asyncio-sync

- Synchronization Primitives
  - Lock
  - Event
  - Condition
  - Semaphore
  - BoundedSemaphore
  - Barrier

asyncio

Synchronization Primitives

Source code: `Lib/asyncio/locks.py`

asyncio synchronization primitives are designed to be similar to those of the `threading` module with two important caveats:

- asyncio primitives are not thread-safe, therefore they should not be used for OS thread synchronization (use `threading` for that);
- methods of these synchronization primitives do not accept the *timeout* argument; use the `asyncio.wait_for` function to perform operations with timeouts.

asyncio has the following basic synchronization primitives:

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`
- `Barrier`

Lock

Implements a mutex lock for asyncio tasks. Not thread-safe.

An asyncio lock can be used to guarantee exclusive access to a shared resource.

The preferred way to use a Lock is an `async with` statement:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

which is equivalent to:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

3.10 Removed the *loop* parameter.

acquire()

Acquire the lock.

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

When more than one coroutine is blocked in `acquire` waiting for the lock to be unlocked, only one coroutine eventually proceeds.

Acquiring a lock is *fair*: the coroutine that proceeds will be the first coroutine that started waiting on the lock.

release()

Release the lock.

When the lock is *locked*, reset it to *unlocked* and return.

If the lock is *unlocked*, a `RuntimeError` is raised.

locked()

Return `True` if the lock is *locked*.

Event

An event object. Not thread-safe.

An asyncio event can be used to notify multiple asyncio tasks that some event has happened.

An Event object manages an internal flag that can be set to *true* with the `~Event.set` method and reset to *false* with the `clear` method. The `~Event.wait` method blocks until the flag is set to *true*. The flag is set to *false* initially.

3.10 Removed the *loop* parameter.

Example:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()
```

```
    # Wait until the waiter task is finished.
    await waiter_task

asyncio.run(main())
```

wait()

Wait until the event is set.

If the event is set, return `True` immediately. Otherwise block until another task calls `~Event.set`.

set()

Set the event.

All tasks waiting for event to be set will be immediately awakened.

clear()

Clear (unset) the event.

Subsequent tasks awaiting on `~Event.wait` will now block until the `~Event.set` method is called again.

is_set()

Return `True` if the event is set.

Condition

A Condition object. Not thread-safe.

An asyncio condition primitive can be used by a task to wait for some event to happen and then get exclusive access to a shared resource.

In essence, a Condition object combines the functionality of an `Event` and a `Lock`. It is possible to have multiple Condition objects share one Lock, which allows coordinating exclusive access to a shared resource between different tasks interested in particular states of that shared resource.

The optional *lock* argument must be a `Lock` object or `None`. In the latter case a new Lock object is created automatically.

3.10 Removed the *loop* parameter.

The preferred way to use a Condition is an `async with` statement:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

which is equivalent to:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

acquire()

Acquire the underlying lock.

This method waits until the underlying lock is *unlocked*, sets it to *locked* and returns `True`.

notify(n=1)

Wake up *n* tasks (1 by default) waiting on this condition. If fewer than *n* tasks are waiting they are all awakened.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a `RuntimeError` error is raised.

locked()

Return `True` if the underlying lock is acquired.

notify_all()

Wake up all tasks waiting on this condition.

This method acts like `notify`, but wakes up all waiting tasks.

The lock must be acquired before this method is called and released shortly after. If called with an *unlocked* lock a `RuntimeError` error is raised.

release()

Release the underlying lock.

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

wait()

Wait until notified.

If the calling task 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. Once awakened, the Condition re-acquires its lock and this method returns `True`.

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to `~Condition.wait` again. For this reason, you may prefer to use `~Condition.wait_for` instead.

wait_for(predicate)

Wait until a predicate becomes *true*.

The predicate must be a callable which result will be interpreted as a boolean value. The method will repeatedly `~Condition.wait` until the predicate evaluates to *true*. The final value is the return value.

Semaphore

A Semaphore object. Not thread-safe.

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 task calls `release`.

The optional *value* argument gives the initial value for the internal counter (`1` by default). If the given value is less than `0` a `ValueError` is raised.

3.10 Removed the *loop* parameter.

The preferred way to use a Semaphore is an `async with` statement:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

which is equivalent to:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

acquire()

Acquire a semaphore.

If the internal counter is greater than zero, decrement it by one and return `True` immediately. If it is zero, wait until a `release` is called and return `True`.

locked()

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

release()

Release a semaphore, incrementing the internal counter by one. Can wake up a task waiting to acquire the semaphore.

Unlike `BoundedSemaphore`, `Semaphore` allows making more `release()` calls than `acquire()` calls.

BoundedSemaphore

A bounded semaphore object. Not thread-safe.

Bounded Semaphore is a version of `Semaphore` that raises a `ValueError` in `~Semaphore.release` if it increases the internal counter above the initial *value*.

3.10 Removed the *loop* parameter.

Barrier

A barrier object. Not thread-safe.

A barrier is a simple synchronization primitive that allows to block until *parties* number of tasks are waiting on it. Tasks can wait on the `~Barrier.wait` method and would be blocked until the specified number of tasks end up waiting on `~Barrier.wait`. At that point all of the waiting tasks would unblock simultaneously.

`async with` can be used as an alternative to awaiting on `~Barrier.wait`.

The barrier can be reused any number of times.

Example:

```
async def example_barrier():
   # barrier with 3 parties
   b = asyncio.Barrier(3)

   # create 2 new waiting tasks
   asyncio.create_task(b.wait())
   asyncio.create_task(b.wait())

   await asyncio.sleep(0)
   print(b)

   # The third .wait() call passes the barrier
```

```
    await b.wait()
    print(b)
    print("barrier passed")

    await asyncio.sleep(0)
    print(b)

asyncio.run(example_barrier())
```

Result of this example is:

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

3.11

wait()

Pass the barrier. When all the tasks party to the barrier have called this function, they are all unblocked simultaneously.

When a waiting or blocked task in the barrier is cancelled, this task exits the barrier which stays in the same state. If the state of the barrier is "filling", the number of waiting task decreases by 1.

The return value is an integer in the range of 0 to `parties-1`, different for each task. This can be used to select a task to do some special housekeeping, e.g.:

```
...
async with barrier as position:
    if position == 0:
        # Only one task prints this
        print('End of *draining phase*')
```

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a task is waiting. It could raise a `CancelledError` if a task is cancelled.

reset()

Return the barrier to the default, empty state. Any tasks waiting on it will receive the `BrokenBarrierError` exception.

If a barrier is broken it may be better to just leave it and create a new one.

abort()

Put the barrier into a broken state. This causes any active or future calls to `~Barrier.wait` to fail with the `BrokenBarrierError`. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

The number of tasks required to pass the barrier.

n_waiting

The number of tasks currently waiting in the barrier while filling.

broken

A boolean that is `True` if the barrier is in the broken state.

BrokenBarrierError

This exception, a subclass of `RuntimeError`, is raised when the `Barrier` object is reset or broken.

3.9

Acquiring a lock using `await lock` or `yield from lock` and/or `with` statement (`with await lock`, `with (yield from lock)`) was removed. Use `async with lock` instead.

□□□ #2
□ □□□ □□□ 2026-10-31 17:03:43 UTC
□ □□□ □□□ 2026-10-31 17:03:45 UTC