Asynchronous to Synchronous Problem

A real-life interview question asking to convert asynchronous execution to synchronous execution.

Asynchronous to Synchronous Problem

This is an actual interview question asked at Netflix.

Imagine we have an **`AsyncExecutor`** class that performs some useful task asynchronously via the method **`execute()`**. In addition, the method accepts a function object that acts as a callback and gets invoked after the asynchronous execution is done. The definition for the involved classes is below. The asynchronous work is simulated using sleep. A passed-in call is invoked to let the invoker take any desired action after the asynchronous processing is complete.

Executor Class

```
class AsyncExecutor:

    def work(self, callback):
        # simulate asynchronous work
        time.sleep(5)
        # let the invoker take action in a callback
        callback()

    def execute_async(self, callback):
        Thread(target=self.work, args=(callback,)).start()
```

An example run would be as follows:

```
from threading import Thread
import time


class AsyncExecutor:

    def work(self, callback):
        time.sleep(5)
        callback()

    def execute_async(self, callback):
        Thread(target=self.work, args=(callback,)).start()


def say_hi():
    print("Hi")


if __name__ == "__main__":
    exec = AsyncExecutor()
    exec.execute_async(say_hi)

    print("main thread exiting")
```

Note how the main thread exits before the asynchronous execution is completed. The message "Hi" is printed after the main thread has exited.

Your task is to make the execution synchronous without changing the original classes (imagine that you are given the binaries and not the source code) so that the main thread waits till the asynchronous execution is complete. In other words, the highlighted **line#23** only executes once the asynchronous task is complete.

Solution

The problem here asks us to convert asynchronous code to synchronous code without modifying the original code. The requirement that the main thread should block till the asynchronous execution is complete hints at using some kind of notification/signalling mechanism. The main thread *waits* on something, which is then *signaled* by the asynchronous execution thread. Semaphore is the first thought that may come to your mind for solving this problem and is the right approach.

Since we can't modify the original code, we'll extend a new class `SynchronousExecutor` from the given `AsyncExecutor` class and override the `execute()` method. The trick here is to invoke the original asynchronous implementation using `super().execute()` inside the overridden method. The class would look as follows:

```
class SyncExecutor(AsyncExecutor):

    def __init__(self):
        self.sem = Semaphore(0)

    def work(self, callback):
        super().work(callback)
        self.sem.release()

    def execute(self, callback):
        super().execute(callback)
        self.sem.acquire()
```

We create a **`Semaphore`** object initialized with a count of zero. The main thread that invokes **`execute()`** gets blocked on the semaphore object's **`acquire()`** method. The base class **`AsynExecutor`** spawns another thread that simulates the work being done and then invokes the **`SyncExecutor`** class's **`work()`** method. The **`SyncExecutor`** class overrides the **`work()`** from the base class. It invokes the base class's implementation and then signals the semaphore allowing the main thread to resume execution. All in all, the main thread remains blocked till the asynchronous execution is complete and the passed in callback has been invoked.

The complete code appears in the code widget below. If you run the program, the message from the main thread prints last.

```
from threading import Thread
from threading import Semaphore
import time


class AsyncExecutor:

    def work(self, callback):
        # simulate work
        time.sleep(5)
        # work is done so now invoke callback
        callback()

    def execute(self, callback):
        Thread(target=self.work, args=(callback,)).start()


class SyncExecutor(AsyncExecutor):

    def __init__(self):
        self.sem = Semaphore(0)

    def work(self, callback):
        super().work(callback)
        self.sem.release()

    def execute(self, callback):
        super().execute(callback)
        self.sem.acquire()


def say_hi():
    print("Hi")


if __name__ == "__main__":
    exec = SyncExecutor()
    exec.execute(say_hi)

    print("main thread exiting")
```

However, for this problem, I was asked to solve using a condition variable which is slightly more complex than when using a semaphore.

Remember, with condition variables we always need an associated condition to check for and a lock to acquire. The underlying logic is the same as when using a semaphore. We'll introduce a variable **`is_done`** that will be set to true once asynchronous execution is complete. The main thread will wait on the condition variable until **`is_done`** is set to true and the thread spawned by the **`AsyncExecutor`** will **`notify()`** on the condition variable as well as set **`is_done`** to true. The **`execute()`** will look like as follows:

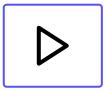
```
    def execute(self, callback):
        super().execute(callback)

        self.cv.acquire()
        while self.is_done is False:
            self.cv.wait()
        self.cv.release()
```

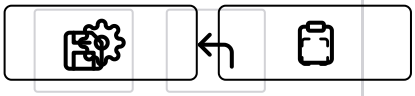
While the **`work()`** will change as follows:

```
    def work(self, callback):
        super().work(callback)

        self.cv.acquire()
        self.cv.notifyAll()
        self.is_done = True
        self.cv.release()
```

The complete code appears below in the code widget and produces the same result as when using a semaphore.

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


class AsyncExecutor:

    def work(self, callback):
        time.sleep(5)
        callback()

    def execute(self, callback):
        Thread(target=self.work, args=(callback,)).start()


class SyncExecutor(AsyncExecutor):

    def __init__(self):
        self.cv = Condition()
        self.is_done = False

    def work(self, callback):
        super().work(callback)

        print("{0} thread notifying".format(current_thread().getName()))
        self.cv.acquire()
        self.cv.notifyAll()
        self.is_done = True
        self.cv.release()

    def execute(self, callback):
        super().execute(callback)

        self.cv.acquire()
        while self.is_done is False:
            self.cv.wait()
        print("{0} thread woken-up".format(current_thread().getName()))
        self.cv.release()


def say_hi():
    print("Hi")


if __name__ == "__main__":
    exec = SyncExecutor()
    exec.execute(say_hi)

    print("main thread exiting")
```





← Back

Barber Shop

Next →

Multithreaded Merge Sort

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/asynchronous-to-synchronous-problem__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)