex`.

> A pointer to a mutual exclusion lock.

> A `soft deprecated` alias to `long long`, the integer type used for the timeout argument of `PyThread_acquire_lock_timed`.
>
> next

> The maximum timeout, in microseconds, supported by `PyThread_acquire_lock_timed`.
>
> > [!NOTE] This differs from `_thread.TIMEOUT_MAX`, which is expressed in seconds.

> The result of acquiring a lock with a timeout.
>
> > Failed to acquire the lock.
>
> > The lock was successfully acquired.
>
> > The lock was interrupted by a signal.

> Allocate a new lock.
>
> On success, this function returns a lock; on failure, this function returns `0` without an exception set.
>
> The caller does not need to hold an `attached thread state`.

3.15 This function now always uses `PyMutex`. In prior versions, this would use a lock provided by the operating system.

Destroy *lock*. The lock should not be held by any thread when calling this.

The caller does not need to hold an `attached thread state`.

Acquire *lock* with a timeout.

This will wait for *microseconds* microseconds to acquire the lock. If the timeout expires, this function returns `PY_LOCK_FAILURE`. If *microseconds* is `-1`, this will wait indefinitely until the lock has been released.

If *intr_flag* is `1`, acquiring the lock may be interrupted by a signal, in which case this function returns `PY_LOCK_INTR`. Upon interruption, it's generally expected that the caller makes a call to `Py_MakePendingCalls` to propagate an exception to Python code.

If the lock is successfully acquired, this function returns `PY_LOCK_ACQUIRED`.

The caller does not need to hold an `attached thread state`.

Acquire *lock*.

If *waitflag* is `1` and another thread currently holds the lock, this function will wait until the lock can be acquired and will always return `1`.

If *waitflag* is `0` and another thread holds the lock, this function will not wait and instead return `0`. If the lock is not held by any other thread, then this function will acquire it and return `1`.

Unlike `PyThread_acquire_lock_timed`, acquiring the lock cannot be interrupted by a signal.

The caller does not need to hold an `attached thread state`.

Release *lock*. If *lock* is not held, then this function issues a fatal error.

> The caller does not need to hold an `attached thread state`.

□□□ #2
□ □□□ □□□ 2026-10-31 16:54:29 UTC
□ □□□ □□□ 2026-10-31 16:54:31 UTC