Thread states and the global interpreter lock

- Detaching the thread state from extension code
  - APIs
- Using the C API from foreign threads
  - Reusing a thread state across repeated calls
  - Attaching/detaching thread states
- GIL-state APIs
- Cautions about fork()
- High-level APIs
- Low-level APIs
- Asynchronous notifications
- Operating system thread APIs

single: global interpreter lock single: interpreter lock single: lock, interpreter

Unless on a `free-threaded build` of `CPython`, the Python interpreter is generally not thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the `global interpreter lock` or `GIL`, that must be held by a thread before accessing Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

As such, only a thread that holds the GIL may operate on Python objects or invoke Python's C API.

single: setswitchinterval (in module sys)

In order to emulate concurrency, the interpreter regularly tries to switch threads between bytecode instructions (see `sys.setswitchinterval`). This is why locks are also necessary for thread-safety in pure-Python code.

Additionally, the global interpreter lock is released around blocking I/O operations, such as reading or writing to a file. From the C API, this is done by `detaching the thread state <detaching-thread-state>`.

single: PyThreadState (C type)

The Python interpreter keeps some thread-local information inside a data structure called `PyThreadState`, known as a `thread state`. Each thread has a thread-local pointer to a `PyThreadState` ; a thread state referenced by this pointer is considered to be `attached <attached thread state>`.

A thread can only have one `attached thread state` at a time. An attached thread state is typically analogous with holding the GIL, except on free-threaded builds. On builds with the GIL enabled, attaching a thread state will block until the GIL can be acquired. However, even on builds with the GIL disabled, it is still required to have an attached thread state, as the interpreter needs to keep track of which threads may access Python objects.

> Even on the free-threaded build, attaching a thread state may block, as the GIL can be re-enabled or threads might be temporarily suspended (such as during a garbage collection).

Generally, there will always be an attached thread state when using Python's C API, including during embedding and when implementing methods, so it's uncommon to need to set up a thread state on your own. Only in some specific cases, such as in a `Py_BEGIN_ALLOW_THREADS` block or in a fresh thread, will the thread not have an attached thread state. If uncertain, check if `PyThreadState_GetUnchecked` returns `NULL`.

If it turns out that you do need to create a thread state, it is recommended to use `PyThreadState_Ensure` or `PyThreadState_EnsureFromView`, which will manage the thread state for you.

Detaching the thread state from extension code

Most extension code manipulating the `thread state` has the following simple structure:

```
Save the thread state in a local variable.
... Do some blocking I/O operation ...
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

single: Py_BEGIN_ALLOW_THREADS (C macro) single: Py_END_ALLOW_THREADS (C macro)

The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

single: PyEval_RestoreThread (C function) single: PyEval_SaveThread (C function)

Here is how these functions work:

The attached thread state implies that the GIL is held for the interpreter. To detach it, `PyEval_SaveThread` is called and the result is stored in a local variable.

By detaching the thread state, the GIL is released, which allows other threads to attach to the interpreter and execute while the current thread performs blocking I/O. When the I/O operation is complete, the old thread state is reattached by calling `PyEval_RestoreThread`, which will wait until the GIL can be acquired.

> Performing blocking I/O is the most common use case for detaching the thread state, but it is also useful to call it over long-running native code that doesn't need access to Python objects or Python's C API. For example, the standard `zlib` and `hashlib` modules detach the `thread state <attached thread state>` when compressing or hashing data.

On a `free-threaded build`, the `GIL` is usually out of the question, but **detaching the thread state is still required**, because the interpreter periodically needs to block all threads to get a consistent view of Python objects without the risk of race conditions. For example, CPython currently suspends all threads for a short period of time while running the garbage collector.

> Detaching the thread state can lead to unexpected behavior during interpreter finalization. See `cautions-regarding-runtime-finalization` for more details.

APIs

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

> These macros are still necessary on the `free-threaded build` to prevent deadlocks.

> This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro.

> This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro.

> This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace.

> This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to `Py_BEGIN_ALLOW_THREADS` without the opening brace and variable declaration.

Using the C API from foreign threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated with them, However, when a thread is created from native code (for example, by a third-party library with its own thread management), it doesn't hold an attached thread state.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the

interpreter by creating a new thread state and attaching it.

The easiest way to do this is through `PyThreadState_Ensure` or `PyThreadState_EnsureFromView`.

> These functions require an argument pointing to the desired interpreter; such a pointer can be acquired via a call to `PyInterpreterGuard_FromCurrent` (for `PyThreadState_Ensure`) or `PyInterpreterView_FromCurrent` (for `PyThreadState_EnsureFromView`) from the function that creates the thread. If no pointer is available (such as when the given native thread library doesn't provide a data argument), `PyInterpreterView_FromMain` can be used to get a view for the main interpreter, but note that this will make the code incompatible with subinterpreters.

For example:

```
// The return value of PyInterpreterGuard_FromCurrent() from the
// function that created this thread.
PyInterpreterGuard *guard = thread_data->guard;

// Create a new thread state for the interpreter.
PyThreadStateToken *token = PyThreadState_Ensure(guard);
if (token == NULL) {
   PyInterpreterGuard_Close(guard);
   return;
}

// We have a valid thread state -- perform Python actions here.
result = CallSomeFunction();
// Evaluate result or handle exceptions.

// Release the thread state. No calls to the C API are allowed beyond this
// point.
PyThreadState_Release(token);
PyInterpreterGuard_Close(guard);
```

Keep in mind that calling `PyThreadState_Ensure` might not always create a new thread state, and calling `PyThreadState_Release` might not always detach it. These functions may reuse an existing attached thread state, or may re-attach a thread state that was previously attached for the current thread.

Reusing a thread state across repeated calls

Creating and destroying a `PyThreadState` is not free, and is more expensive on a `free-threaded build`. A foreign thread that calls into the interpreter many times -- for example, a worker thread in a native thread pool -- should avoid creating a fresh thread state on every entry and destroying it on every exit. Instead, set up one thread state when the thread starts (or lazily on its first call into Python), attach and detach it around each call, and tear it down once when the thread exits.

Manage the thread state explicitly with `PyThreadState_New`, attaching and detaching it with `PyEval_RestoreThread` and `PyEval_SaveThread`. This happens in three distinct phases, at different points in the thread's life.

When the thread starts, create one thread state for it. `interp` is the target interpreter, captured by the code that created this thread while it held an attached thread state (for example via `PyInterpreterState_Get`):

```
PyThreadState *tstate = PyThreadState_New(interp);
```

Then, on each call into Python -- which may happen many times over the thread's life -- attach the thread state, make the Python C API calls that require it, and detach again so the thread does not hold the GIL while off doing non-Python work:

```
PyEval_RestoreThread(tstate);
result = CallSomeFunction();  /* your Python C API calls go here */
PyEval_SaveThread();
```

When the thread is finished calling into Python, destroy the thread state once:

```
PyEval_RestoreThread(tstate);
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

The general-purpose entry points for calling in from a foreign thread -- `PyThreadState_Ensure` and the older `PyGILState_Ensure` -- do *not* guarantee a persistent thread state: their thread-state lifetime is deliberately implementation-defined, so a matched acquire/release pair may create and destroy a thread state each time. Use `PyThreadState_New`, as shown here, whenever you specifically want to reuse one thread state across calls.

The code that created the foreign thread must arrange for the shutdown sequence to run before the thread exits, and before `Py_FinalizeEx` is called. If interpreter finalization begins first, the shutdown `PyEval_RestoreThread` call will hang the thread rather than return (see `cautions-`

`regarding-runtime-finalization`). If the thread exits without running the shutdown sequence, the thread state is leaked for the remainder of the process.

Attaching/detaching thread states

> Ensure that the thread has an attached thread state for the interpreter protected by *guard*, and thus can safely invoke that interpreter.
>
> It is OK to call this function if the thread already has an attached thread state, as long as there is a subsequent call to `PyThreadState_Release` that matches this one (meaning that "nested" calls to this function are permitted).
>
> The function's effect (if any) will be reversed by the matching call to `PyThreadState_Release`.
>
> On error, this function returns `NULL` *without* an exception set. Do not call `!PyThreadState_Release` in this case.
>
> On success, this function returns a pointer value that must be passed to the matching call to `!PyThreadState_Release`.
>
> The conditions in which this function creates a new `thread state` are considered unstable and implementation-dependent. If you need to control the exact lifetime of a thread state, consider using `PyThreadState_New`. However, do not avoid this function solely on the basis that the lifetime of the thread state may be inconsistent across versions; changes to this function will be done with caution and in a backwards-compatible manner. In particular, the saving of thread-local variables and similar state will be retained across Python versions.
>
> The exact behavior of whether this function creates a new thread state is described below, but be aware that this may change in the future.
>
> First, this function checks if an attached thread state is present. If there is, this function then checks if the interpreter of that thread state matches the interpreter guarded by *guard*. If that is the case, this function simply marks the thread state as being used by a `PyThreadState_Ensure` call and returns.
>
> If there is no attached thread state, then this function checks if any thread state has been used by the current OS thread. (This is returned by `PyGILState_GetThisThreadState`.) If there was, then this function checks if that thread state's interpreter matches *guard*. If it does, it is re-attached and marked as used.

Otherwise, if both of the above cases fail, a new thread state is created for *guard*. It is then attached and marked as owned by `PyThreadState_Ensure`.

3.15

Get an attached thread state for the interpreter referenced by *view*.

The behavior and return value are the same as for `PyThreadState_Ensure`; additionally, if the function succeeds, the interpreter referenced by *view* will be implicitly guarded. The guard will be released upon the corresponding `PyThreadState_Release` call.

3.15

Undo a `PyThreadState_Ensure` or `PyThreadState_EnsureFromView` call.

This must be called exactly once for each successful *Ensure* call, with *token* set to that call's return value.

The state that was attached before the corresponding *Ensure* call (if any) will be attached when `PyThreadState_Release` returns.

The exact behavior of whether this function deletes a thread state is considered unstable and implementation-dependent.

Currently, this function will decrement an internal counter on the attached thread state. If this counter ever reaches below zero, this function emits a fatal error (via `Py_FatalError`).

If the attached thread state is owned by `PyThreadState_Ensure`, then the attached thread state will be deallocated and deleted upon the internal counter reaching zero. Otherwise, nothing happens when the counter reaches zero.

3.15

An opaque token retrieved from a `PyThreadState_Ensure` call and passed to a corresponding `PyThreadState_Release` call.

GIL-state APIs

The following APIs are generally not compatible with subinterpreters and will hang the process during interpreter finalization (see `cautions-regarding-runtime-finalization`). As such, these APIs were `soft deprecated` in Python 3.15 in favor of the `new APIs <c-api-foreign-threads>`.

> The type of the value returned by `PyGILState_Ensure` and passed to `PyGILState_Release`.
>
> > The GIL was already held when `PyGILState_Ensure` was called.
> >
> > The GIL was not held when `PyGILState_Ensure` was called.

> Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the `attached thread state`. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release`. In general, other thread-related APIs may be used between `PyGILState_Ensure` and `PyGILState_Release` calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.
>
> The return value is an opaque "handle" to the `attached thread state` when `PyGILState_Ensure` was called, and must be passed to `PyGILState_Release` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure` must save the handle for its call to `PyGILState_Release`.
>
> When the function returns, there will be an `attached thread state` and the thread will be able to call arbitrary Python code.
>
> This function has no way to return an error. As such, errors are either fatal (that is, they send `SIGABRT` and crash the process; see `Py_FatalError`), or the thread will be permanently blocked (such as during interpreter finalization).

> > [!WARNING] Calling this function when the interpreter is finalizing will infinitely hang the thread, which may cause deadlocks. `cautions-regarding-runtime-finalization` for more details.
> >
> > In addition, this function generally does not work with subinterpreters when used from foreign threads, because this function has no way of knowing which interpreter created the thread (and as such, will implicitly pick the main interpreter).
>
> 3.14 Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.
>
> 3.15 Use `PyThreadState_Ensure` or `PyThreadState_EnsureFromView` instead.

> Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding `PyGILState_Ensure` call (but generally this state will be unknown to the caller, hence the use of the GIL-state API).
>
> Every call to `PyGILState_Ensure` must be matched by a call to `PyGILState_Release` on the same thread.
>
> 3.15 Use `PyThreadState_Release` instead.

> Get the `thread state` that was most recently `attached <attached thread state>` for this thread. (If the most recent thread state has been deleted, this returns `NULL`.)
>
> If the caller has an attached thread state, it is returned.
>
> In other terms, this function returns the thread state that will be used by `PyGILState_Ensure`. If this returns `NULL`, then `PyGILState_Ensure` will create a new thread state.
>
> This function cannot fail.
>
> 3.15 Use `PyThreadState_Get` or `PyThreadState_GetUnchecked` instead.

> Return `1` if the current thread has an `attached thread state` that matches the thread state returned by `PyGILState_GetThisThreadState`. If the caller has no attached thread state or it otherwise doesn't match, then this returns `0`.
>
> If the current Python process has ever created a subinterpreter, this function will *always* return `1`.
>
> This is mainly a helper/diagnostic function.
>
> 3.4
>
> 3.15 Use `PyThreadState_GetUnchecked() != NULL` instead.

Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C `fork` call. On most systems with `fork`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

The fact that only the "current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any `lock-objects` in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `!pthread_atfork` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork` directly rather than through `os.fork` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. `PyOS_AfterFork_Child` tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly, which `os.fork` does. This means finalizing all other `PyThreadState` objects belonging to the current interpreter and all other `PyInterpreterState` objects. Due to this and the special nature of the `"main" interpreter <sub-interpreter-support>`, `fork` should only be called in that interpreter's "main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec` will be called immediately after.

High-level APIs

These are the most commonly used types and functions when writing multi-threaded C extensions.

> This data structure represents the state of a single thread. The only public data member is:
>
> > This thread's interpreter state.

> Detach the `attached thread state` and return it. The thread will have no `thread state` upon returning.

> Set the `attached thread state` to *tstate*. The passed `thread state` **should not** be `attached <attached thread state>`, otherwise deadlock ensues. *tstate* will be attached upon returning.
>
> > [!NOTE] Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to `cautions-regarding-runtime-finalization` for more details.
>
> 3.14 Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

> Return the `attached thread state`. If the thread has no attached thread state, (such as when inside of `Py_BEGIN_ALLOW_THREADS` block), then this issues a fatal error (so that the caller needn't check for `NULL`).
>
> See also `PyThreadState_GetUnchecked`.

> Similar to `PyThreadState_Get`, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.
>
> 3.13 In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

> Set the `attached thread state` to *tstate*, and return the `thread state` that was attached prior to calling.
>
> This function is safe to call without an `attached thread state`; it will simply return `NULL` indicating that there was no prior thread state.
>
> `PyEval_ReleaseThread`
>
> > [!NOTE] Similar to `PyGILState_Ensure`, this function will hang the thread if the runtime is finalizing.

Low-level APIs

> Create a new thread state object belonging to the given interpreter object. An `attached thread state` is not needed.

> Reset all information in a `thread state` object. *tstate* must be `attached <attached thread state>`
>
> 3.9 This function now calls the `!PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete`.
>
> 3.13 The `!PyThreadState.on_delete` callback was removed.

> Destroy a `thread state` object. *tstate* should not be `attached <attached thread state>` to any thread. *tstate* must have been reset with a previous call to `PyThreadState_Clear`.

> Detach the `attached thread state` (which must have been reset with a previous call to `PyThreadState_Clear`) and then destroy it.
>
> No `thread state` will be `attached <attached thread state>` upon returning.

Get the current frame of the Python thread state *tstate*.

Return a `strong reference`. Return `NULL` if no frame is currently executing.

See also `PyEval_GetFrame`.

tstate must not be `NULL`, and must be `attached <attached thread state>`.

3.9

Get the unique `thread state` identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be `attached <attached thread state>`.

3.9

Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`, and must be `attached <attached thread state>`.

3.9

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the `PyThreadState_LeaveTracing` function.

3.11

Resume tracing and profiling in the Python thread state *tstate* suspended by the `PyThreadState_EnterTracing` function.

See also `PyEval_SetTrace` and `PyEval_SetProfile` functions.

3.11

Set the stack protection start address and stack protection size of a Python thread state.

On success, return `0`. On failure, set an exception and return `-1`.

CPython implements `recursion control <recursion>` for C code by raising `RecursionError` when it notices that the machine execution stack is close to overflow. See for example the `Py_EnterRecursiveCall` function. For this, it needs to know the location of the current thread's stack, which it normally gets from the operating system. When the stack is changed, for example using context switching techniques like the Boost library's `boost::context`, you must call `~PyUnstable_ThreadState_SetStackProtection` to inform CPython of the change.

Call `~PyUnstable_ThreadState_SetStackProtection` either before or after changing the stack. Do not call any other Python C API between the call and the stack change.

See `PyUnstable_ThreadState_ResetStackProtection` for undoing this operation.

3.15

Reset the stack protection start address and stack protection size of a Python thread state to the operating system defaults.

See `PyUnstable_ThreadState_SetStackProtection` for an explanation.

3.15

Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to store a state in the dictionary. It is okay to call this function when no `thread state` is `attached <attached thread state>`. If this function returns `NULL` and no exception has been raised, then the caller should assume no thread state is attached.

`Attach <attached thread state>` *tstate* to the current thread, which must not be `NULL` or already `attached <attached thread state>`.

The calling thread must not already have an `attached thread state`.

> [!NOTE] Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to `cautions-regarding-runtime-finalization` for more details.

3.8 Updated to be consistent with `PyEval_RestoreThread`, `Py_END_ALLOW_THREADS`, and `PyGILState_Ensure`, and terminate the current thread if called while the interpreter is finalizing.

3.14 Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

`PyEval_RestoreThread` is a higher-level function which is always available (even when threads have not been initialized).

> Detach the `attached thread state`. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the `attached thread state` --- if it isn't, a fatal error is reported.
>
> `PyEval_SaveThread` is a higher-level function which is always available (even when threads have not been initialized).

Asynchronous notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

> Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.
>
> When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:
>
> - on a `bytecode` boundary;

- with the main thread holding an `attached thread state` (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the `thread state <attached thread state>` is detached.

This function doesn't need an `attached thread state`. However, to call this function in a subinterpreter, the caller must have an `attached thread state`. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> [!WARNING] This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use `PyThreadState_EnsureFromView`.

3.1

3.9 If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

3.12 This function now always schedules *func* to be run in the main interpreter.

> Execute all pending calls. This is usually executed automatically by the interpreter.
>
> This function returns `0` on success, and returns `-1` with an exception set on failure.
>
> If this is not called in the main thread of the main interpreter, this function does nothing and returns `0`. The caller must hold an `attached thread state`.
>
> 3.1

> 3.12 This function only runs pending calls in the main interpreter.

> Schedule an exception to be raised asynchronously in a thread. If the thread has a previously scheduled exception, it is overwritten.
>
> The *id* argument is the thread id of the target thread, as returned by `PyThread_get_thread_ident`. *exc* is the class of the exception to be raised, or `NULL` to clear the pending exception (if any).
>
> Return the number of affected thread states. This is normally `1` if *id* is found, even when no change was made (the given *exc* was already pending, or *exc* is `NULL` but no exception is pending). If the thread id isn't found, return `0`. This raises no exceptions.
>
> To prevent naive misuse, you must write your own C extension to call this. This function must be called with an `attached thread state`. This function does not `steal` any references to *exc*. This function does not necessarily interrupt system calls such as `~time.sleep`.
>
> 3.7 The type of the *id* parameter changed from `long` to `unsigned long`.

Operating system thread APIs

> Sentinel value for an invalid thread ID.
>
> This is currently equivalent to `(unsigned long)-1`.

> Start function *func* in a new thread with argument *arg*. The resulting thread is not intended to be joined.
>
> *func* must not be `NULL`, but *arg* may be `NULL`.

> On success, this function returns the identifier of the new thread; on failure, this returns `PYTHREAD_INVALID_THREAD_ID`.
>
> The caller does not need to hold an `attached thread state`.

> “ Return the identifier of the current thread, which will never be zero.
>
> This function cannot fail, and the caller does not need to hold an `attached thread state`.
>
> `threading.get_ident` and `threading.Thread.ident` expose this identifier to Python.

> “ Get general information about the current thread in the form of a `struct sequence <struct-sequence-objects>` object. This information is accessible as `sys.thread_info` in Python.
>
> On success, this returns a new `strong reference` to the thread information; on failure, this returns `NULL` with an exception set.
>
> The caller must hold an `attached thread state`.

> “ This macro is defined when the system supports native thread IDs.

> “ Get the native identifier of the current thread as it was assigned by the operating system's kernel, which will never be less than zero.
>
> This function is only available when `PY_HAVE_THREAD_NATIVE_ID` is defined.
>
> This function cannot fail, and the caller does not need to hold an `attached thread state`.
>
> `threading.get_native_id`

> Terminate the current thread. This function is generally considered unsafe and should be avoided. It is kept solely for backwards compatibility.
>
> This function is only safe to call if all functions in the full call stack are written to safely allow it.
>
> > [!WARNING] If the current system uses POSIX threads (also known as "pthreads"), this calls `pthread_exit(3)`, which attempts to unwind the stack and call C++ destructors on some libc implementations. However, if a `noexcept` function is reached, it may terminate the process. Other systems, such as macOS, do unwinding.
> >
> > On Windows, this function calls `_endthreadex()`, which kills the thread without calling C++ destructors.
> >
> > In any case, there is a risk of corruption on the thread's stack.
>
> 3.14

> Initialize `PyThread*` APIs. Python executes this function automatically, so there's little need to call it from an extension module.

> Set the stack size of the current thread to *size* bytes.
>
> This function returns `0` on success, `-1` if *size* is invalid, or `-2` if the system does not support changing the stack size. This function does not set exceptions.
>
> The caller does not need to hold an `attached thread state`.

> Return the stack size of the current thread in bytes, or `0` if the system's default stack size is in use.
>
> The caller does not need to hold an `attached thread state`.

□□□ #2
□ □□□ □□□ 2026-10-31 16:54:39 UTC
□ □□□ □□□ 2026-10-31 16:54:42 UTC