18.5.7. Synchronization primitives

Source code: Lib/asyncio/locks.py

Locks:

- `Lock`
- `Event`
- `Condition`

Semaphores:

- `Semaphore`
- `BoundedSemaphore`

asyncio lock API was designed to be close to classes of the `threading` module (`Lock`, `Event`, `Condition`, `Semaphore`, `BoundedSemaphore`), but it has no *timeout* parameter. The `asyncio.wait_for()` function can be used to cancel a task after a timeout.

18.5.7.1. Locks

18.5.7.1.1. Lock

class `asyncio.`**`Lock`**(*, *loop=None*)

Primitive lock objects.

A primitive lock is a synchronization primitive that is not owned by a particular coroutine when locked. A primitive lock is in one of two states, 'locked' or 'unlocked'.

It is created in the unlocked state. It has two basic methods, `acquire()` and `release()`. When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another coroutine changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a `RuntimeError` will be raised.

When more than one coroutine is blocked in acquire() waiting for the state to turn to unlocked, only one coroutine proceeds when a release() call resets the state to unlocked; first coroutine which is blocked in acquire() is being processed.

`acquire()` is a coroutine and should be called with `yield from`.

Locks also support the context management protocol. `(yield from lock)` should be used as the context manager expression.

This class is not thread safe.

Usage:

```
lock = Lock()
...
yield from lock
try:
    ...
finally:
    lock.release()
```

Context manager usage:

```
lock = Lock()
...
with (yield from lock):
    ...
```

Lock objects can be tested for locking state:

```
if not lock.locked():
    yield from lock
else:
    # lock is acquired
    ...
```

locked()

Return `True` if the lock is acquired.

coroutine **acquire**()

Acquire a lock.

This method blocks until the lock is unlocked, then sets it to locked and returns `True`.

This method is a coroutine.

release()

Release a lock.

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

There is no return value.

18.5.7.1.2. Event

class `asyncio.`**`Event`**(*, *loop=None*)

An Event implementation, asynchronous equivalent to `threading.Event`.

Class implementing event objects. An event manages a flag that can be set to true with the `set()` method and reset to false with the `clear()` method. The `wait()` method blocks until the flag is true. The flag is initially false.

This class is not thread safe.

`clear`()

Reset the internal flag to false. Subsequently, coroutines calling `wait()` will block until `set()` is called to set the internal flag to true again.

`is_set`()

Return `True` if and only if the internal flag is true.

`set`()

Set the internal flag to true. All coroutines waiting for it to become true are awakened. Coroutine that call `wait()` once the flag is true will not block at all.

coroutine **`wait`**()

Block until the internal flag is true.

If the internal flag is true on entry, return `True` immediately. Otherwise, block until another coroutine calls `set()` to set the flag to true, then return `True`.

This method is a coroutine.

18.5.7.1.3. Condition

class `asyncio.`**`Condition`**(*lock=None*, *, *loop=None*)

A Condition implementation, asynchronous equivalent to `threading.Condition`.

This class implements condition variable objects. A condition variable allows one or more coroutines to wait until they are notified by another coroutine.

If the *lock* argument is given and not `None`, it must be a `Lock` object, and it is used as the underlying lock. Otherwise, a new `Lock` object is created and used as the underlying lock.

This class is not thread safe.

coroutine **`acquire`**()

Acquire the underlying lock.

This method blocks until the lock is unlocked, then sets it to locked and returns `True`.

This method is a coroutine.

`notify`(*n=1*)

By default, wake up one coroutine waiting on this condition, if any. If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method wakes up at most *n* of the coroutines waiting for the condition variable; it is a no-op if no coroutines are waiting.

> **Note:** An awakened coroutine does not actually return from its `wait()` call until it can reacquire the lock. Since `notify()` does not release the lock, its caller should.

`locked`()

Return `True` if the underlying lock is acquired.

`notify_all`()

Wake up all coroutines waiting on this condition. This method acts like `notify()`, but wakes up all waiting coroutines instead of one. If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

`release`()

Release the underlying lock.

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a `RuntimeError` is raised.

There is no return value.

coroutine **`wait`**()

Wait until notified.

If the calling coroutine has not acquired the lock when this method is called, a `RuntimeError` is raised.

This method releases the underlying lock, and then blocks until it is awakened by a `notify()` or `notify_all()` call for the same condition variable in another coroutine. Once awakened, it re-acquires the lock and returns `True`.

This method is a coroutine.

coroutine **`wait_for`**(*predicate*)

Wait until a predicate becomes true.

The predicate should be a callable which result will be interpreted as a boolean value. The final predicate value is the return value.

This method is a coroutine.

18.5.7.2. Semaphores

18.5.7.2.1. Semaphore

class `asyncio.`**`Semaphore`**(*value=1*, ***, *loop=None*)

A Semaphore implementation.

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other coroutine calls `release()`.

Semaphores also support the context management protocol.

The optional argument gives the initial value for the internal counter; it defaults to `1`. If the value given is less than `0`, `ValueError` is raised.

This class is not thread safe.

coroutine **acquire**()

Acquire a semaphore.

If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately. If it is zero on entry, block, waiting until some other coroutine has called `release()` to make it larger than `0`, and then return `True`.

This method is a coroutine.

locked()

Returns `True` if semaphore can not be acquired immediately.

release()

Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another coroutine is waiting for it to become larger than zero again, wake up that coroutine.

18.5.7.2.2. BoundedSemaphore

class `asyncio.`**BoundedSemaphore**(*value=1*, *, *loop=None*)

A bounded semaphore implementation. Inherit from `Semaphore`.

This raises `ValueError` in `release()` if it would increase the value above the initial value.