Async Sleep Problem

This lesson discusses how we can implement a method similar to asyncio.sleep API.

Async Sleep Problem

Problem

Asyncio already provides us with an API to sleep asynchronously **`asyncio.sleep()`**. In fact, this is one of the most commonly used asyn APIs in introductory examples explaining async.io. The problem at hand is to implement our own coroutine that sleeps asynchronously. The signature of the coroutine is as follows.

```
# Implement the following coroutine where
# sleep_for is defined in seconds
async def asleep(sleep_for):
    pass
```

This makes for an excellent interview problem as it is small enough to be completed in an hour and tricky enough to test a candidate's thought-process.

Solution

The first thought to cross your mind will be to use **`time.sleep()`** API to wait out the requested sleeping time. However, the API is a blocking one

and will block the thread that executes it. Obviously, this rules out invoking the API using the main thread. But it doesn't preclude us from executing this API on a different thread.

This insight leads us to a possible solution. We can create a **`Future`** object and await it in the **`asleep()`** coroutine. The only requirement is now to have another thread resolve the future after **`sleep_for`** seconds have elapsed. The partial solution looks as follows:

First Cut

```
async def asleep(sleep_for):
    future = Future()

    Thread(target=sync_sleep, args=(sleep_for, future)).start
()
    await future


def sync_sleep(sleep_for, future):
    # sleep synchronously
    time.sleep(sleep_for)

    # resolve the future
    future.set_result(None)
```

Let's run our first version in the code widget and see if it works.

```
from threading import Thread
from threading import current_thread
from asyncio import Future
import asyncio
import time


async def asleep(sleep_for):
    future = Future()
    Thread(target=sync_sleep, args=(sleep_for, future)).start()
    await future


def sync_sleep(sleep_for, future):

    # sleep synchronously
    time.sleep(sleep_for)

    # resolve the future
    future.set_result(None)

    print("Sleeping completed in {0}".format(current_thread().getName()), flush=Tru


if __name__ == "__main__":
    start = time.time()
    work = list()
    work.append(asleep(1))

    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(work, return_when=asyncio.ALL_COMPLETED))
    print("main program exiting after running for {0}".format(time.time() - start))
```

Surprisingly, the above program hangs and doesn't complete even though the message from the method **`sync_sleep()`** is printed. Somehow the coroutine **`asleep()`** is never resumed after the future it is awaiting has been resolved. The reason is that **`Future`** isn't thread-safe. Fortunately, asyncio provides a method to execute a coroutine on a given loop in a thread-safe manner. The API is **`run_coroutine_threadsafe()`**.

So we have a way to resolve the future in a thread-safe manner however, we need to do that in yet another coroutine since the API `run_coroutine_threadsafe()` takes in only coroutines. This requires us to slightly modify our `sync_sleep()` method as follows:

```
def sync_sleep(sleep_for, future, loop):
    # sleep synchronously
    time.sleep(sleep_for)

    # define a nested coroutine to resolve the future
    async def sleep_future_resolver():
        # resolve the future
        future.set_result(None)

    asyncio.run_coroutine_threadsafe(sleep_future_resolver(),
 loop)
```

We define a nested coroutine `sleep_future_resolver` that resolves the `Future` object. Also, note that `sync_sleep` now takes in the event loop as a parameter. This should be the same event loop that executed the `asleep()` coroutine in the first place. Changes to `asleep()` coroutine are shown below:

```
async def asleep(sleep_for):
    future = Future()
    # get the current event loop
    current_loop = asyncio.get_running_loop()
    Thread(target=sync_sleep, args=(sleep_for, future, current
_loop)).start()

    await future
```

Let's test our solution in the code widget below:

```
from threading import Thread
from threading import current_thread
from asyncio import Future
import asyncio
import time


async def asleep(sleep_for):
    future = Future()
    current_loop = asyncio.get_event_loop()
    Thread(target=sync_sleep, args=(sleep_for, future, current_loop)).start()

    await future


def sync_sleep(sleep_for, future, loop):
    # sleep synchronously
    time.sleep(sleep_for)

    # define a nested coroutine to resolve the future
    async def sleep_future_resolver():
        # resolve the future
        future.set_result(None)

    asyncio.run_coroutine_threadsafe(sleep_future_resolver(), loop)
    print("Sleeping completed in {0}\n".format(current_thread().getName()), flush=T


if __name__ == "__main__":
    start = time.time()
    work = list()
    work.append(asleep(5))
    work.append(asleep(5))
    work.append(asleep(5))
    work.append(asleep(5))
    work.append(asleep(5))

    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(work, return_when=asyncio.ALL_COMPLETED))
    print("main program exiting after running for {0}".format(time.time() - start))
```

The output shows that sleeping takes place in the threads we spawn and not the main thread. Furthermore, even though we submit the

`asleep()` coroutine five times to sleep for five seconds each but the total runtime of the program is roughly five seconds as it should be if we implemented the solution correctly.

As an exercise consider what would happen if we created five threads and had each thread invoke **`time.sleep()`**, will the program in that case take five or twenty five seconds to complete? Run the code widget below and observe the time taken by thee program to complete.

```
from threading import Thread
from threading import current_thread
import time


def sync_sleep(sleep_for):
    time.sleep(sleep_for)
    print("Sleeping completed in {0}".format(current_thread().getName()))


if __name__ == "__main__":
    start = time.time()

    threads = list()

    for _ in range(0, 5):
        threads.append(Thread(target=sync_sleep, args=(5,)))

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    print("main program exiting after running for {0}".format(time.time() - start))
```

The synchronous sleep test still takes five seconds to complete! You may wonder what is the difference between our asynchronous sleep versus synchronous sleep programs? The answer is the asynchronous sleep call

synchronous sleep programs? The answer is the asynchronous sleep call is non-blocking whereas the synchronous sleep call is blocking.

Internally though, the scheduler on seeing a thread is about to block on a sleep call for five seconds, switches it out for another thread and only resumes executing it after at least five seconds have elapsed.

Back

Web Crawler Example

Next

Chat Server Example

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/async-sleep-problem__asyncio__python-concurrency-for-senior-engineering-interviews)